

Modeling and sharing architectural decisions, Part 1: Concepts

From retrospective decision capturing to proactive decision modeling

Skill Level: Intermediate

[Olaf Zimmermann \(ozimmer@de.ibm.com\)](mailto:ozimmer@de.ibm.com)
Executive IT Architect and Research Staff Member
IBM

[Nelly Schuster \(nes@zurich.ibm.com\)](mailto:nes@zurich.ibm.com)
Software Engineer
IBM

[Peter Eeles \(peter.eeles@uk.ibm.com\)](mailto:peter.eeles@uk.ibm.com)
Executive IT Architect
IBM

05 Aug 2008

Architectural decisions capture precious knowledge that is worth sharing. Text templates and tools designed solely for documentation purposes fail to facilitate such knowledge exchange. In this series of articles, learn about a domain meta model specifically designed to capture and share architectural decisions, explore a reusable architectural decision model for SOA, and find out more about Architectural Decision Knowledge Wiki, a Web 2.0 collaboration platform. This first article outlines why and how architects should consciously identify, make, and enforce architectural decisions.

Introduction

This series of articles is about architectural decision modeling and its application to Service-Oriented Architecture (SOA) design and deployment. In this article, learn what architectural decisions are, what their relation with architectural viewpoints is,

and why IT architects should consciously identify, make, and enforce them. This article introduces architectural viewpoints, existing templates and tools for capturing architectural decisions, and a domain meta model that was extended and optimized for decision-maker collaboration and reuse. You'll also learn about the Architectural Decision Knowledge Wiki, a prototype that supports our decision modeling concepts.

Recap: Architectural viewpoints

As architects, we view systems from different viewpoints. An example of this approach is the “4+1 views model of software architecture” (see [Resources](#)) defined by Philippe Kruchten, one of the originators of the IBM Rational® Unified Process® (RUP®). This model defines five viewpoints: the logical, process, development, physical, and scenario viewpoints. Kruchten's work fed into IEEE standard 1471 (equivalent to ISO / IEC, 42010), the standard for documenting architecture descriptions.

Documentation of views may take a variety of forms, such as component-and-connector pictures, Unified Modeling Language (UML) diagrams, or domain-specific architecture description languages. For example, the focus on a process view, where concurrency concerns are paramount, is very different from a focus on the physical view, where distribution concerns are more important.

Considering an architecture from multiple, concurrent views is particularly relevant to the study of architectural decision modeling. By looking at separate views of the architecture, you can home in on the decisions that need to be made with respect to each of those views.

Architectural decisions

Generally, architectural decisions are conscious design decisions that may concern the software system as a whole, or one or more core components. Architectural decisions are significant because they may directly or indirectly determine whether a system meets its *nonfunctional requirements*, such as software quality attributes.

Nonfunctional requirements (NFRs)

[ISO 9126-2001](#) defines 27 software quality attributes in areas such as functionality, usability, reliability, efficiency, maintainability, and portability. Many more taxonomies exist. Other types of NFRs are legacy constraints and environmental factors.

Models and notations defined by an architecture description standard, such as the 4+1 views model, help capture and communicate the results of the design process. Architectural decisions, on the other hand, typically answer “why” questions, not just

“what” and “how” questions. Often, architectural decision rationale is captured in textual form, either free form or structured with the help of templates or spreadsheets.

The scope of architectural decisions varies. Many architectural decisions are specific to a particular view, such as the architectural decision to assign a certain functional responsibility to a logical component. Others serve as glue between different views. There are benefits to considering decisions related to the relationship between views. An example is the rationale for placing a component on a node which implies, using the 4+1 views model, a relationship between the logical and physical views. Thus, architects make decisions in every aspect of their work. An architectural decision log can be an overarching *über-view* of the architecture, because it explains the rationale for the design described in all of the views.

Example

Selecting an overall logical layering scheme for a system under construction qualifies as an architectural decision, according to the definition in the previous section. First and foremost, layering is an architectural pattern, as described in *Pattern-Oriented Software Architecture* (see [Resources](#)). Selecting the layers pattern is an architectural decision driven by the desire for organization and flexibility; components in a particular layer should talk only to one another and to lower layers. Interfaces that encapsulate and hide implementation details isolate the components in different layers from one another as much as possible. If the pattern is applied, layer implementations can seamlessly switch from one technology to another, and there is no need to rewrite the elements contained within one of the adjacent layers.

When developing enterprise applications, some of the layering alternatives are:

Single layer

All logic in the application is kept together. Logical components are not assigned to any particular layer.

Presentation domain data source layering

This layering encompasses the traditional three display, processing, and persistence layers used in logical enterprise application design.

The *presentation layer* contains all rich or thin client logic for displaying user interfaces to human users. Components in this layer may belong to both the client and the server tier in a physically distributed system, for instance a Web browser and Java™ servlets.

The *domain layer* contains business logic, which is activated in response to stimuli from the presentation layer. Examples are executable business

processes, calculations, and modifications of business objects.

The *data source layer* makes enterprise data, such as customer profiles and order information, persist. It also provides interfaces that let the domain layer access this data when executing its logic.

The names were introduced by Kyle Brown *et al* in *Enterprise Java Programming with IBM WebSphere* and adopted by Martin Fowler in *Patterns of Enterprise Application Architecture* (see [Resources](#)).

Enterprise SOA layering

The following four layers were defined by Dirk Krafzig in *Enterprise SOA* (see [Resources](#)):

- The *enterprise layer* contains application front ends and public enterprise services.
- The *process layer* defines long-running, multi-step, multi-actor business processes, such as order management or load processing, which are composed of basic services and make use of intermediary services.
- The *intermediary layer* provides integration capabilities and other technical capabilities, such as authentication, authorization, and logging.
- The *basic layer* encompasses business services that are discrete, atomic units of work, each corresponding to a certain business task (activity) to be invoked as part of a process (or directly from the enterprise layer). Examples include customer address lookup and validation, profile updates, and risk calculations. Proxies for externally provided public enterprise services also reside on this layer.

SOA solution stack layering

This layering with 5+4 resource, service component, service, process, consumer, integration, quality of service (QoS), information, and governance layers was first defined by Ali Arsanjani in “Service modeling and architecture” (see [Resources](#)) and then extended by the SOA solutions stack (S3) project (see [Resources](#)). This layering scheme is not as strict as the previous two. It places integration and QoS capabilities, which may be provided by middleware, into vertical layers. By default, the vertical layers are visible to all other layers. This softens the strict hierarchy in the original layering pattern. Such a semi-strict layering scheme is easier to adhere to. However, additional architectural principles must be established to govern which interactions between layers are permitted.

