Reusable Architectural Decision Models for Enterprise Application Development

Olaf Zimmermann[1], Thomas Gschwind[1], Jochen Küster[1],
Frank Leymann[2], and Nelly Schuster[1]

[1] IBM Research GmbH
Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland
{olz, thg, jku, nes}@zurich.ibm.com
[2] Universität Stuttgart, Institute of Architecture of Application Systems
Universitätsstraße 38, 70569 Stuttgart, Germany
frank.leymann@iaas.uni-stuttgart.de

Abstract. In enterprise application development and other software construction projects, a critical success factor is to make sound architectural decisions. Text templates and tool support for capturing architectural decisions exist, but have failed to reach broad adoption so far. One of the inhibitors we perceived on large-scale industry projects is that architectural decision capturing is regarded as a retrospective and therefore unwelcome documentation task which does not provide any benefit during the original design work. A major problem of such a retrospective approach is that the decision rationale is not available to decision makers when they identify, make, and enforce decisions. Often a large, possibly distributed, community of decision makers is involved in these three steps. In this paper, we propose a new conceptual framework for proactive decision identification, decision maker collaboration, and decision enforcement. Based on a meta model capturing reuse and collaboration aspects explicitly, our framework instantiates decision models from requirements models and reusable decision templates. These templates capture knowledge gained on other projects employing the same architectural style. As an exemplary application of these concepts to service-oriented architecture shows, reusable architectural decision models can speed up the decision identification and improve the quality of the decision making. Reusable architectural decision models can also simplify the exchange of architecture design rationale within and between project teams, and expose decision outcome as model transformation parameters in model-driven software development.

Keywords: Architectural decision, architectural knowledge, MDA, SOA.

1 Introduction

Having been neglected both in academia and industry for a long time, the importance of *architectural decision capturing* is now widely acknowledged [15][20][28]. However, existing work focuses on capturing and representing decisions that have been made already. Little emphasis is spent on anticipating the required decisions based on

S. Overhage et al. (Eds.): QoSA 2007, LNCS 4880, pp. 15 – 32, 2007.

experience from previous projects, on recommending proven decision making techniques for these decisions, and on team collaboration aspects. In collaborative environments, decision making responsibilities are assigned to various team members; consensus must be found, and decision outcome communicated.

As a consequence, capturing architectural decisions remains a challenge for practicing architects. Reported inhibitors for capturing decisions include no appreciation from project sponsors, lack of time, and insufficient tool support [27]. Hence, intuition often is the only, but not always a suitable, decision driver; there is no systematic reuse of already gained knowledge. This lack of rigor leads to acceptance issues and quality problems with the software architectures under construction.

This paper aims to alleviate these problems by proposing a conceptual framework for three decision capturing steps we observed and practiced on our own enterprise application development projects [30][33]. We refer to these three conceptual steps as decision identification, making, and enforcement. As we will explain, today's practices support each of these steps only insufficiently. In our framework, reusable decision templates and semi-automatic decision model instantiation speed up the decision identification step. We aim to improve the quality of the decision making with decision dependency modeling, catalogs of decision drivers, and recommendations for decision making techniques. Finally, we propose decision injection into model transformations, code aspects, and configuration policies as an additional means of enforcing decisions in model-driven software development. A common meta model explicitly capturing reuse and collaboration aspects connects the three steps. Our reusable decision modeling framework is complementary to software engineering methodologies such as the Rational Unified Process (RUP) [19]; decision making can become a dedicated part of the work breakdown structure defined by the software engineering methodology of choice. The framework also is complementary to traditional component-and-connector modeling of software architecture design [3]; decisions explicitly refer to elements of design models such as logical components.

The remainder of this paper is structured in the following way: Section 2 introduces background and related work; Section 3 presents the requirements and the meta model for our conceptual framework for architectural decision modeling with reuse, and how the framework facilitates decision identification, making and enforcement. Section 4 applies our approach to the design of enterprise applications employing Service-Oriented Architecture (SOA) as their primary architectural style. Section 5 concludes with a summary and an outlook to future work.

