Software Architecture: Central Concerns, Key Decisions

If the applications software supporting your services and essential business systems, or the software in your products, is becoming bigger and messier, it is time to consider whether software architecture ought to be a core competency of your business. So, it is fair to ask "what is software architecture?"

This paper seeks to answer that question, not in terms of a simple definition, but by helping us understand the full nature of software architecture. First, we explore the concerns that are uniquely, or most appropriately, addressed by software architecture. Next, we consider the decisions and descriptions that characterize and formulate an architecture. We present our layered model of software architecture, which both organizes architectural decision making and architecture description. Finally, we consider how to communicate the architecture.

With this fundamental understanding of what software architecture is, architects can turn to the question of how to create architectures that are good, right and successful, and managers can consider how to hire and develop great architects.

by Ruth Malan and Dana Bredemeyer
Bredemeyer Consulting
ruth_malan@bredemeyer.com
dana@bredemeyer.com

From "Chapter 1. Software Architecture: Central Concerns, Key Decisions". The book is titled *Software Architecture Action Guide,* by Ruth Malan and Dana Bredemeyer.

Introduction

Software may not be the first thing your customers associate with your products or services, but it is, visibly or not, impacting your ability to impress and keep customers. Whether yours is a manufacturing company producing products with software content, or a services company using applications to support your service offerings, your reliance on software to create competitive differentiation has increased dramatically over the past few decades. While the signature competencies of your industry may be the obvious place to focus strategic attention, software architecture has emerged as a competency that a broad variety of businesses, including traditional software companies, have to develop, and do so quickly. Such is the pace of our times that while we are sorting out what software architecture is, we are trying to raise it to the level of business competency!

In this paper, we aim to help you understand what constitutes software architecture, and how best to express it. We consider what key concerns it addresses—what are the distinguishing concerns that, if not dealt with by software architecture, would be seriously debilitating for the project or system being built? We then consider the nature of decisions that characterize architecture. Next, we make the notion of software architecture actionable, by describing the different views that help architects address the key concerns of their architecture. Finally, we consider what it takes to communicate an architecture to different stakeholder groups, and deliver on the promises of architecture.

Central Concerns Addressed by Software Architecture

Over the past few decades, the complexity of software systems has increased substantially. If we consider only the complexity inherent in managing something that takes hundreds and even thousands of person-years to develop, then many of the software systems around today have complexity comparable to that of a skyscraper. As Kruchten (Bosch, 2000) and Booch et al (1999) observe, we cannot use the same ad hoc approach to build skyscrapers that we use to build dog-houses. A decade ago, Garlan noted that

> "as the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem... This is the software architecture level of design." (Garlan, 1992)

Clearly then, complexity is a key concern that we would like software architecture to address. This complexity presents itself in two primary guises:

- *intellectual intractability.* The complexity is inherent in the system being built, and may arise from broad scope or sheer size, novelty, dependencies, technologies employed, etc. Software architecture should make the system more understandable and intellectually manageable—by providing abstractions that hide unnecessary detail, providing unifying and simplifying concepts, decomposing the system, etc.
- *management intractability.* The complexity lies in the organization and processes employed in building the system, and may arise from the size of the project (number of people involved in all aspects of building the system), dependencies in the project, use of outsourcing, geographically distributed teams, etc. Software architecture should make the development of the system easier to manage—by enhancing communication, providing better work partitioning with decreased and/or more manageable dependencies, etc.

Given that we need to decompose the system to address complexity, what new problems emerge that have to be dealt with by the architecture?

- How do we break this down into pieces? A good decomposition satisfies the principle of loose coupling between components (or pieces), facilitated by clean interfaces, simplifying the problem by dividing it into reasonably independent pieces that can be tackled separately.
- Do we have all the necessary pieces? The structure must support the functionality or services required of the system. Thus, the dynamic behavior of the system must be taken into account when

designing the architecture. We must also have the necessary infrastructure to support these services.

- Do the pieces *fit* together? This is a matter of interface and relationships between the pieces. But good fit—that is fit that maintains system integrity—also has to do with whether the system, when composed of the pieces, has the right properties[1].