The logical layering-scheme decision pertains to the logical viewpoint. Any of the

alternatives here suggest a different way of organizing the top-level components in a logical view in a strict or semi-strict hierarchy. There are many trade-offs. For instance, introducing too few layers carries the risk of violating the separation of concerns principle because coupling many components in a layer rather tightly has a negative impact on quality attributes such as maintainability and portability. However, if you introduce many layers, you must be careful not to introduce too many levels of indirections, which when projected to the physical viewpoint, might decrease performance.

Logical layering versus physical distribution

It's important not to confuse the logical and physical viewpoints in this discussion. Logical layering does not imply physical distribution.

Introducing many physical layers (often called tiers) *might* be an issue, depending on the NFRs for the system under construction. Fowler's *First Law of Distributed Object Design* states "Don't distribute your objects!"

Importance of capturing and sharing architectural decisions

While the selected logical layering scheme is (hopefully) exposed in the component models representing the logical view of the system (such as a UML class diagram), the reasons for preferring one layering scheme over another are probably less explicit. In some cases, architects might prefer this knowledge to remain tacit. In most cases, though, it's very useful to explicitly capture and share the knowledge.

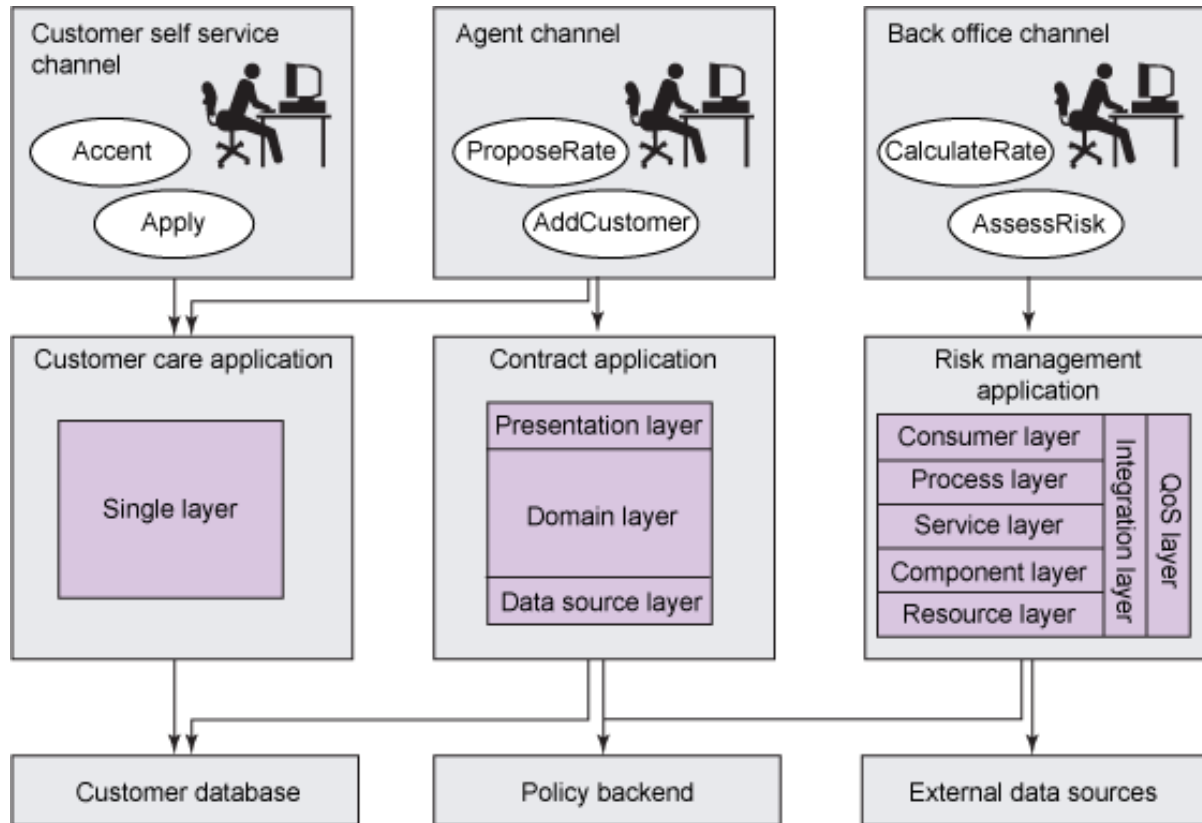
Studying decisions is an excellent way to learn from more experienced architects. Just like final design diagrams, decision logs can be highly informative because they elaborate on alternatives not chosen (and the reasons why), and they show how the architecture evolved over time because requirements changed. Harmonizing the architectural decision-capturing process simplifies reusing and learning from architectural decisions made on previous projects. Mistakes are a great opportunity to learn something, but the mistakes don't always have to be our own!

Continuing with the layering example, assume you're in charge of building an SOA to modernize three existing customer care, contract, and risk management applications, and associated storage, server, network, and middleware infrastructure for an insurance company. The insurance company reaches out to its customers directly, but also indirectly touches them through independent agents. Both customers and agents are geographically distributed. Conversations among customers, agents, and the insurance company can run for days and months during contract negotiations and claim handling. A business domain analyst has modeled these conversations in a business process model (BPM).

Assume that the enterprise architect for the insurance company has defined one

layering scheme: enterprise SOA layering. But the three existing applications were developed independently of one another, do not acknowledge this enterprisewide architectural decision, and use single layer, presentation domain data source layering and SOA solution stack layering, respectively. Figure 1 shows the application landscape.

Figure 1. Layering examples



When you start to redesign this application landscape for SOA, it's critical to have agreement on a common layering scheme—or at least make clear in which part of the system which scheme is used and how the different schemes relate to each other. Rework will be an obvious consequence of not identifying this architectural decision early enough, or deferring it because it's a tough call. For example, diagrams have to be redrawn rather late in the project, or additional documentation is required to explain the relationships among the layers in the different layering schemes. A more severe consequence is that bloated and ambiguous terminology causes communication problems within the project team.

Even worse, using multiple layering schemes might cause application programming interface (API) mismatches. Such misunderstandings and interface mismatches make different parts of the system rather difficult to integrate. In turn, it might become impossible to reuse and share functions across application and layer boundaries. In the example, it's not clear how a Java servlet residing in the presentation layer of the contract application should access a component provided

by the risk management application: via the consumer layer, the process layer, the service layer, the integration layer, or all of them?

The boundaries of logical layers are often used to implement technical responsibilities, such as access management (user authentication and authorization), parameter validation, transaction management, and error handling. If the logical layers within a system are not aligned with one another, these functions might be implemented in very different ways (for example, in terms of interface granularity and connection management). Such differences lead to significant integration efforts and sometimes a failure to meet the system's NFRs.