2 Background and Related Work

Our work extends several recent contributions to software architecture research, which in turn are based on existing work in design decision rationale research. We also draw upon the rich architectural knowledge captured by the patterns community.

In [20], Kruchten et al. define an ontology that describes the attributes that should be captured for a decision, the types of decisions to be made, how decisions are made (i.e., their lifecycle), and decision dependencies. In their work, Kruchten et al. also focus on the visualization of the decisions. In [6], Falessi et al. present the decision, goal, and alternatives framework to capture design decisions. Their motivation is to

increase the maintainability of a software system by identifying why a certain approach has been chosen, and which design decisions have to be updated when the system is changed. In our work we build on both of these approaches, especially the ontology put forward by Kruchten and the use cases identified by Falessi, and apply them to enterprise application development. Unlike existing work, we investigate proactive decision identification to ease the reuse of architectural rationale. We are particularly concerned with collaboration and automation aspects.

Jansen and Bosch [15] view a software architecture as a composition of a set of design decisions. Their model for architectural design decisions focuses on the time dimension, defining a dedicated entity representing architectural modifications occurring over the software lifecycle. Other decision capturing templates exist in industry and academia, which can also be viewed as informally specified meta models [1][28]. None of these models is rich enough to support decision identification in requirements models, and there is no genuine support for decision reuse and collaboration. We could not find an alignment of these works with software engineering methods and patterns; platform-independent concerns are not separated from platform-specific ones. Our work enhances the existing modeling ideas in these directions.

Design decision research in the 1990s [21] focused on facilitating the decision making step; explicit identification and enforcement steps are not present. For instance, Questions, Options and Criteria (QOC) diagrams [22] raise a design question, which points to the available solution options; decision criteria are associated with the options. Selecting an option can lead to follow-on questions. Many active and passive Decision Support Systems (DSS) have been proposed. Most of the existing work focuses on management decision support; however, Svahnberg et al. suggest a quality-driven multi-criteria decision support method for software architecture selection [26]. This method allows multiple team members to score already identified architecture candidates based on weighted quality attributes. The scores lead to a suggestion and stimulate a consensus discussion. However, identification and reuse of required decisions, available alternatives and relevant quality criteria are out of scope. QOC diagrams and DSS complement our work and can be leveraged during our decision making step.

In the patterns community, several schools of thought and many pattern templates exist [5][9][11]. Requirements linkage typically is informal and appears in textual *intent* or *forces* sections. Many pattern languages remain on an abstract, conceptual level; others specialize on a single problem or technology domain such as *enterprise application architecture* [7] or *process-driven SOA* [29]. Patterns for process-driven SOA describe how to automate the management of long-running business processes such as loan approval processing or order management along supply chains (problem domain) with workflow engines and communication middleware (technology domain). The activity flow in such processes can be specified using Business Process Modeling (BPM) tools and implemented as a network of communicating Web services [34]. In general, the relationship between architectural patterns and reusable decision models is synergetic. In this paper, enterprise application development serves as the sample domain; hence, SOA patterns appear as conceptual architecture alternatives in the reusable architectural decision model we introduce in Section 4.

3 A Conceptual Framework for Decision Modeling with Reuse

To overcome the limitations of the existing decision capturing approaches, we structure the architectural decision making process into three conceptual steps, decision identification, making, and enforcement.[1] *Decision identification* scopes the architecture design work on a particular software development project. Requirements and earlier decisions trigger the identification of individual decisions. During *decision making*, architects select alternatives according to certain decision drivers, which either are context-specific requirements or general software quality attributes [3][14]. This step is the core of the three-step process; making sound technical decisions on software development projects is what practicing architects are primarily responsible for. *Decision enforcement* deals with sharing the results of the decision making with the stakeholders and the project team, and getting them accepted. Figure 1 illustrates:

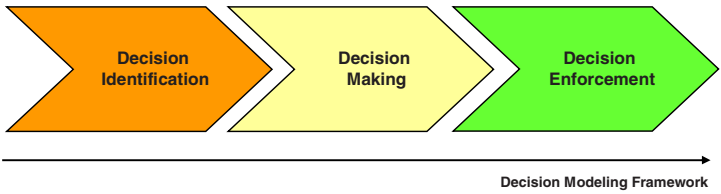

Fig. 1. Decision making steps

Each of the three steps has its own specific requirements, all of which have to be addressed by an underlying common meta model. In the remainder of this Section, we first investigate these requirements, then derive the required meta model elements from them and finally discuss how we support the identification, making, and enforcement steps.