We refer to broad-scoped qualities or properties of the system as *cross-cutting concerns*, because their impact is diffuse or systemic. It may be a matter of preferring not to isolate these concerns because the decomposition is being driven by other concerns, or it may be that no matter how you might "slice-and-dice" the system, multiple parts are going to have to collaborate to address these cross-cutting concerns. At any rate, to effectively address cross-cutting concerns, they must be approached first at a more broad-scoped level. Many system qualities (also known as non-functional requirements or service-level agreements) are of this nature. They include performance, security and interoperability requirements. To make the picture more complicated, the system qualities may conflict, so that trade-offs have to be made among alternative solutions, taking into account the relative priorities of the system qualities.

Another concern that is key to architecture is whether the solution is congruent with the environment. This is not *just* a matter of interface and relationships with external systems, but of consistency and harmony with them. It is also a matter of being congruent with the strategy of the business and the purpose of users.

Not only should the architecture fit in the context of legacy systems, enhancing not destroying the value of past investments, but it should stylize what has been proven to work well and avoid repeating what does not. In addition to integrating lessons learned, it should identify and exploit opportunities for reuse within the system, and across systems. Further, it should anticipate the future, taking into account trends and likely (and even unlikely) future scenarios.

As software systems have been growing in complexity, the industry has been learning valuable lessons about building complex systems. Some individuals have acquired more proficiency, through experience and a fairly unique set of skills (Bredemeyer and Malan, 2002), at solving complex-system problems. Organizations rightly want to build competitive strength by magnifying the skills of the uniquely talented and experienced few at the apex of the organization's *system*-design prowess. These are becoming broadly known as architects, and their responsibility is to make architectural decisions that will manifest as the architecture of the software system.

Architectural Decisions

A distinctive characteristic of architectural decisions is that they need to be made from a broad-scoped or system perspective. Any decision that could be made from a more narrowly-scoped, local perspective, is not architectural (at the current level of system scope). This allows us to distinguish between detailed design and implementation decisions on the one hand, and architectural decisions on the other—the former have local impact, and the latter have systemic impact. That is, architectural decisions impact, if not all of the system, at least different parts of the system, and a broad-scoped perspective is required to take this impact into account, and to make the necessary trade-offs across the system.

---

1. In seminars, Russell Ackoff puts this challenge to his audience (we have paraphrased, based on memory): Collect together a team of the best automotive design engineers in the world. Assign them the task of selecting the best car component of each type. Will they be able to create the world's best car from these components? No, of course not!

   Even in a field like automobiles that is mature enough that there is a "dominant design" identifying the essential components of the system, these components are not simply interchangeable. Even if they could plug together, they are designed with different quality specifications (individually, they have different "externally visible properties" or guarantees). Typically, they are designed in conjunction with other associated components to deliver some more systemic property.

For example, if the system under consideration is an individual application, any decisions that could be made by component designers or implementers should be deferred to them and not appear as part of the architecture. If the scope of the architecture is a family of applications (or product line), then any decision that relates only to a single application (or product) should be deferred at least to the application architecture and not be part of the application family architecture.

However, a decision may have systemic impact but not be very important, in which case it is also not architectural. By nature, architectural decisions should focus on high impact, high priority areas that are in strong alignment with the business strategy, as shown in Figure 1.

	Low Impact	High Impact *(high priority, important to business)*
Systemic *(broad scope)*	not architectural (*this could be a trap)*	**focus of architectural decisions**
Local	not architectural	not generally architectural *(though might set architecture guidelines and policies as needed)*

Figure 1: Decision Scope and Impact

Based on our discussion of key concerns addressed by software architecture, we see that, at a minimum, architectural decisions have to do with

- system priority setting
- system decomposition and composition
- system properties, especially cross-cutting concerns
- system fit to context
- system integrity

Let us consider each of these in turn, though they are certainly *not* mutually exclusive sets of decisions!

System Priority Setting

In the design of any complex system, one has to pick where to excel, and where to make the myriad compromises necessary to get the system built. It is essential to make priorities explicit so that attention can be focused on high-priority areas, and so that trade-offs between conflicting concerns can be made rationally, and decisions can be justified in terms of agreed priorities. Architects need to lead the priority-setting process for technical aspects of the system. This is a highly strategic process, and has to be informed by:

- the business, including business strategy and direction, core competencies and resources, and politics
- the market including customers, competitors, suppliers and channel
- technology including trends and opportunities
- constraints including existing technology investments and legacy systems
- challenges and impediments to the success of the system, the development of the system, and the business.

