Architecture Decisions: Demystifying Architecture

Jeff Tyree and Art Akerman, *Capital One Financial*

You can make your architecture more transparent and clarify its rationale for all stakeholders by explicitly documenting major architecture decisions.

It's hard to count the "mystical" software architectures we've reviewed over the years. Their creators seem to want us to take many things on faith. They require us to assume that the solution is somehow tied to business drivers, that their architectural choices have rationales and are implementable, and that they adequately considered competing alternatives. Yet at times, the architecture seems to have no grounding in business needs, and it's difficult to see if the architects understand their decisions' implications for the environment in which the architecture is to be deployed. We believe that a key to demystifying architecture products—to dispel or reduce the "magic"—lies in the architecture decisions concept.

Why architecture decisions?

Decisions. We make them every day. Some are big, some small. So what's the big deal? In most architecture development processes, decisions aren't documented explicitly but are implicit in the models the architect builds. Architects make structuring decisions, such as choosing patterns, in the component (logical) model, and deployment decisions, such as choosing runtime patterns, in the physical model. However, stakeholders such as developers, customers, and even other architects don't have the energy to pore through architectural views to understand the architecture. Developers want clear, decisive guidance on how to proceed with a design. Customers want a clear understanding of the environmental changes that must occur and assurance that the architecture meets their business needs. Other architects want a clear, salient understanding of the architecture's key aspects, including the rationale and options the original architect considered.

Traditional architectural approaches such as RM-ODP (Reference Model for Open Distributed Processing), 4+1, or RUP (Rational Unified Process) don't satisfy these wants in a clear and simple manner.[1–3] These approaches break down in several areas, such as

- *Conveying change*. In an evolutionary environment, it is challenging to document architecture changes through conventional views in a way developers or designers can understand. Developers don't want to wade

0740-7459/05/$20.00 © 2005 IEEE

A simple document describing key architecture decisions can go a long way in demystifying past and future system architectures.

through a lengthy component model just to find a few key items that have changed. In addition, in an environment where object orientation isn't the norm, many developers aren't familiar with component models and interaction diagrams. They're just looking for changes to the architecture's key aspects that will influence their design.

- *Conveying implications*. Traditional approaches don't clearly state the architecture's implications. What are the organizational impacts? What are the training needs?
- *Conveying rationale and options*. Traditional approaches tend to rely on models to convey the majority of the architecture's information. They don't focus on the rationale and options the architect considered. Without these two key elements, stakeholders begin to ask the same questions again and again—questions that have long been answered.
- *Ease of traceability*. Traceability is challenging in many respects, regardless of the development lifecycle phase. The challenge of mapping the objectives (which we define as business needs, risks, system issues, change cases, and nonfunctional requirements) to specific architectural elements is unmanageable. We need a simpler way to ensure that the architecture meets its objectives.
- *Providing agile documentation*. Traditional approaches tend to rely on architecture expression through a set of views. We've found it necessary to provide alternative documentation forms that balance architectural and agile objectives.

To address these breakdowns, we believe that architecture decisions are the missing link. If we elevate architecture decisions to first-class status and explicitly document and socialize them, they become an effective tool to help stakeholders understand the architecture. If an architect doesn't have time for anything else, these decisions can provide a concrete direction for implementation and serve as an effective tool for communication to customers and management.

What exactly are architecture decisions?

A complex architecture probably reflects thousands of decisions, big and small. Is it necessary for an architect to make and explicitly document all of them? We agree with Ruth Malan and Dana Bredemeyer,[4] who argue that an architect should make as few decisions as possible, deferring the rest until later in the lifecycle. This lets the architect maintain a balance between guiding and constraining the technical organization. An architect absolutely should make the decisions that identify the system's key structural elements, their externally visible properties, and their relationships.[5] To test a decision's architectural significance, an architect should ask the following question: does this decision affect one or more system qualities (performance, availability, modifiability, security, and so on)? If so, an architect should make this decision and document it completely.