Decision capturing problems

When capturing and sharing such architectural decisions, such as logical layering scheme, three concrete problems commonly occurring on development and integration projects are:

Decision identification

Practitioners often have to spend a lot of time finding out what they have to worry about during their architecture design work, even when not operating on a green field. They have to pull the required knowledge from the vast body of knowledge available and run the risk of reinventing the wheel. In the insurance SOA case, a high-level architectural decision, such as logical layering scheme, might be simple to identify.

However, many lower level architectural decisions dealing with concerns of transactionality, security, or reliability have to be made to ensure the integrity of business processes. Such decisions are not easy to detect, even for experienced architects, if the available design alternatives are not made explicit in technology standards and product documentation.

Decision making

For most design issues, multiple architecture alternatives exist. Evaluating all of them takes time. In the example, almost every article and book on SOA suggests its own layering scheme. It's hard to make an informed decision without experience with the alternatives. Consequently, a random choice might be made, or an implicit one dictated by a particular consulting method, development tool, or runtime platform. With such an ad hoc approach to decision making, NFRs are not always addressed properly. When requirements change (and they always do), it's hard to analyze the impact.

Decision enforcement

It's also known as a control problem. Often, the code or the installation turns out not to be in line with what the architect specified in UML models or informal component-and-connector diagrams. Lack of architectural integrity can be a

consequence of such a disconnect.

For example, the recommended practice is to reflect the logical layering scheme chosen at the architectural level in naming conventions and other organizing principles at the development level, such as XML namespaces in Web Services Description Language (WSDL) and XML schema files, as well as Java package names. If you break this convention, it's hard to trace how the code and deployment artifacts relate back to the architectural building blocks.

Decision identification, decision making, and decision enforcement activities are often isolated from one another. There can be disconnects between logical and physical models. There can also be disconnects between practitioners working for different lines of business in one company (such as presales and project architects), or working for different companies (consumers and providers of professional services). Such disconnects can affect cost, quality, and risk level. Architectural decisions are likely to be treated as one-of-a-kind, with personal experience and gut feel driving the decision making.

Architectural decisions should be identified, made, and enforced systematically. With the many, many decisions to be made, and the often subtle dependencies among them, you need methods and tools for capturing and sharing them.

Current tools and methods to capture architectural decisions

Given the importance of making the right architectural decisions, it's surprising how little attention they have received in software engineering and software architecture.

The term *architectural decision* is mentioned, but not defined in detail, in [Software Architecture in Practice](#). In 2000, [Applied Software Architecture](#) proposed the concept of *issue cards* and defined a simple template for them. Each issue deals with one or more architectural decisions. The book gives many informal examples of issues, but it does not define a full process for issue management.

The IBM Global Services Method, which has been used with IBM clients since 1998 (now known as IBM Unified Method Framework, or UMF), defines a work product called ARC 100 that defines a text table template for decision capturing. Having worked with an IBM reference architecture that comes with a completed ARC 100 artifact containing architectural decisions about Web application design, [Art Akerman and Jeff Tyree](#) defined another rich decision-capturing template structured into 13 sections: issue, decision, status, group, assumptions, constraints, positions, argument, implications, related decisions, related artifacts, related principles, and notes. Later on, they proposed an entire ontology to support development of software architectures.

In 2006, [Philippe Kruchten](#), [Patricia Lago](#), and [Hans van Vliet](#) presented an ontology for architectural decisions, defining the following:

- Types, such as executive, existence, and property architectural decisions
- Dependencies, such as constrains, forbids, enables, subsumes, conflictsWith, overrides, comprises, isAnAlternativeTo, isBoundTo, and isRelatedTo
- A decision life cycle: idea, tentative, decided, approved, challenged, rejected, and obsolete

The article "[Using Patterns to Capture Architectural Decisions](#)" proposed how to minimize the capturing effort by referencing already published patterns, such as layers.

Defining templates and referencing patterns will help you on the road to more systematic and rigorous decision capturing. But it will not remove real-world inhibitors for sustainable and maintainable decision sharing. You'd still suffer from budget and scheduling problems, a lack of tools, and no immediate benefit.

From text-based decision capturing to decision modeling

If decision capturing and sharing are useful, but hard and therefore not practiced, what is missing to cut the Gordian Knot? Two of the missing pieces are:

- A *decision modeling framework* (process)
- A rich *domain meta model* that defines the decision knowledge more thoroughly than the existing text templates

Such a framework and domain meta model can also support the development of tools for architectural decision capturing and sharing.

Why should we model architectural decisions rather than capture them in structured or unstructured text? On large projects, such as the insurance SOA in the example, there are hundreds—if not thousands—of architectural decisions to make and document.

There are many dependencies among the architectural decisions.

Two important types of dependencies are:

- Logical constraints, such as *(in)compatibilities* between alternatives.
- Temporal dependencies, such as *triggers* and *prunes*.

For example, a logical constraint is: if SOA is chosen as an architectural style in one

architectural decision, by definition the remote invocation paradigm must be document messages and not remote objects.

When you select a pattern from a pattern language or catalog, you are making an architectural decision. Selecting an architectural pattern such as Broker, introduced by [Frank Buschmann](#) *et al*, triggers the selection of a pattern variant, such as hub or direct connection. However, not all architectural decisions deal with pattern selection. In this example, a technology selection decision for one or more Broker technologies and transport protocols, such as HTTP or Java Messaging Service (JMS), follows the pattern selection.

An example of pruning is: when PHP is the language of choice to implement Web service providers or other logical components, there's no need to decide on a particular Java Virtual Machine (JVM) version.

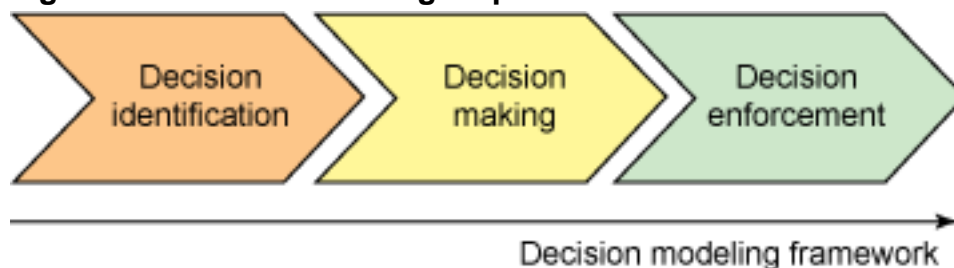
Designing a complex software system is a collaborative effort. It involves multiple external and internal stakeholders, all of whom contribute their personal decision drivers and other knowledge. The text template approach does not scale, and it also leads to collaboration and maintenance problems (think of manual management of cross-references and document-level version control).