Decomposition and Composition

Fundamental to software architecture is the structure of the system in terms of the primary structural elements or components of the system, and their interrelationships. Associated architectural decisions seek to address such concerns as complexity (applying the principles of “separation of concerns” and “divide and

© 2002 BREDEMEYER CONSULTING

conquer") and portability and flexibility (applying the principle of localizing areas that are likely to change together). While isolating particular concerns, the architect (or architecture team) has to ensure that the functionality or services of the system can be delivered by the components working in collaboration. That is, the system responsibilities (services and associated service level) have to be assigned to the components. This has to be done from a system perspective, so that consistent assumptions are made about each component's responsibilities. Further, the responsibilities and associated assumptions must be documented to provide the proper context for designing and developing each component relatively independently, on the one hand, and for using components without knowledge of the internals, on the other. These are the externally visible properties of the component (a la Bass et al., 1997; see definitions in the Appendix).

System Properties and Cross-Cutting Concerns

System decomposition isolates some particular concerns so that they can be addressed independently. Different partitioning choices tend to isolate different (sets of) concerns. The remaining concerns are, by nature, cross-cutting—they impact various parts of the system.

At a minimum, sets of collaborating components have to be considered together, to properly address each cross-cutting concern. For example, performance typically has to be considered in terms of the patterns of interaction that the architectural structure allows, and is not just a matter of optimizing the performance of the parts separately.

Sometimes a cross-cutting concern, like interoperability, warrants specific architectural attention, and a mechanism is designed to address it. The mechanism may be designed as a set of collaborating components, and in particular, specific roles of those components, focused on addressing the cross-cutting concern. The result may, for example, be to place an interface on a component that does not fit with its core cohesive set of responsibilities.[1]

System Fit to Context

Just as a building architect cannot overlook such contextual factors as the building site, hook-up to city services like sewer and utilities, and the competencies and resources in local supply, the software architect cannot overlook the context of the system. The key technical considerations alluded to by "system fit to context" have to do with interoperability, consistency and interface with external systems. However, fit within the development organization's culture and capabilities, are also considerations to be factored into architectural decisions and choices.

System Integrity

System integrity means having, or being conceived of as having, a unified overall design, form, or structure. It has to do with congruence of the pieces in the large, as well as in the small. We see this in building architecture, where architectural integrity has a fair amount to do with gross structural forms (levels, roof line, space layout), but also has to do with other details—even fine details like the size and style of windows and other architectural trim. You can see this in the extreme in the case of Frank Lloyd Wright, who even designed furniture for some of the homes he architected. From the building architecture analogy, we

1. One example is the IUnknown interface on every component in COM systems, that is part of the mechanism for allowing components to be dynamically interchanged. Quite a number of mechanisms of this nature have been formalized and commercialized as middleware, given the pervasive nature of the architectural problem they address. Another example is JetSend, which provides appliance interconnection mechanisms, and CCOW, which provides context consistency across medical applications (the Common Clinical Context (CCOW) Architecture Specification is available from HL7 at http://www.hl7.org). Many such mechanisms start out life as a differentiator, solving some pervasive problem or creating an innovative opportunity for a particular business. They may be applied within a product or across a product family or set of applications. In the latter case, they are often treated as part of the infrastructure (variously known as framework or platform) for the product family.

also see that architectural integrity has both to do with structural integrity and aesthetic notions of fit. Further, fit includes internal balance, compatibility and harmony among the parts, as well as fit to context and to purpose.

This has two immediate implications. Firstly, as architects are designing the high-level structures of the system, they need to create a sense of what system integrity is for that system, and they need to ensure that the architectural components, with their properties and relationships, accede to, and form a foundation for, this integrity. We see a key role to be played here by vision, architectural style, principles, and concepts chosen to provide a consistent approach from high-level structuring to detailed design.

Secondly, some decisions may have nothing to do with the high-level structures of the system, but if they have to do with the integrity of the architecture then they may be considered architectural. System integrity then, opens the door for architects to make decisions that might otherwise be considered the charter of component designers and implementers. However, we strongly encourage minimalist architecture, by which we mean keeping the architecture decision set as small as possible.

Minimalist Architecture

The *only* justifiable reason for restricting the intellectual freedom of designers and implementers is demonstrable contribution to strategic goals and systemic properties that otherwise could not be achieved (Malan and Bredemeyer, 2002 b). Architects are highly valuable, essential technical assets of any company, and their attention should not be squandered on decisions that are not, truly, architectural. Similarly, designers and implementers are also part of the critical capacity to produce innovation and value, and their ability to do this should not be unnecessarily restricted but rather channeled appropriately to fulfill the architectural vision and the business strategy it implements.