Properly documenting these architecture decisions is critical because architects make them in complex environments and they involve trade-offs. How many times have we looked at an architecture and been surprised (or even terrified) by the decisions it was based upon? Our first reaction is to ask several rhetorical questions: "What were these people thinking? Had they never heard of sound principles of good design? Did they think that the system wouldn't live longer than a month?" Well, okay, maybe a few were inexperienced, short-term thinkers. But most had good intentions and did what seemed right in the moment. The decisions made sense under the circumstances, which cost and schedule constrained. However, looking back, after the dust has settled and the original system designers are long gone, we have no context around these decisions; we have no history. All we can do is shake our heads in disbelief. In the end, as Gustave Flaubert reportedly wrote in 1871, "Our ignorance of history causes us to slander our own times."

A simple document describing key architecture decisions can go a long way in demystifying past and future system architectures. IBM's e-Business Reference Architecture Framework, where architecture decisions are a key deliverable, has helped us learn how to document these decisions.[6] Table 1, derived from the REMAP (Representation and Maintenance of Process Knowledge) and DRL (Decision Representation Language) metamodels, lists the essential information for each decision.[7] We've added two additional fields, related principles and notes.

Table 1

Architecture decision description template

Issue	Describe the architectural design issue you're addressing, leaving no questions about why you're addressing this issue now. Following a minimalist approach, address and document only the issues that need addressing at various points in the life cycle.
Decision	Clearly state the architecture's direction—that is, the position you've selected.
Status	The decision's status, such as pending, decided, or approved.
Group	You can use a simple grouping—such as integration, presentation, data, and so on—to help organize the set of decisions. You could also use a more sophisticated architecture ontology, such as John Kyaruzi and Jan van Katwijk's, which includes more abstract categories such as event, calendar, and location.[8] For example, using this ontology, you'd group decisions that deal with occurrences where the system requires information under event.
Assumptions	Clearly describe the underlying assumptions in the environment in which you're making the decision—cost, schedule, technology, and so on. Note that environmental constraints (such as accepted technology standards, enterprise architecture, commonly employed patterns, and so on) might limit the alternatives you consider.
Constraints	Capture any additional constraints to the environment that the chosen alternative (the decision) might pose.
Positions	List the positions (viable options or alternatives) you considered. These often require long explanations, sometimes even models and diagrams. This isn't an exhaustive list. However, you don't want to hear the question "Did you think about … ?" during a final review; this leads to loss of credibility and questioning of other architectural decisions. This section also helps ensure that you heard others' opinions; explicitly stating other opinions helps enroll their advocates in your decision.
Argument	Outline why you selected a position, including items such as implementation cost, total ownership cost, time to market, and required development resources' availability. This is probably as important as the decision itself.
Implications	A decision comes with many implications, as the REMAP metamodel denotes. For example, a decision might introduce a need to make other decisions, create new requirements, or modify existing requirements; pose additional constraints to the environment; require renegotiating scope or schedule with customers; or require additional staff training. Clearly understanding and stating your decision's implications can be very effective in gaining buy-in and creating a roadmap for architecture execution.
Related decisions	It's obvious that many decisions are related; you can list them here. However, we've found that in practice, a traceability matrix, decision trees, or metamodels are more useful. Metamodels are useful for showing complex relationships diagrammatically (such as Rose models).
Related requirements	Decisions should be business driven. To show accountability, explicitly map your decisions to the objectives or requirements. You can enumerate these related requirements here, but we've found it more convenient to reference a traceability matrix. You can assess each architecture decision's contribution to meeting each requirement, and then assess how well the requirement is met across all decisions. If a decision doesn't contribute to meeting a requirement, don't make that decision.
Related artifacts	List the related architecture, design, or scope documents that this decision impacts.
Related principles	If the enterprise has an agreed-upon set of principles, make sure the decision is consistent with one or more of them. This helps ensure alignment along domains or systems.
Notes	Because the decision-making process can take weeks, we've found it useful to capture notes and issues that the team discusses during the socialization process.