Because text-based decision logs have scalability and collaboration issues, on many projects an architectural decision document is begun at project initiation time, but is not kept up-to-date (especially when there are delivery pressures and change management issues). However, an out-of-date decision log is worthless, or might even have a negative impact if someone tries to consult it later in the project or during maintenance.

Decision-making process

To overcome the limitations of template-based decision capturing, we first structure the architectural decision-making process into three conceptual steps, as shown in Figure 2. It's straightforward and necessary to set the scene for later stages.

Figure 2. Decision modeling steps



1. *Decision identification* scopes the architecture design work on a software development project. Requirements and earlier decisions trigger the

identification of individual decisions in each phase that requires architecture design activities by a particular role. A catalog of required decisions can be assigned to the design model elements appearing in the pattern language or reference architecture in use.

2. During *decision making*, architects select alternatives according to certain decision drivers. This step is the core of the process. Architects are primarily responsible for making sound technical decisions and documenting the architecture from the 4+1 viewpoints.
3. *Decision enforcement* involves documenting decisions and sharing results of the decision-making step with stakeholders and the project team to subsequently adopt. Recording the decisions is a basic form of decision enforcement. Monitoring whether the decisions ensure that the system under construction will meet its design goals (architectural evaluation) also falls in this step.

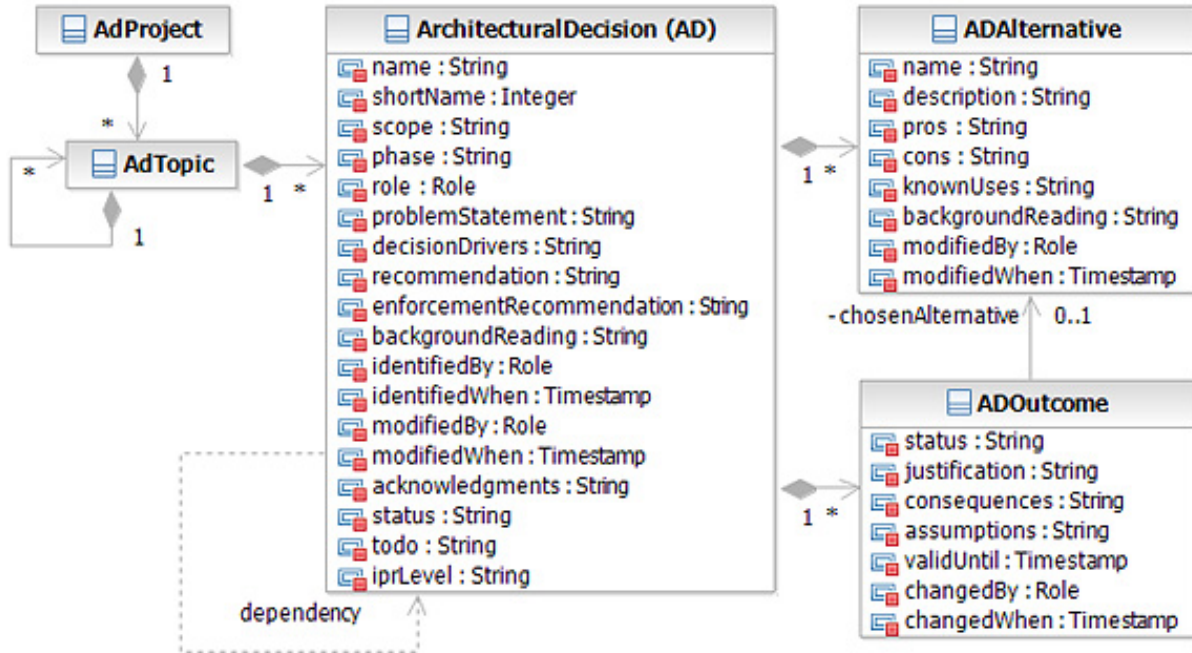
Each architectural decision (AD) goes through these three steps. Frequently, multiple decisions are identified, made, and enforced together. The decision modeling steps are themselves subsumed into the overall development process, regardless of whether the process follows a waterfall, iterative, or agile approach, or where the project is in the overall time line. The concepts discussed in this article can therefore be applied to many different methods, including the [Open Unified Process \(OpenUP\)](#), RUP, UMF, Scrum, Lean, and so on.

A meta model for decision capturing and modeling

The next step is to agree on a format for the decisions to be identified, made, and enforced. Following a best-of-breed approach, we studied existing templates, such as those introduced earlier in this article, as well as the “Grouping ADs by Topic” viewpoint in the [Architects’ Workbench](#), an Eclipse-based tool that implements the meta model defined by the IBM Architecture Description Standard (ADS). We also reflected on our own decision-capturing practices to identify additional attributes, for instance, those supporting decision maker collaboration and architectural decision reuse.

The domain meta model in Figure 3 is the result of several harvesting and refactoring iterations. We have been using it since 2006.

Figure 3. Domain meta model



There are three core domain entities:

- [ArchitecturalDecision \(AD\)](#)
- [ADAlternative](#)
- [ADOutcome](#)

AD and ADAlternative provide reusable background information only. ADOutcome instances then can be created dynamically, referring to design model element instances (for example, appearing in a logical view).

The rationale for this modeling choice is that the same architectural decision might pertain to several elements in a design model, such as multiple business processes. For example, the BPM for the sample insurance SOA might state that five business processes have to be implemented as a set of composed Web services. While attributes such as `problemStatement`, `backgroundReading`, and `ADrecommendation` are the same for all five processes when deciding for a certain remote invocation paradigm, the justification for having chosen a particular alternative might differ.

ArchitecturalDecision (AD) entity

Every architectural decision has a `name` and a `shortName`. The `problemStatement` describes what the decision is about. It characterizes an architectural decision issue on an introductory level, often in question form. The `backgroundReading` attribute contains links to external resources such as books, papers, or Web pages that can provide more information about the decision context.

The `decisionDrivers` attribute lists types of (N)FRs. Personal experiences and preferences are popular decision drivers. This attribute provides objective information about the key factors influencing the decision.

Decision drivers

Decision driver is a neutral term. You have probably come across many decision drivers, but not necessarily this term. Decision drivers may include NFRs such as performance and scalability, as well as nontechnical constraints like project budget and availability of skills.

The patterns community uses the term *forces* synonymously with *decision drivers*, while those in the software architecture research arena use *quality attribute*.