Three principles can be applied to achieve a minimalist architecture. The first has already been mentioned: if a decision could be made at a more narrow scope, defer it to the person or team who is responsible for that scope.

Second, only address architectural decisions at *high-priority* architecturally significant requirements. Architecturally significant requirements include strategic objectives, important distinct services the system must offer, and qualities or properties of the system that are systemic or have broad impact across (parts of) the system (Malan and Bredemeyer, 2002 a).

Third, as decisions are added to the architecture, they should be evaluated from the point of view of their impact on the overall ability of the organization to adopt the architecture. A decision may address a highly critical concern, but if it would cause the architecture effort to be derailed, it should be put aside, at least for now.

As part of the minimalist architecture discipline, each architectural decision should have a well-reasoned and documented rationale. Architectural decisions optimize at a broader scope, but may be suboptimal at a more local scope. The rationale should enable those with local visibility to understand the impact on the overall system of any locally optimized, but globally suboptimal, deviations from the architectural decision. On the other hand, providing rationale allows for “checks and balances” on the architecture, in that decisions that would substantially better achieve the architecturally significant requirements, without compromising higher-priority architectural requirements, can reasonably be brought up in contention with the architectural decisions.

Architecture Decision Framework

From the above discussion, we see that architectural decisions may be at different levels of abstraction. To be sure, the primary focus is on the *architecture*—the structural elements of the system together with their externally visible properties and relationships. However, there are higher-level decisions that guide and

constrain the system decomposition and structuring decisions (*meta-architecture*), and there may be lower-level decisions that guide and constrain the next level(s) of design and implementation (*architecture guidelines and policies*). This is captured in the layered model of software architecture shown in Figure 2.

We have found that this model provides a highly effective separation of concerns, helping architects to organize their decision-making process and providing focus for action. Indeed, our Visual Architecting Process (Malan and Bredemeyer, 2002) is organized around this model.

At the same time, the model organizes the architecture description, which consists of models, descriptions, explanations, etc., that capture the architectural decisions and help different stakeholders visualize the architecture and see how their concerns are addressed by it. The architectural description (ideally, at any rate) guides the creation of the system, and it is what we return to when we want to evolve the system.

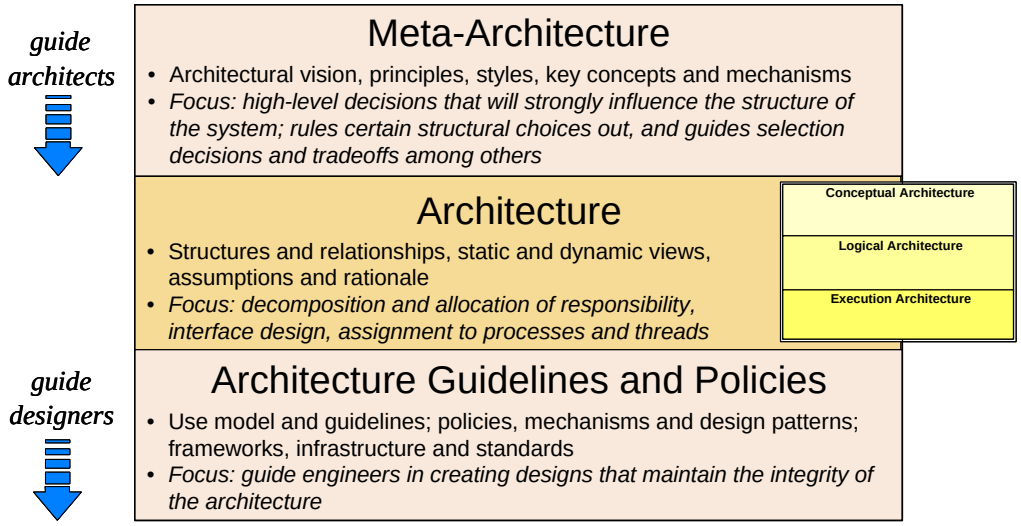

Figure 2: Software Architecture Decision Framework

Meta-Architecture

The meta-architecture is a set of high-level decisions that will strongly influence the integrity and structure of the system, but is not itself the structure of the system. The meta-architecture, through style, patterns of composition or interaction[1], principles, and philosophy, rules certain structural choices out, and guides selection decisions and trade-offs among others. By choosing communication or co-ordination mechanisms that are repeatedly applied across the architecture, a consistent approach is ensured and this simplifies the architecture. It is also very useful at this stage, to find a metaphor or organizing concept[2] that works for your system. It will help you think about the qualities that the system should have, it may even help you think about what components you need (in Conceptual Architecture), and it will certainly help you make the architecture more vivid and understandable.