Generally, only two views of the decisions exist. The architect provides the first, which the template above describes, to the technical stakeholders. The grouping (categorization) of decisions allows for filtering based on the technical stakeholders' interests. For example, data architects reviewing the decisions can focus only on the decisions grouped as *data*. We also find it helpful to use a simple coloring scheme (red, blue) to point out controversial or incomplete decisions. The architect provides the second view to management or business stakeholders, generally as a Microsoft PowerPoint presentation. This view summarizes key decisions and their implications.

Once the team reaches a final architectural decision, they'll need to "socialize" the result—that is, convince the rest of the organization that they've chosen appropriately. The architecture decision template is useful because it provides a common language for discussing decisions. Reviewers can easily see the decision's status, rationale, and impacts. In practice, this has proven to be much more powerful than reviewing, for example, component models. The team should socialize controversial decisions early and often and other decisions during the normal review process.

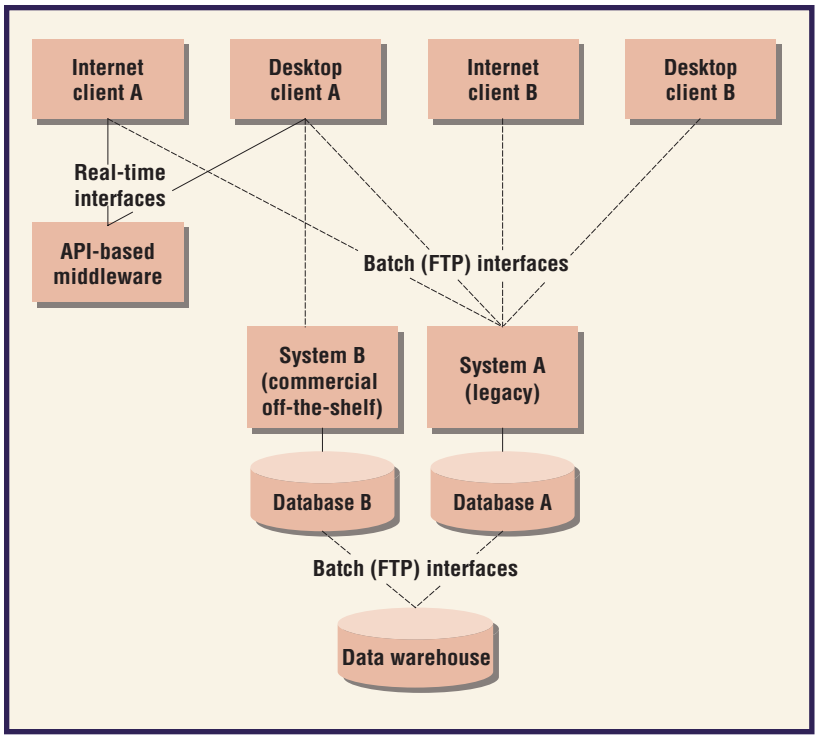

Figure 1. Current architecture.

An illustrative example

For this example, we consider a large financial organization's IT systems. These systems suffer from shortcomings typical of most enterprises: business functionality is duplicated across multiple systems, interfaces tightly couple various parts of the infrastructure, and system maintenance costs are higher than desired.

To start addressing these challenges and meet emerging business needs, the organization developed an architectural vision to guide its systems' transformation through numerous infrastructure improvement projects. Discussing the process for constructing these types of architectural visions is beyond this article's scope; "An Architectural Process for System Evolution" covers it in detail.[9]

Current state

The company's most urgent business need involves letting the customers receive an immediate response when they apply for a new financial product (credit card, loan, savings account, and so on). The organization performs the complex approval process in batches, with notification by letter. An in-house-developed system, which is showing its age, performs the existing batch process (System A in Figure 1). Its processing flow and business rules are hard-coded with few configuration options.