Three attributes link the AD entity with the software engineering method in use:

- `scope` captures the type of design model element the architectural decision pertains to. You can reference design model element types defined in any viewpoint here. An example is the architectural decision for a transport binding, which relates to the design of individual (Web) service operations, thus its scope is "operation." A decision that does not relate to a specific element in the design model might have scope "project."
- `phase` denotes the phase in which the architectural decision has to be decided, providing a link to general-purpose methodologies such as RUP or UMF. A decision such as remote invocation paradigm is made in an early phase, such as "solution outline." In solution outline, high-level requirements are gathered and an initial architecture is defined to comprehend the outline of the system under construction. Subsequent phases might be "macro design" and "micro design."
- `role` captures the type of architect responsible for the architectural decision, such as "application architect," "integration architect," or "infrastructure architect."

The `recommendation` attribute is an important feature in the domain meta model. It can be used to establish a general best practices guideline, or rule of thumb, for selecting an alternative under certain decision drivers. Most architects appreciate simple, easy-to-follow advice about technical design issues. However, such advice is often free form, and it's not always clear which NFRs drove the design behind the best practice guideline.

If a team blindly follows best practices advice, the requirements and architectural

context of the best practice origin might not match the best practice usage. The `recommendation` attribute puts the best practice recommendation in perspective and clarifies that architectural thinking and conscious decision making is still required. In some cases it's OK to break the best practice rule if you can justify it by the project-specific decision drivers—and if the alternative is technically feasible and free of undesired consequences (evaluated through prototyping, perhaps).

recommendation provides a home for best practices

Currently, even the most splendid best practices advice is shared with informal pictures or bulleted lists and often omits information about decision drivers and rationale.

Our domain meta model lets you attach the best practice `recommendation` to a required AD, which defines the context in which it's applicable. Such context-aware recommendations are *grounded best practices*.

The `enforcementRecommendation` gives a hint as to which enforcement mechanism to choose. For instance, enforcing the decision in development manually, through coaching or code reviews, or with the help of model transformation and code generation tools. Each architectural decision is identified by someone (`identifiedBy`) at a particular time (`identifiedWhen`). Modifications of an architectural decision are captured with the attributes `modifiedBy` and `modifiedWhen`. As editorial information, intellectual property rights (`iprLevel`), contributors (`acknowledgments`), maturity (`status`), and remaining tasks (`todo`) regarding the decision description can be captured.

ADAlternative

An architectural decision can include one or more ADAlternatives, such as architecture and design patterns, technologies and standards, and tools and run time platforms, depending on the type of architectural decision they belong to.

ADAlternative entities have `name`, `description`, `pros` and `cons` attributes. `pros` and `cons` comment how the ADAlternative scores relative to the decision drivers. They also let you capture information about `knownUses` and `backgroundReading` references. Modifications are captured with the attributes `modifiedBy` and `modifiedWhen`.

ADOutcome

You can instantiate several ADOutcome entities, also called decision instances, for an AD entity (architectural decision issue). The decision instance then refers to a chosen alternative, and includes `justification`, `consequences`, and `assumptions`. The `justification` content should refer to AD

decisionDrivers, ADAlternative pros and cons, and AD recommendation.

changedBy, changedWhen, and status model the decision ownership and life cycle. An ADOutcome entity, also called a decision instance, might have one of the following states:

open	A description and several alternatives are captured, but the decision was not made yet.
decided	Now an alternative was chosen.
confirmed	The responsible person approved the decision.
rejected	The made decision was rejected due to some reasons.

Once made, an ADOutcome entity is validUntil a certain date.

Dependencies between ADs and ADAlternatives

Decision dependencies are explicitly modeled as UML associations between architectural decisions. The dependencies might have different types:

influences

This dependency type expresses a generic dependency. It is bidirectional.

decomposesInto

A complex design problem is split into several smaller, more manageable ones for a divide-and-conquer strategy to problem solving.

refinedBy

This dependency type describes dependencies between architectural decisions located in different levels of abstraction and refinement, for example, conceptual, technology, or vendor asset level. AD levels will be discussed in more detail in the next part of this series.

forces, isIncompatibleWith

The selection of an alternative in one decision might force the selection of another one in another decision. A slightly weaker statement is that two alternatives might be incompatible with each other.

triggers

As soon as architectural decision A is decided, architectural decision B can be decided. This influence type is used for temporal dependencies.

prunes

If architectural decision A is decided for a certain alternative, architectural decision B is no longer relevant and can be marked as such.

Managing decision dependencies

Managing dependencies between decisions and, optionally, also their alternatives is a complex, labor-intensive effort. Tool support for decision modeling is required; manual, text template-based decision capturing falls short.

Example

In the insurance SOA example, the logical layering scheme architectural decision might be resolved as shown below.

Sample decision with ADOutcome information

AD shortname	Exe-01	AD name	Logical Layering Schema		
Topic hierarchy	InsuranceSoaProject - BusinessExecutiveLevel - ExecutiveDecisions - KeyTechnologyProcessAndToolDecisions				
Scope	Project	Phase	Solution outline	Role	Lead architect
Problem statement	Which logical layering scheme, possibly defined in a public or internal reference architecture, should frame the design work? Architectural consistency starts with speaking one language—and hearing the same things.				
Decision drivers	Software engineering method and viewpoint catalog chosen. Standards for industry or application genre, and selected reference architecture. Existing client assets and enterprise architecture efforts. Vendor preferences.				
Alternatives	[1] Single layer [2] Presentation Domain Data Source Layering, introduced by Brown and Fowler [3] Service layering scheme in "Enterprise SOA" [4] IBM SOA Solution Stack (S3), superseding 5+2 SOMA layers [5] Other SOA layering schemes, such as from CBDI or SOA in Practice [6] Not applicable				
Recommendation	The S3 layering and the service type taxonomies from the <i>Enterprise SOA</i> and <i>SOA in Practice</i> are useful. The S3 layering is intuitive and can be mapped to older layering schemes, such as Presentation Domain Data Source Layering.				