1. For example, layered architecture or dataflow style. See Shaw and Garlan (1996), Buschmann et al. (1996) and Schmidt et al. (2000) for excellent work on architectural styles and architectural patterns.
2. In the highly recommended Harvard Business Review classic, "The Power of Product Integrity" by Clark and Fujimoto, the powerful system concept "Honda for the rugby player in a business suit" helped the Honda design team think about what kind of system to build, and influenced the team through all the myriad decisions and trade-offs that had to be made as the architecture and design progressed.

Architecture

Architecture is at the center of our layered decision model (Figure 2), and at the center of the architecting activity. It is where the system structures are created, taking into account system priorities and constraints, and ensuring that the system will achieve the system objectives and architectural requirements. This work is informed and constrained by the decisions made in the Meta-Architecture.

Within the architecture layer, we use different views to enhance the understandability of the architecture and to focus on particular concerns separately. We distinguish between *Conceptual, Logical* and *Execution* views, as shown in Figure 3 and described below.

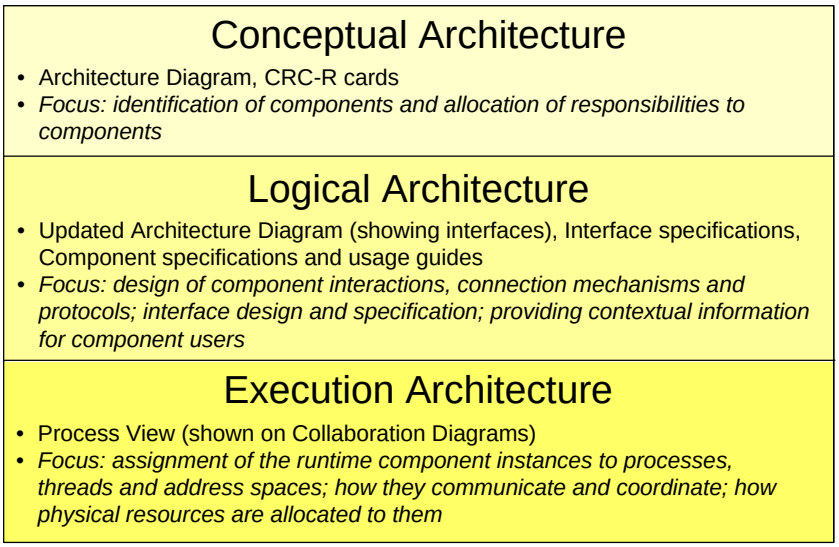

Figure 3: Architecture Views

Conceptual Architecture

The Conceptual Architecture identifies the high-level components of the system, and the relationships among them. Its purpose is to direct attention at an appropriate decomposition of the system without delving into details. Moreover, it provides a useful vehicle for communicating the architecture to non-technical audiences, such as management, marketing, and users. It consists of the Architecture Diagram (without interface detail) and an informal component specification for each component.

Logical Architecture

In Logical Architecture, the externally visible properties of the components are made precise and unambiguous through well-defined interfaces and component specifications, and key architectural mechanisms are detailed. The Logical Architecture provides a detailed “blueprint” from which component developers and component users can work in relative independence. It incorporates the detailed Architecture Diagram (with interfaces), Component and Interface Specifications, and Component Collaboration Diagrams, along with discussion and explanations of mechanisms, rationale, etc.

Execution Architecture

An Execution Architecture is created for distributed or concurrent systems. The process view shows the mapping of components onto the processes of the physical system, with attention being focused on such concerns as throughput and scalability. The deployment view shows the mapping of (physical) components in the executing system onto the nodes of the physical system.

© 2002 BREDEMEYER CONSULTING

Architectural Views and Structure and Behavior

Both structural and behavioral views are important to thinking through and representing architecture:

- *Structural View.* If we accept that "architecture is the high-level structure of the system comprised of components, their interrelationships, and externally visible properties" (adaptation of the Bass, Clements, Kazman definition), the structural view is central. It consists of the Architecture Diagram (stereotyped UML Class Diagram), and Component and Interface Specifications.
- *Behavioral View.* In decomposing the system into components and designing their interfaces, and in designing mechanisms to address key cross-cutting concerns, we have to answer the question "How does this work?" Likewise, in understanding and using the architecture, we have to be able to answer the same question. This is the role of the behavioral view, with its Component Collaboration or Sequence Diagrams (stereotyped UML Sequence and Collaboration Diagrams).