The company recently deployed a commercial off-the-shelf solution (System B) to handle the approval process for one type of financial product. The COTS package is based on a flexible workflow engine, in which non-IT personnel can create rules. This solution works in batch and interactive mode but must be heavily customized to support other product types.

Both systems interact with numerous client applications. Since none of these interactions happen in real time, they're done by FTP-based file exchange. The company has an internally developed API-based middleware platform, which many customer-facing applications use. This middleware platform has become a bottleneck for developing new functionality because the company must coordinate any change with all client systems using it. Neither System A nor System B exposes its functionality through this middleware.

Architecture decisions

The objectives define a problem that a technical solution must address. We start by identifying the architecture decisions we need to make:

- How can we transform the current batch business process into an interactive one?
- How can we connect a decisioning back-end system to the client applications so it can receive requests and respond with decisions in real time?
- How can we make the same flexibility with respect to changing business rules that exists in System B available to all financial products?

The first key decision must address implementation of the interactive approval process. We consider three alternatives: rearchitecting existing batch logic in System A, extending System B to handle a new product type, or developing a replacement for System A.

Selecting an approach

We first identify the criteria we'll use to select the best approach. Obviously, the ability to satisfy business needs is essential. Providing instantaneous decisions would differentiate the company's offerings from its competitors. This leads to the following objectives or goals:

Table 2

Alternative approaches for implementing interactive approval processing

	Selection criteria: will the solution …	Rearchitect System A	Extend System B	Replace System A
N1	Enable interactive approval?	Yes	Yes	Yes
N2	Be ready for delivery in six months?	Yes	Yes	No
N3	Reduce time to market for future enhancements?	No	Yes	Yes
N4	Reduce costs?	No	Yes	No
N5	Reduce risks?	No	Yes	No
N6	Disrupt business operations as little as possible?	Unknown	Unknown	No
N7	Meet desired system characteristics?	No	Yes	Unknown
N8	Support enterprise principles (reuse existing infrastructure, buy before build)?	Yes	Yes	No
N9	Use proven technologies?	Yes	Yes	No

- N1: Provide instant approval for customers online and over the phone.
- N2: Deploy the capability within six months because of competitive pressures.
- N3: Reduce time to market for future enhancements by giving business customers greater control over changing business rules.
- N4: Reduce the infrastructure's operational costs.
- N5: Reduce risks associated with tight coupling and redundant business logic.

Our solution should also minimize potential disruptions to normal business operations (N6)

Table 3

Decision DOI: Extend System B to implement interactive approval processing

Issue	Current IT infrastructure doesn't support interactive approval functionality for most financial products.
Decision	Extend System B beyond its original functional boundaries to implement interactive approval processing for the financial products it handles.
Status	Approved
Grouping	System structuring
Assumptions	We must deliver new capabilities in six months. We can't increase the project budget by more than 10 percent. We'll use existing client applications.
Constraints	None
Positions	Rearchitect existing batch logic in System A. Extend System B to handle a new product type. Develop a replacement for System A.
Argument	Extending System B to handle approval processing for all financial products will reduce duplicate business logic, let all lines of business use flexible workflow and rules engines to improve time to market for new products, and reduce maintenance costs and operational risks. This solution also has a solid chance of meeting project timelines because the IT organization is already familiar with the proposed technology.
Implications	The team will need to develop a real-time interface between online and phone client applications and System B. System B will become a mission-critical platform because multiple lines of business depend on it. The team needs to develop and deploy adequate disaster-recovery procedures for this system. The rollout strategy should focus on minimizing the risk of negatively affecting System B's other financial products.
Related decisions	See Figure 2.
Related requirements	See Table 2.
Related artifacts	None
Related principles	Reuse existing infrastructure, buy before build. Use proven technologies.
Notes	None