Decision Outcomes	Decision instance: SOA modernization project, risk management application subproject										
	<table border="1"> <tr> <td>Status</td> <td>decided by ArchieTek on 2008-08-05 16:48:59.951000 Valid until July 31, 2009</td> </tr> <tr> <td>Chosen alternative</td> <td>[4] IBM SOA Solution Stack (S3), superseding 5+2 SOMA layers</td> </tr> <tr> <td>Justification</td> <td>The recommendation works for us. It's an SOA project, following the SOMA method, which also works with the S3 layers. Not aware of any mature insurance industry standard.</td> </tr> <tr> <td>Consequences</td> <td>Will need to define a mapping from "Enterprise SOA" scheme and the presentation, domain, and data source layers to S3.</td> </tr> <tr> <td>Assumptions</td> <td>For now, will only use the original 5+2 layers. Are going forward with IBM GBS.</td> </tr> </table>	Status	decided by ArchieTek on 2008-08-05 16:48:59.951000 Valid until July 31, 2009	Chosen alternative	[4] IBM SOA Solution Stack (S3), superseding 5+2 SOMA layers	Justification	The recommendation works for us. It's an SOA project, following the SOMA method, which also works with the S3 layers. Not aware of any mature insurance industry standard.	Consequences	Will need to define a mapping from "Enterprise SOA" scheme and the presentation, domain, and data source layers to S3.	Assumptions	For now, will only use the original 5+2 layers. Are going forward with IBM GBS.
Status	decided by ArchieTek on 2008-08-05 16:48:59.951000 Valid until July 31, 2009										
Chosen alternative	[4] IBM SOA Solution Stack (S3), superseding 5+2 SOMA layers										
Justification	The recommendation works for us. It's an SOA project, following the SOMA method, which also works with the S3 layers. Not aware of any mature insurance industry standard.										
Consequences	Will need to define a mapping from "Enterprise SOA" scheme and the presentation, domain, and data source layers to S3.										
Assumptions	For now, will only use the original 5+2 layers. Are going forward with IBM GBS.										
Background reading	" Design an SOA solution using a reference architecture " introduces the S3 layering. Layering pattern is explained in depth in pattern literature (in POSA series, for example).										
Related decisions	is influenced by Exe-00 PrimaryArchitecturalStyle influences Gov-01 TerminologyHardening influences Gov-02 StandardsAdoption										
Editorial information	Acknowledgments: the entire SOA project team was involved ModifiedWhen 2008-08-05 16:50:46.394000 Status: published, to be reviewed semi-annually ToDo: keep up to date										

Decision drivers, pros and cons of alternatives (not present in this example), recommendation, and justification are particularly relevant when you need to capture design rationale. The justification attribute:

- Must answer "why" precisely, avoiding common sense statements.
- Should use the decision drivers and comment why the recommendation was followed or not.
- Must refer to actual project requirements, and not just reusable background information captured as decision drivers.

The following table gives some general examples for good and bad justifications (not specific to any AD).

Positive and negative examples of justifications

Decision driver type	Valid justification	Counter example
Wants and needs of external stakeholders	Alternative A best meets user expectations and functional	End users want it, but no evidence for a pressing

	requirements as documented in user stories, use cases, and business process model.	business need. Technical project team never challenged the need for this feature. Technical design is prescribed in the requirements documents.
Architecturally significant requirements	Nonfunctional requirement XYZ has higher weight than any other requirement and must be addressed; only alternative A meets it.	Do not have any strong requirements that would favor one of the design options, but alternative B is the market trend. Using it will reflect well on the team.
Conflicting decision drivers and alternatives	Performed a trade-off analysis, and alternative A scored best. Prototype showed that it's good enough to solve the given design problem and has acceptable negative consequences.	Only had time to review two design options and did not conduct any hands-on experiments. Alternative B does not seem to perform well, according to information online. Let's try alternative A.
Reuse of an earlier design	Facing the same or very similar NFRs as successfully completed project XYZ. Alternative A worked well there. A reusable asset of high quality is available to the team.	We've always done it like that. Everybody seems to go this way these days; there's a lot of momentum for this technology.
Prefer do-it-yourself over commercial off-the-shelf (build over buy)	Two cornerstones of our IT strategy are to differentiate ourselves in selected application areas, and remain master of our destiny by avoiding vendor lock-in. None of the evaluated software both meets our functional requirements and fits into our application landscape. We analyzed customization and maintenance efforts and concluded that related cost will be in the same range as custom development.	Price of software package seems high, though we did not investigate total cost of ownership (TCO) in detail. Prefer to build our own middleware so we can use our existing application development resources.
Anticipation of future needs	Change case XYZ describes a feature we don't need in the first release but is in plan for next release. Predict that concurrent requests will be x per second shortly after global rollout of the solution, planned for Q1/2009.	Have to be ready for any future change in technology standards and in data models. All quality attributes matter, and quality attribute XYZ is always the most important for any software-intensive system.

Architectural decisions and reusable assets

A big benefit of modeling architectural decisions is to capture knowledge associated with the system under consideration. Another benefit is that architectural decisions can be treated as reusable assets in their own right. Decision assets can be shared between system development efforts, and even across entire organizations and enterprises. A widespread adoption of architectural decisions as reusable assets requires a set of relevant standards to ensure a consistent representation that can be enforced in appropriate tools.

Reusable SOA Decision Model

This article introduces a domain meta model to partially address elements that are specific to the definition of architectural decisions. A future article will introduce a reusable architectural decision model for SOA design and deployment.

Architectural decisions aren't the only reusable assets we encounter on development projects. Others include: patterns (of which there are many types), architectural styles, reference architectures, application frameworks, architectural mechanisms, packaged applications, and legacy applications. See "[Understanding architectural assets](#)" for a summary.

In the big picture of a complete software development effort, there is a need for standards to define all types of reusable asset, including reusable architectural decisions. A key standard is the [Reusable Asset Specification \(RAS\)](#), which defines two aspects of a reuse strategy. The first is a standard for describing a reusable asset. The second is a standard for an interface to a RAS-compliant repository, called a RAS repository service. A nice feature of the RAS standard is that it allows you to define profiles of different types of assets. You can define the precise attributes and semantics required for an architectural decision RAS asset and, of course, make the resulting "Architectural decision RAS profile" available in a tool that supports the RAS standard.

One such tool is IBM Rational Asset Manager. It implements the RAS standard and provides features that support a strategic reuse initiative, such as the ability to:

- Control the life cycle of any asset (from submitted to approved to retired).
- Provide a rating for each asset.
- Keep track of the practitioners who have used the asset already.

Of course, there's more to achieving successful asset reuse in an organization, but that's beyond the scope of this article. This article did not discuss the process for creating or applying assets, which would include: roles in an organization that support a strategic reuse initiative, tasks you can perform (especially creation, use, and maintenance of assets), and artifacts that specifically support a reuse initiative (such as an asset catalog).