3.1 Requirements

Having interviewed close to 100 practicing software architects, we identified the following design goals and use cases for our decision modeling framework.

Design goals. Supporting the decision identification, making, and enforcement steps requires extending existing practices for building up architectural knowledge, particularly if the decision making responsibilities are shared within and across teams. Therefore, providing *team collaboration support* is a mandatory design goal – architectural decision making is a team effort, and for budgetary and other reasons, software development projects today typically are carried out by geographically distributed teams. Furthermore, it should be possible to *harvest architectural decisions from completed projects*; a *small overhead* for capturing fresh decisions is desirable.

Use cases. In [6], thirteen general use cases for design decision rationale capturing are identified, covering a wide range of activities such as design problem detection,

[1] Finer grained models exist, for example in systems theory [10] and DSS research [26].

validation, documentation, coordination, and communication. With respect to our design goals, they lead to the following seven concrete primary use cases:

1. *Obtain architectural knowledge* from third parties, e.g., company-wide enterprise architecture groups or practitioner communities in consulting firms.
2. *Adopt and filter obtained decision knowledge* according to project specific needs: delete, update, and add architectural decisions and alternatives, and manage dependencies between decisions.
3. *Delegate decision making authorities* to subsystem architects and lead developers and support review activities with bidirectional feedback loops.
4. *Involve network of peers* in search of additional architectural expertise during decision making, requiring a common understanding of problem and solution space; hence, it is important to align terminology as much as possible.
5. *Enforce decision outcome* via pattern-based generation of work products, for example documentation and code snippets serving as architectural templates.
6. *Inject decisions* into design models, code, and deployment artifacts.
7. *Share gained architectural knowledge* with third parties such as the actors from use case 1, after having sanitized the project deliverables.

3.2 Meta Model Underpinning and Connecting the Framework Steps

To be able to support the use cases from Section 3.1 and automate parts of our three-step process, a common meta model is required. Figure 2 shows our proposal, which is inspired by previous research [1][15][20], the IBM e-business Reference Architecture Framework used in [28] and our own decision documentation practices [30][33]:

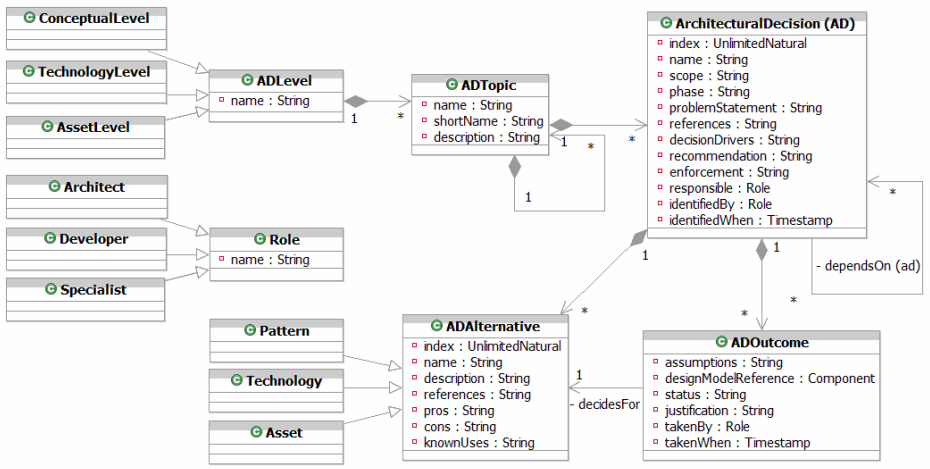

Fig. 2. Meta model in conceptual modeling framework for architectural decision reuse

There are three core domain entities, *Architectural Decision (AD)*, *ADAlternative*, and *ADOutcome*. In line with [15], we separate the outcome from the background information, in our case to facilitate reuse. AD and ADAlternative provide background

information only; attributes such as *problemStatement* characterize an AD on an introductory level, while *references* and *knownUses* point to further information.