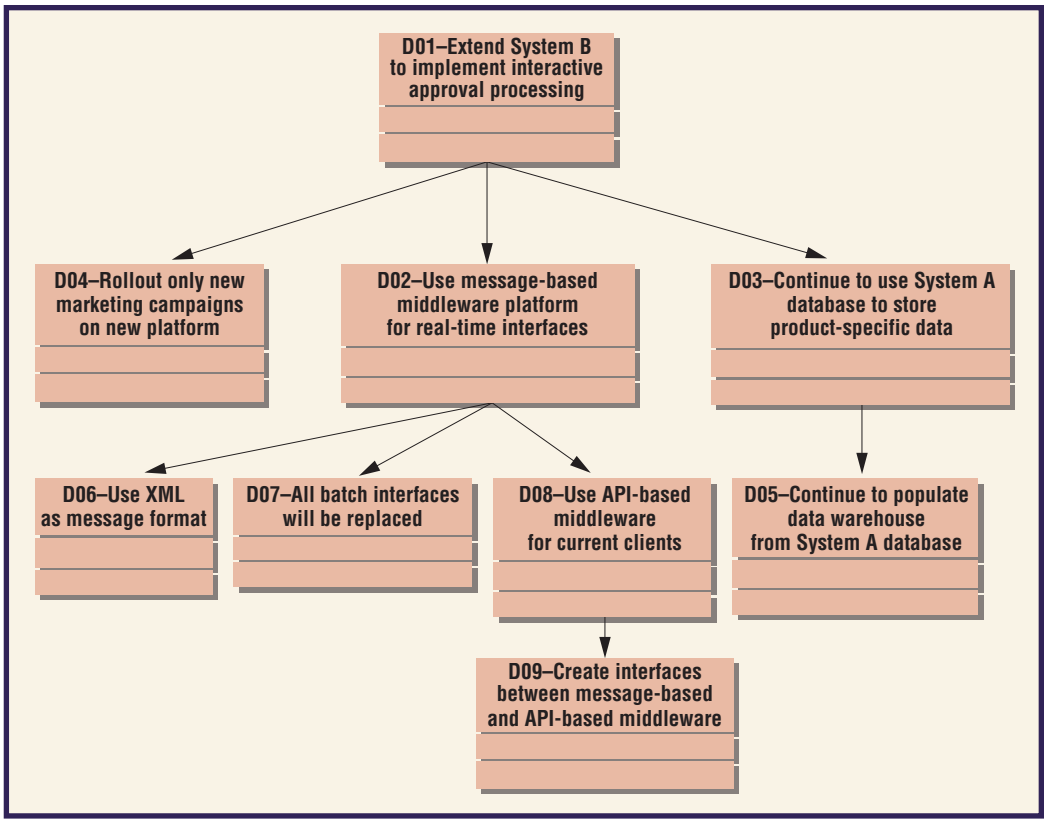

Figure 2. Architecture decisions model for the financial-institution example. Nine decisions all connect to D01.

and meet desired system characteristics, such as performance, capacity, reliability, and so on (N7). Finally, it should be consistent with existing enterprise architecture principles. The following principles apply to our situation:

- Reuse existing infrastructure, *buy before build*. To meet time commitments to the business, we need to reuse as much of the current infrastructure as possible with the aim of gradually replacing custom-developed components with COTS components in the application's future releases (N8).
- Use proven technologies. Although many promising technologies seem to be on the horizon, it's unwise to put the business at risk by relying on unproven technologies (N9).

We could analyze the alternatives in a number of ways, but we prefer a simple comparison of each option's pros and cons (see Table 2).

Clearly, the second alternative has more benefits. The only drawback is the risk of a negative impact on other lines of business that use System B. However, we can mitigate this risk through a well-planned rollout strategy.

Documenting, analyzing, and socializing the decision

Now we have our first decision: *D01—Extend System B to implement interactive approval processing*, which we document using our standard format (see Table 3).