Tool support: Architectural Decision Knowledge Wiki

After you have established a domain meta model for modeling and sharing architectural decisions, tool support is next. You'll want to support the following seven primary use cases of IT architects operating in collaborative environments:

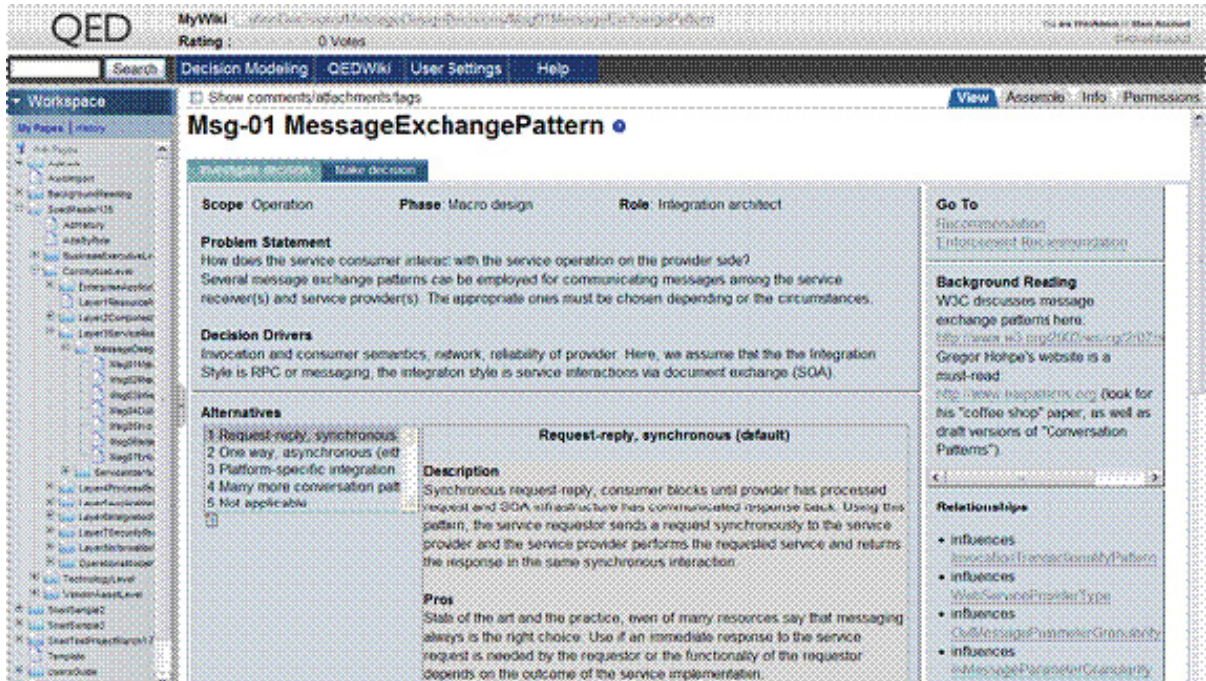
- *Obtain architectural knowledge* from third parties, such as companywide enterprise architecture groups or practitioner communities in services firms (asset consumption).
- *Adopt and filter* decision knowledge according to project specific needs: delete, update, and add architectural decisions and alternatives, and manage dependencies between decisions.
- *Delegate* decision making authorities to subsystem architects and lead developers, and support review activities with bidirectional feedback loops.
- *Involve* network of peers in search of additional architectural expertise during decision making (when creating ADOutcome instances).
- *Enforce* decision outcome with pattern-based generation of work products, such as documentation and code snippets serving as architectural templates.
- *Inject* decisions into design models, code, and deployment artifacts.
- *Share* gained architectural knowledge with third parties, such as the actors from the first use case, after having sanitized the project deliverables (asset creation).

[Architectural Decision Knowledge Wiki](#) is a research prototype that implements our domain meta model and many of these primary use cases. Architectural Decision Knowledge Wiki supports Create, Read, Update, Delete, and Search (CRUDS) operations on the entities in the domain meta model, ADProjects, ADTopics, architectural decisions, ADAAlternatives, and ADOutcomes. There is a decision dependency relationship editor. Architectural Decision Knowledge Wiki also supports import and export of decision models and ARC 100 report generation.

Architectural Decision Knowledge Wiki has powerful search and filter capabilities, leveraging domain meta model attributes such as scope, phase, and role. Once a decision space reaches a certain size (which will happen, due to the complexity of the problem and technology domains IT architects are dealing with), users do not always want or need to see everything. Text templates don't provide such powerful search capabilities. For instance, in Architectural Decision Knowledge Wiki you can search for all architectural decisions the integration architect handles in the macro design phase of an SOA project when designing a channel.

Figure 4 shows a decision captured in Architectural Decision Knowledge Wiki. On the left is a Workspace explorer showing ADProjects, ADTopics, and architectural decision in tree form. The main part of the screen is organized according to the master-details pattern for user interface (UI) design. One page shows one architectural decision at a time and lists all alternatives (master). Details for only one ADAlternative are shown (here: request-reply). The attributes are the ones defined in the domain meta model, and the content (not bold) is project-specific. The decision deals with the selection of a message exchange pattern.

Figure 4. Decision modeling example



Summary

This article outlined the key concepts for proactive architectural decision modeling with reuse. It presented a model-based approach to decision capturing and sharing that complements the modeling of other architectural artifacts, such as logical component models and physical operational models. The approach emphasizes reuse and cross-role (even cross-project) knowledge exchange and collaboration. You learned how the three-step process and the domain meta model support these concepts. Several decision modeling examples from the SOA domain were reviewed. You also learned about the prototypical tool called Architectural Decision Knowledge Wiki.

With the domain meta model, reusable content, and collaboration tool, IT architects can better address the decision identification, making, and enforcement problems on development and integration projects. The tool and content make use of the domain meta model. Apart from these two dependencies, the model, content, and tool can

be used independently of each other. However, they unveil their full potential and strengths when combined.

After a domain meta model for decision capturing is established and tool supported, you can explore the following advanced concepts:

- Semi-automatic decision identification in requirements and high-level design models, using the decision scoping concept, creating multiple AD outcome instances per decision
- Querying a model by various attributes, such as decision drivers and status
- Defining decision types, such as pattern selection and adoption, technology selection and profiling, as well as asset selection and configuration
- Modeling decision-dependency patterns, such as top-down refinement chain and technology limitation
- Semi-automatic alignment of architectural decisions with design models and code

Stay tuned for Part 2 of this series, which will look at SOA examples and the role a reusable architectural decision model can take in SOA design and deployment.

Acknowledgments

The authors would like to acknowledge: Frank Leymann (Stuttgart University); Steve Abrams, Jim Amsden, Kyle Brown, Celso Gonzalez, Ed Grossmann, Bertrand Portier (IBM Software Group); Bernd Dammrose, Petra Kopp, Stefan Pappé (IBM Global Technology Services), Paul Bate, Peter Kaufmann, Sven Milinski, Christoph Mikšovic, Vlad Ostrowski, Christian Ringler, Jose-Matias Sanchez, Ian Turton (IBM Global Business Services); Philippe Spaas (IBM Sales and Distribution); Grady Booch, Doug Dykeman, Thomas Gschwind, Doug Kimelman, Wolfgang Kleinoeder, Jana Koehler, Jochen Küster, Ronny Polley (IBM Research).