The rationale behind this modeling choice is that the same AD might pertain to many elements in a design model, e.g., business processes and Web service operations. The design model element types are referenced via the *scope* attribute in the AD. ADOutcome instances then can be created dynamically, and refer to design model element instances via a *designModelReference*. To give an example, an order management process model might state that five business processes have to be implemented as a set of composed Web services [30]; while attributes such as problem statement, references, and *recommendation* are the same for all five processes, the *justification* might differ, depending on the individual *decision drivers*. Decision drivers include project-specific non-functional requirements (including environmental issues such as skill availability) and general software quality factors. The patterns community uses the term *forces* synonymously.

Closely related ADs are grouped into *ADTopic*s, which can form a hierarchy. Each ADTopic hierarchy is assigned to one of three *ADLevel*s of abstraction, *ConceptualLevel, TechnologyLevel,* or *AssetLevel*. This novel structure is motivated by our observation that when designing enterprise applications, the technical discussions often circle around detailed features of certain vendor products, or the pros and cons of specific technologies, whereas many highly important strategic decisions and generic concerns are underemphasized. These discussions are related, but should not be merged into one. We therefore go through two refinements steps. This is good practice, e.g., Fowler [8] and RUP with its elaboration points recommend such an approach for UML class diagrams used as design models. We adopted this recommendation for decision models and made the three abstraction levels explicit in our meta model.

Several attributes such as *responsible*, *takenBy* and *status* model decision ownership and lifecycle in response to the collaboration use cases from Section 3.1. The *phase* attribute provides a link to general-purpose methodologies such as RUP. These and all other model attributes can queried, e.g., when looking for all open decisions to be made in the inception phase of an enterprise application development project.

Decision dependencies are explicitly modeled as associations between ADs. At present, we use a single *dependsOn* dependency type, but are in the process of adopting the taxonomy from [20]. To give an example, for our order management business processes, a conceptual decision for a PROCESS AUTOMATION PARADIGM is required: Should the processes be made executable in a WORKFLOW ENGINE, or be realized in traditional PROGRAMMING LANGUAGE CODE? If a workflow engine is decided for, a related technology decision is to agree on an EXECUTABLE WORKFLOW LANGUAGE, e.g., BUSINESS PROCESS EXECUTION LANGUAGE (BPEL) [23]. Once BPEL has been decided upon, a BPEL ENGINE can be selected, e.g., ACTIVE BPEL, IBM WEBSPHERE PROCESS SERVER or ORACLE BPEL PROCESS MANAGER.[2]

3.3 Step 1: Decision Identification

Let us now investigate state of the art and the practice for the first step in our framework, decision identification. Next, we discuss how our decision identification support can increase productivity and improve quality.

[2] In this and all further examples, we set ADs and ADALTERNATIVES in THIS FONT.

State of the art. Pattern languages [7][11], domain-specific plugins for software engineering methods [16], technical papers and vendor documentation can be studied to identify required technical decisions. In theory, these sources of information provide deep coverage of all design concerns. However, the consumability of the vast amount of information is a key issue. Architectural decisions are often hidden behind various other material not targeting architects and therefore not being presented appropriately.

Project reality. During our decision modeling work with practicing architects, it became apparent that ad-hoc decision identification solely based on personal experience is the state of the practice, as opposed to diligent literature studies, or systematic reuse of knowledge already gained in a community. As a consequence, much time is spent in early project phases (requirements analysis, high level design) to identify the critical design issues, invent potential solutions, and agree upon decision criteria, particularly if the team lacks experience. This time would be better invested in studying the business problem to be solved, and in the actual decision making.