Structural and behavioral views are applicable for each of the Conceptual, Logical and Execution Architecture views (or layers), as shown in Figure 4.

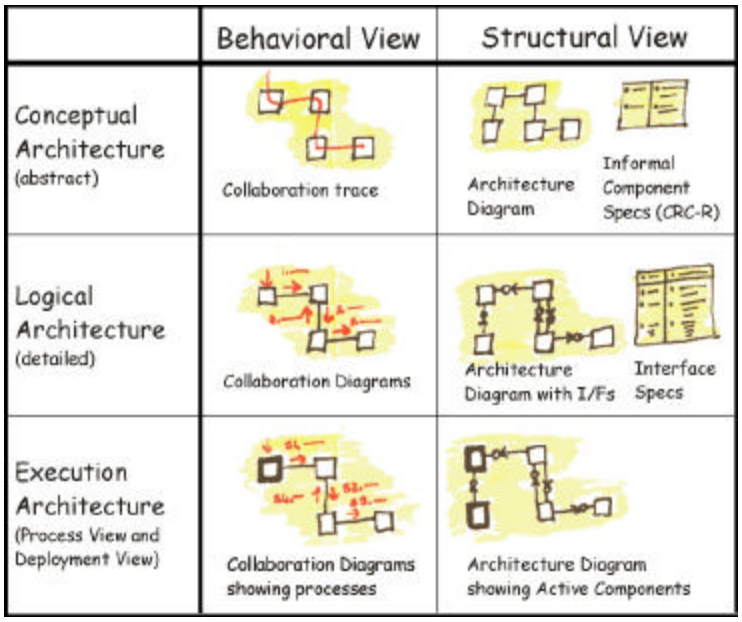

Figure 4: Architecture Views with Structure and Behavior

In general, however, you would want to at least have:

- an Architecture Diagram and informal component descriptions for Conceptual Architecture,
- Interface Specifications, Component Specifications and an updated (Logical) Architecture Diagram showing interfaces and relationships for Logical Architecture. You should also include Collaborations Diagrams for key Use Case steps. Note that these may be used to *describe* system behavior (they are simply illustrative), or to *prescribe* how system behavior is to be accomplished, and you should be clear about which of these you intend. In general, you should avoid being prescriptive unless you have a strong architectural reason for curtailing the creative options open to designers and developers.

Architectural Guidelines and Policies

To help maintain system integrity or to address cross-cutting concerns, architects may include decisions focused at guiding or constraining lower-level design or even implementation in the architecture decision set. Recall our caution: the *only* justifiable reason for restricting the intellectual freedom of designers and implementers is demonstrable contribution to strategic and systemic properties that otherwise could not be achieved. That said, there is a fair amount that architects can valuably do to help designers and implement-

ers in applying the architecture and in paying attention to the right characteristics of the problem so that their decisions, in turn, are in alignment with all that is explicit in the architecture and all that is implicit in the architecture concept.

Communicating Architectural Decisions

The architectural decisions are impotent unless remembered, bought-into and understood. The primary goals of architecture documentation are to:

- *Record the architects' decisions*. To meet this goal, the documentation must be complete and unambiguous.
- *Communicate the architecture*. To meet this goal, you must consider what each of your stakeholders needs to know, and how best to convey what they need to know. The comprehensive architecture specification document (set) that addresses the first goal is, realistically, probably going to become shelfware and will not serve the goal of communication!

Let us consider what needs to be recorded and communicated.

Architecture Drivers

Though not part of the architecture as such, the drivers that shape the architecture are important to make explicit and sharable. They include:

- Architecture Vision, expressing the desired state that the architecture will bring about.
- Architectural Requirements, capturing stakeholder goals and architecturally significant behavioral (functional) requirements as well as system qualities (non-functional requirements) and constraints.
- Assumptions, Forces and Trends, documenting assertions about the current business, market and technical environment now and over the architecture planning horizon.