As a result of this decision, we must analyze numerous implications and possibly turn them into separate decisions. Some of these decisions will determine specific tools we'll use to implement different functions; others will address design strategies (see Figure 2). We might even make decisions about the testing, deployment, and migration processes. All these new decisions will be connected to D01—that is, changes in D01 will likely ripple across the whole decision hierarchy (see Table 3's metamodel). Alternatively, we might find that a downstream decision creates a suboptimum solution. For example, it might be too complex to introduce interfaces between the old and new middleware platforms (D09). We'd then have to alter some or all dependent decisions (D08, D02, and even D01). This shows that the decision-developing process is iterative.

The architect makes each decision using the same process—identifying a problem, devel-

oping a set of alternatives, and assessing their viability. Together, these decisions paint a clear picture of the final solution (see Figure 3):

- The team will consolidate approval processing onto a new COTS platform (System B). They'll use the flexible rules-based workflow engine to configure all the marketing campaigns' aspects for all financial products.
- To minimize the amount of new work, System B will use the data structures from a legacy database (Database A).
- The team will use the new message-oriented middleware platform to facilitate real-time interaction between client applications and back-end systems.
- To ensure the projects' timely delivery, the team has decided to use API-based middleware for these clients, which already have live connections with it. The message-oriented middleware will still provide a gateway to the back-end systems, which would require interfaces between two middleware platforms.

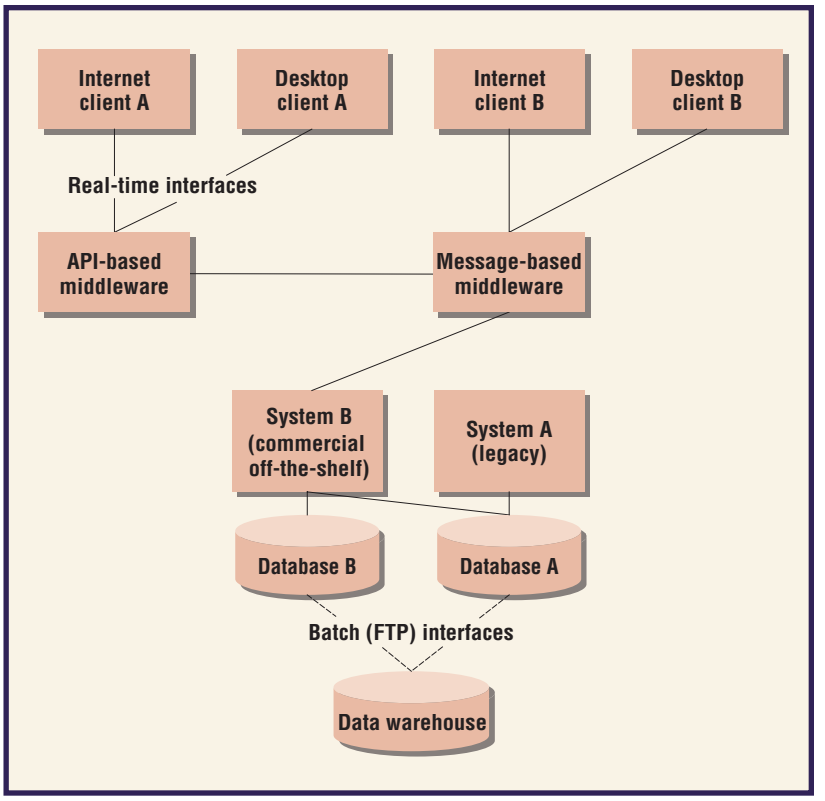

Figure 3. Future architecture.

The architect should now review the decisions with the rest of the project team and the project stakeholders. Once the architect obtains buy-in on the solution, he or she can further define the architecture. The next step would be to elaborate the decisions through a series of architectural views (component models, deployment models, and so on).