Resources

Learn

- Read about the "[4+1 views model of software architecture](#)" approach defined by Philippe Kruchten.
- [IEEE standard 1471](#) (equivalent to ISO / IEC, 42010) is the standard for documenting architecture descriptions.
- [Pattern-Oriented Software Architecture](#), by Buschmann, Meunier, Rohnert, Sommerlad, and Stal, shows the progression and evolution of the pattern approach into a system of patterns capable of describing and documenting large-scale applications.
- [Enterprise SOA](#), by Krafzig, Banke, and Slama, describes best practices for Service-Oriented Architecture.
- In his [handbook](#) of software architecture, Grady Booch discusses decision making forces and classifies patterns.
- Read [Enterprise Java Programming with IBM WebSphere](#), by Brown *et al*, to learn more about layered enterprise applications.
- Read more by [Martin Fowler](#), including his "First Law of Distributed Object Design" and his book, [Patterns of Enterprise Application Architecture](#).
- Learn more about "[Service-oriented modeling and architecture](#)" (developerWorks, Nov 2004) from Ali Arsanjani.
- "[Design an SOA solution using a reference architecture](#)" (developerWorks, Mar 2007) discusses how to improve your development process using the SOA solution stack.
- [Software Architecture in Practice \(2nd Edition\)](#), by Bass, Clements, and Kazman, introduces the concepts and practices of software architecture.
- [Applied Software Architecture](#), by Hofmeister, Nord and Soni, provides practical guidelines and techniques for producing quality software designs.
- "[Using ontology to support development of software architectures](#)" by Akerman and Tyree, IBM Systems Journal 2006, proposes an approach to software development that focuses on architecture decisions and involves the use of ontology.
- "[Architecture decisions: demystifying architecture](#)," by Tyree and Akerman, includes a decision capturing template.
- "[An Ontology of Architectural Design Decisions in Software-Intensive Systems](#)," by Philippe Kruchten, presents a possible ontology of architectural design decisions, their attributes and relationships, for complex, software-intensive

systems.

- "[Using Patterns to Capture Architectural Decisions](#)" by Harrison, Avgeriou, and Zdun investigates various methods and tools to help architects effectively document their decisions.
- "[Model-Driven and Pattern-Based Integration of Process-Driven SOA Models](#)," by Zdun and Dustdar, proposes integration of process-driven SOA models via a model-driven software development approach that is based on proven practices documented as software patterns.
- Learn more about [Introduction to OpenUP](#), a lean Unified Process that applies iterative and incremental approaches within a structured life cycle.
- "[Architectural thinking and modeling with the Architects' Workbench](#)" presents key AWB innovations, and discusses how their design was motivated by insights into architectural work and feedback from IT architects.
- [CBDI Service Oriented Architecture Practice Portal](#) offers independent guidance for service architecture and engineering.
- [SOA in Practice - The Art of Distributed System Design](#) by Nicolai M. Josuttis gives a very precise and fundamental overview of what it means to realize a SOA in practice.
- "[Understanding architectural assets](#)" (developerWorks, Sep 2007) discusses the various kinds of reusable assets available to the software architect, explains their characteristics and interrelationships, and offers tips on how best to make use of them.
- [Reusable Asset Specification \(RAS\)](#) from the Object Management Group.
- The [SOA Decision Modeling \(SOAD\)](#) Web site has a guide to relevant publications. The QOSA 2007 paper "Reusable Architectural Decision Models for Enterprise Application Development" is a good starting point. For a closer look, read "Combining Architectural Patterns and Decisions into a Comprehensive and Comprehensible Design Method" from WICSA 2008.
- Learn about [other projects](#) of the Business Integration Technologies team at the Zurich Research Lab.
- Browse the [technology bookstore](#) for books on these and other technical topics.
- In the [Architecture area on developerWorks](#), get the resources you need to advance your skills in the architecture arena.
- Find resources to help you architect enterprise and software systems in [free IT architecture kits](#) from IBM.
- Stay current with [developerWorks technical events and webcasts](#).

Get products and technologies

- The [Architectural Decision Knowledge Wiki](#) is a research prototype that implements our domain meta model and many of the primary use cases.
- Download [IBM product evaluation versions](#) and get your hands on application development tools and middleware products from IBM® DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Participate in the [IT architecture forum](#) to exchange tips and techniques and to share other related information about the broad topic of IT architecture.
- Check out [developerWorks blogs](#) and get involved in the [developerWorks community](#).

About the authors

Olaf Zimmermann

Olaf Zimmermann is a research staff member at the IBM Zurich Research Lab. He is an Open Group Master Certified and IBM senior certified executive IT architect. His research focuses on using architectural decision models to support collaborative SOA design activities. Previously, Olaf was a solution architect, helping IBM clients design enterprise-scale SOA and Web services and Java 2 Enterprise Edition (J2EE) solutions on numerous service projects. He has also taught around the world on emerging middleware technologies. Earlier in his career, Mr. Zimmermann worked as a scientific consultant at the IBM European Networking Center (ENC) in Heidelberg, Germany. He is a regular conference speaker and a coauthor of the *Perspectives on Web Services* (Springer, 2003). Mr. Zimmermann holds a graduate "Diplom-Informatiker" degree in computer science from the Technical University in Braunschweig, Germany.

Nelly Schuster

Nelly Schuster is a member of the business integration technologies group at the IBM Zurich Research Laboratory. She holds a Diplom-Ingenieur (FH) degree in media computer science from Hochschule der Medien, Stuttgart, and Nanyang Technological University, Singapore. Ms. Schuster commenced design and implementation of the Architectural Decision Knowledge Wiki as part of her diploma thesis at the IBM Zurich Research Lab. Her current research interests include collaborative software engineering and model-driven development.

Peter Eeles

Peter Eeles is an executive IT architect working within IBM Rational. In this capacity he assists organizations in their adoption of the Rational Unified Process and the IBM development toolset in architecture-centric initiatives. Since 1985 he has spent much of his career architecting, project managing, and implementing large-scale, distributed systems. Prior to joining Rational, he was a founding member of Integrated Objects, where he was responsible for the development of a distributed object infrastructure. He coauthored *Building J2EE Applications with the Rational Unified Process* (Addison-Wesley, 2002) and *Building Business Objects* (John Wiley and Sons, 1998) and was a contributing author to *Software Architectures* (Springer-Verlag, 1999).

Trademarks

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Rational Unified Process, RUP, Tivoli, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.