Architecture Views

In our Software Architecture Decision Framework (Figures 2 and 3), we presented a set of standardized views. These are what we have found to be useful in guiding architects as they make architectural decisions—that is, they provide useful thinking tools for considering decisions and choosing among alternatives. They also become the foundation for the architecture specification, by which we mean the complete set of architecture decisions at the chosen level(s) of abstraction, specificity and precision.

Architecture Documentation

With the architecture drivers and various architecture views, you have the raw material from which to compose documents and presentations targeted at different audiences (“viewpoints,” in the parlance of IEEE1471). At a minimum, your document set should include:

- *Reference Specification*. The full set of architecture drivers, views, and supplements such as the architecture decision matrix and issues list, provides your reference specification.
- *Management Overview.* For management, you would want to create a high-level overview, including vision, business drivers, Architecture Diagram (Conceptual) and rationale linking business strategy to technical strategy.
- *Component Documents*. For each component owner, you would ideally want to provide a system-level view (Logical Architecture Diagram), the Component Specification for the component and Interface Specifications for all of its provided interfaces, as well as the Collaboration Diagrams that feature the component in question.

Other Communication Vehicles

Communication cannot be restricted to documents, no matter how carefully targeted. To be applied, the architecture must be understood. To be supported, it must be bought-into. Architects and their sponsors need to do presentations of various sorts to various audiences, from strategic to project management, to designers, implementers and testers, and to marketing and customers. It also helps to create tutorials and workshops on the architecture, showing how to apply it in developing applications or products. Further, architects need to generally rove about, addressing questions and concerns as they arise. We strongly encourage architecture teams to create a communication plan early, and identify what information is needed and what needs to be provided to whom, and in what format. This builds the necessary time for creating communication vehicles to produce alignment and understanding into the architecture project plan.

Conclusion

We have considered what software architecture addresses, and how you express software architecture. But all the documentation and presentations in the world will not suffice, unless the software architecture is:

- *good*—it is technically sound and clearly represented
- *right*—it meets the needs and objectives of key stakeholders, and
- *successful*—it is actually used in developing systems that deliver strategic advantage.

Of course, how one creates an architecture that is all of these things is beyond the scope of this conclusion! Our Visual Architecting Process (VAP) is, however, the focus of a companion paper (Malan and Bredemeyer, 2002 c). This process covers the techniques, including architectural modeling and architecture validation, used in creating a technically sound architecture. It covers architectural requirements and driving trade-offs to create the right architecture, and it covers the organizational process steps that help ensure that the architecture is embraced and used informedly so that, ultimately, the architecture is successful.

Much of what it takes to ensure that an architecture is successful, relies on the non-technical skills of the architect(s). These include organizational politics and leadership. Our paper on the Role of the Architect (Bredemeyer and Malan, 2002) will help you to identify the characteristics and competencies of architects capable of translating from business strategy to an effective technical implementation of that strategy through an architecture that is good, right and successful.

Acknowledgments

We would like to thank our colleague, David Redmond-Pyle, for motivating us to greatly improve this paper, and for his very helpful suggestions for specific improvements to it. We cannot overlook the stimulating conversations we have had on architecture with our other associates at Bredemeyer Consulting, in particular Aaron Lafrenze and Raj Krishnan.

Also, we thank the community of architects who continue to be our source of insight and inspiration. We are very privileged to have worked with high calibre architects at companies around the world, and learned much about the nature of good, right and successful architecture from them. We would especially like to thank Joe Sventek at Agilent Technologies, Bill Baddely, Bill Crandall, Derek Coleman, Martin Griss, Reed Letsinger, Holt Mebane, and Keith Moore at Hewlett-Packard, and Rob Seliger at Sentillion, who all made an early and powerful impact on our views of architecture. More recently, Ilia Fortunov and Lars Lindstedt at Microsoft, Bill Branson at Frank Russell, not to mention the literally hundreds of architects who have taken our workshops or interacted with us on consulting projects, have made substantive contributions to our views. We thank you all!

References

Bachmann, Felix, L. Bass, J. Carriere, P. Clements, D. Garlan, J. Ivers, R. Nord, and R. Little, *Software Architecture Documentation in Practice: Documenting Architectural Layers*, draft available at http://www.sei.cmu.edu/publications/documents/00.reports/00sr004.html

Bass, Clements, and Kazman. *Software Architecture in Practice*, Addison-Wesley, 1997.

Bredemeyer, Dana and Ruth Malan, "The Role of the Architect", white paper published on the *Resources for Software Architects* web site, http://www.bredemeyer.com/papers.htm, 2002.