Did it work?

In the beginning, we identified numerous breakdowns in conventional architecture-development approaches. By explicitly focusing on architecture decisions, we were able to address these issues:

- *Conveying change*. The decisions represent key changes in a way the developers and designers can understand easily. They clearly identify the affected systems and interfaces.
- *Conveying implications*. The decisions describe more than just a solution—they also communicate the essential risks and issues. The team is informed about where it should focus its attention. For example, the rollout strategy is critical.
- *Conveying rationale and options*. The decision-making process is transparent. Anybody can read a decision description in Table 3 and understand how the team developed it. People might still challenge some subjective assessments—for example, that this solution had a solid chance to meet delivery timeframes. If the assumptions change, a decision would have to change as well.
- *Ease of traceability*. Table 2 lets you trace a decision back to requirements.
- *Providing agile documentation*. In an agile process, the team doesn't have time to wait for the architect to completely develop and document the architecture. The architect communicates each decision separately, with the caveat that it's subject to change due the effects of downstream work (see Figure 2). As long as they understand these relationships and risks, a team can start using the decisions.

We'd like architecture decisions to have a permanent place in the software architecture development process. A good place to start would be to add them to a standard conceptual model for an ar-

Other Approaches: Design Rationale and Views

In this article, we describe architecture decisions simply as an instance of applying design rationale within the context of software architecture. Design rationale research has been active for nearly two decades. For an overview of other approaches, see *Design Rationale: Concepts, Techniques and Use*.[1] When that work was done, a key issue design rationale researchers faced was how much to modify existing design practice to incorporate design rationale. We attempt to address this still-relevant question in the context of software architecture.

In *Documenting Software Architectures: Views and Beyond*,[2] Paul Clements and his colleagues emphasize design rationale's importance. They provide an outline for decision description as well as guidelines on which decisions to justify. Clements states that "perhaps the most important concept associated with software architecture documentation is that of the view." We would argue that architecture decisions are the most important concept.

In *Documenting Software Architectures in an Agile World*,[3] Clements and his colleagues make a proposal for reconciling the Views and Beyond and agile approaches. They propose that the architect "using the view selection scheme of the V&B approach, decide which architecture views [he] would want to produce, given enough resources. … Choosing a view identifies a family of design decisions that the architect needs to resolve and be able to express." Our approach is different. We first determine what decisions are important. These decisions then drive architecture, and hence the views.

References

1. T.P. Moran and J.M. Carroll, eds., *Design Rationale: Concepts, Techniques, and Use*, Lawrence Erlbaum Associates, 1996.
2. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Pearson Education, 2003.
3. P. Clements et al., *Documenting Software Architectures in an Agile World*, tech. report CMU/SEI-2003-TN-023, Software Eng. Inst., 2003.

chitecture description. The current IEEE model includes rationale but not its implications or a clean mapping between it and other architectural elements.[10] This would involve standardizing on a metamodel (notation) for representing the rationale, combining aspects of such notations as IBIS (Issue-Based Information System), QOC (Questions, Options, and Criteria), DRL (Decision Representation Language), Seiichi Komiya's proposed extensions,[11] and the elements we've proposed in this article. Laurent Karsenty's study provides insight into what additional information the architects should capture as part of the metamodel.[12]

Paul Clements and his colleagues provide a potential standard outline for documenting architectures.[13] You could use our article to define a framework for how the Views and Beyond approach represents design rationale. However, decisions drive views, rather than views driving decisions. From this perspective, decisions are the architecture's primary representation.

Architecture decisions and design rationale in general could benefit from better modeling-tool support. We've been using Rose to create decision relationship maps; however, the next step would be to establish relationships between the decision entities and the components they influence (packages, classes, deployment units, and so on). Incorporating ideas such as design decision trees could help in organizing and traversing decisions.[14]

We see opportunities for using decisions to document traceability between requirements and technical implementation. Since decisions represent a solution's major building blocks, it makes sense to use them to measure how well the solution satisfies its purpose and helps identify conflicting decisions. You can also use the approach to prove that alternatives, documented in the decisions, don't provide the desired qualities. In "Modeling Conflict Management in Design: An Explicit Approach," Frances Brazier and her colleagues provide an approach for managing conflicting decisions that should be leveraged as part of a holistic traceability methodology.[15]

Finally, to quote Ruth Malan and Dana Bredemeyer, an architecture is successful if "it is actually used in developing systems that deliver strategic value."[16] The challenge of architecture construction isn't technical, but gaining various stakeholders' consensus on the architecture direction. We've found that socializing architecture decisions provides a mechanism for meeting this challenge.

References

1. J. Putman, *Architecting With RM-ODP*, Prentice Hall, 2001.
2. P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 42–50.
3. P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, 2000.
4. R. Malan and D. Bredemeyer, "Less is More with Minimalist Architecture," *IEEE IT Professional*, vol. 4, no. 5, 2002, pp. 46–48.

5. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 2003.
6. G. Flurry and W. Vicknair, "The IBM Application Framework for e-Business," *IBM Systems J.*, vol. 40, no. 1, 2001, pp. 8–24.
7. P. Louridas and P. Loucopoulos, "A Generic Model for Reflective Design," *ACM Trans. Software Eng. and Methodology*, vol. 9, no. 2, 2000, pp. 199–237.
8. J.K. Kyaruzi and J. van Katwijk, "Beyond Components-Connections-Constraints: Dealing with Software Architecture Difficulties," *Proc. 14th IEEE Int'l Conf. Automated Software Eng.*, IEEE Press, 1999, pp. 235–242.
9. A. Akerman, J. Tyree, and L. Coglianese, "An Architectural Process for System Evolution," *Enterprise Architect Magazine*, vol. 2, no. 1, 2004, www.ftponline.com/ea/magazine/spring/features/aakerman.
10. *IEEE Std. 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE, 2000.
11. S. Komiya, "A Model for the Recording and Reuse of Software Design Decisions and Decision Rationale," *Proc. 3rd Int'l Conf. Software Reuse: Advances in Software Reusability*, IEEE Press, 1994, pp. 200–201.
12. L. Karsenty, "An Empirical Evaluation of Design Rationale Documents," *Proc. SIGCHI Conf. Human Factors in Computing Systems: Common Ground* (CHI 96), M.J. Tauber, ed., ACM Press, 1996, pp. 150–156.
13. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Pearson Education, 2003.
14. J. Savolainen, "Tools for Design Rationale Documentation in the Development of a Product Family," 1999, www.ece.utexas.edu/~perry/prof/wicsa1/final/savolainen.pdf.
15. F.M.T. Brazier et al., "Modeling Conflict Management in Design: An Explicit Approach," *Artificial Intelligence for Eng. Design, Analysis and Manufacturing* (AIEDAM), special issue on conflict management in design, vol. 9, no. 4, 1995, pp. 353–366.
16. R. Malan and D. Bredemeyer, "Software Architecture: Central Concerns, Key Decisions," 2002, www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF.

About the Authors

Jeff Tyree is a solutions architect at Capital One Financial. His research interests include large-scale system design, system evolution processes, refactoring, and performance engineering. He received his master's degree in mathematics from the University of Tennessee, Knoxville. Contact him at 11013 W. Broad St., Glen Allen, VA 23060; jeff.tyree@capitalone.com.

Art Akerman is a system architect at Capital One Financial and a practicing member of the World Wide Institute of Software Architects. His research interests include creation of formal education programs for software architects, communicating architecture to the development community, and making architecture development more practical and less time consuming. He received his master's degree in management of information technology from the University of Virginia. Contact him at 11013 W. Broad St., Glen Allen, VA 23060; art.akerman@capitalone.com.