Booch, G., J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

Bosch, Jan, *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*, Addison-Wesley, 2000

Buschman, Frank, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *A System of Patterns: Pattern-Oriented Software Architecture*, Wiley, 1996.

Hofmeister, Nord and Soni, Applied Software Architecture, Addison-Wesley, 2000. (especially Ch. 6 pp. 125-157)

Malan, Ruth and Dana Bredemeyer, "Architectural Requirements", column published on the *Resources for Software Architects* web site, http://www.bredemeyer.com/ArchitectingProcess/ArchitecturalRequirements.htm, 2002 a.

Malan, Ruth and Dana Bredemeyer, "Minimalist Architecture", column published on the *Resources for Software Architects* web site, http://www.bredemeyer.com/Architecture/MinimalistArchitecture.htm, 2002 b.

Malan, Ruth and Dana Bredemeyer, "The Visual Architecting Process", white paper published on the *Resources for Software Architects* web site, http://www.bredemeyer.com/papers.htm, 2002 c.

Ogush, M., D. Coleman, and D. Beringer, "A Template for Documenting Software Architectures", March 2000, used to be available on http://www.architecture.external.hp.com/Download/download.htm

Rechtin, E. *Systems Architecting: Creating and Building Complex Systems*. Prentice-Hall, 1991.

Schmidt, Douglas, M. Stal, H. Rohnert, F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000

Seliger, R. "An Approach to Architecting Enterprise Solutions". *HP Journal*, Feb 1997

Shaw, Mary and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996

Youngs, R., D. Redmond-Pyle, P. Spaas, and E. Kahan, "A Standard for Architecture Description", *IBM Systems Journal*, Vol 38 No 1. http://www.research.ibm.com/journal/sj/381/youngs.html

Key UML References for Software Architects

Arlow and Neustadt, *UML and the Unified Process: Practical Object-Oriented Analysis and Design*, Addison-Wesley, 2002

Booch, G., J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

Cheesman, Jon and John Daniels, *UML Components*, Addison-Wesley 2001.

Rumbaugh, J., I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

The official OMG Unified Modeling Language (UML) Documentation is available at http://www.omg.org.

Appendix A: System Definitions

UML 1.3: A system is a collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints.

IEEE Std. 610.12-1990: A system is a collection of components organized to accomplish a specific function or set of functions.

Appendix B: Software Architecture Definitions

The following definitions are by influential writers in this field. They are organized chronologically, with the most recent first. (You can also check out the SEI's great collection of Software Architecture Definitions.)

UML 1.3: Architecture is the organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.

Jazayeri, Ran, and van der Linden. *Software Architecture for Product Families: Principles and Practice,* Addison Wesley Longman, 2000. Software architecture is a set of concepts and design decisions about the structure and texture of software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domains.

Booch, Rumbaugh, and Jacobson, *The UML Modeling Language User Guide,* Addison-Wesley, 1999: An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition.

Bass, Clements, and Kazman. *Software Architecture in Practice,* Addison-Wesley 1997: 'The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

By “externally visible” properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction.”

Garlan and Perry, guest editorial to the *IEEE Transactions on Software Engineering,* April 1995: Software architecture is “the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.”

IEEE Std. 610.12-1990: Architecture is the organizational structure of a system.

Mirriam-Webster’s Collegiate Dictionary. Architectural: having, or conceived of as having, a single unified overall design, form, or structure.

About Bredemeyer Consulting

Bredemeyer Consulting provides a range of consulting and training services focused on Enterprise, System and Software Architecture. We provide training and mentoring for architects, and typically work with architecture teams, helping to accelerate their creation or renovation of an architecture. We also work with strategic management, providing consulting focused on developing architectural strategy and organizational competency in architecture.

We manage the ***Resources for Software Architects*** web site (see http://www.bredemeyer.com). This highly acclaimed site organizes a variety of resources that will help you in your role as architect or architecture program manager. A number of Bredemeyer Consulting's *Action Guides*, presentations and white papers are on the Papers and Downloads page (http://www.bredemeyer.com/papers.htm). You may also be interested in our *Software Architecture* and *Enterprise Architecture* Workshops, as well as our *Architectural Leadership* class. For more information, please see http://www.bredemeyer.com/training.htm.

BREDEMEYER CONSULTING
Bloomington, IN 47401
Tel: 1-812-335-1653
http://www.bredemeyer.com