Our approach. As Figure 3 shows, we propose the initial decision model for a project team to be instantiated from project-specific requirements models and reusable decision templates. *Reference architectures* play a key role here, providing a common technical vocabulary and architectural patterns for a certain domain [3]. Architectural decisions cannot live in isolation; they have to be bound to design model elements, which can be found in the reference architecture. We refer to this binding step as *decision scoping*. In contrast to the *pull* model employed in practice today, we *push* the initial to-do list to the architecture team. We expect this reuse approach to increase productivity significantly, and to have a positive effect on quality. The decision templates serve as a completeness check list which can be seen as an early, informal review of the architectural work.

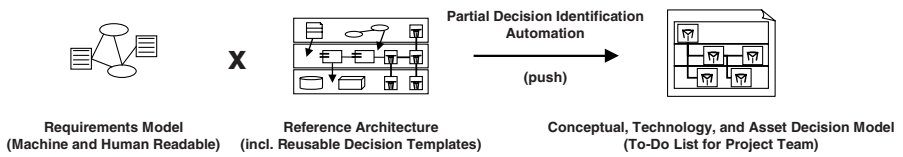

Fig. 3. Semi-automatic decision identification in requirements model and reference architecture

We do not aim to populate the entire design space; there will always be project-specific design issues worth capturing ad hoc. However, proactive decision identification works fine for many common design issues. For instance, in [34] we captured 26 architectural decisions dealing with WEB SERVICES as INTEGRATION TECHNOLOGY. These decisions cover interface design issues such as SELECTION OF INTERFACE DESCRIPTION LANGUAGE and MODELING STARTING POINT (BUSINESS REQUIREMENTS vs. EXISTING IT ASSET) These decisions were reused successfully on several Web services projects conducted by others [12].

3.4 Step 2: Decision Making

The actual decision making is the second step of our three-step framework.

State of the art. Architecture Tradeoff Analysis Method (ATAM) [3], Attribute-Driven Design (ADD) and Decision Support Systems (DSS), as well as many semi-formal techniques such as Strengths, Weaknesses, Opportunities, Threats (SWOT) tables can be used to support decision making. ATAM was originally positioned as an evaluation and review instrument, but can also be used during earlier decision making stages. Without customization, generic techniques such as ADD do not provide reusable, domain-specific advice. Many decision making techniques require information not yet available during the early elaboration stages or use the strategy to address one Non-Functional Requirement (NFR) at a time and hence do not take side effects caused by decision dependencies into account. As a consequence, not all techniques are equally suited for all decision types.

Project reality. Architectural decision making is often perceived as an art rather than part of an engineering process. Decisions makers often are biased; phrases like "this has always worked for me" or "this is the industry trend" justify decisions instead of sound technical judgment backed by tradeoff analysis activities or technical evaluations. Frequently, a single driver is overemphasized. For instance, we have seen architects use a simplistic "brain/heart/guts" model. In summary, personal experience, preferences, and intuition often are the main decision drivers; external forces such as vendor interests or strategic decisions motivated by potential future needs and synergies have a large, not always beneficial, impact on the decision making. Consequently, the technically best solution is not always selected. Such ill-motivated and -fated decision making often is a root cause for project failure as the quality of the produced software architecture degrades.

Our approach. Aiming to objectify the decision making, we integrate a collection of proven decision support techniques into our framework, which accompany and use the decision models created during the identification step. We also provide a list of decision drivers per decision, e.g., highlighting specific NFRs and software quality factors, but also non-technical factors such as political issues, license costs, and available skills.

Depending on the type of decision to be made, we select from a continuum of support techniques, e.g., simple recommendations, semi-structured SWOT tables, ADD [3], QOC diagrams [22], hands-on evaluations and formal alternative scoring algorithms [26]. A benefit of this approach is that it provides the decision makers with a technique well suited for a particular decision, as well as tangible advice that is aligned with requirements and background information (e.g., vendor best practices). Figure 4 illustrates.

In our opinion, it is neither feasible nor desirable to fully automate the decision making. The importance of tradeoffs in specific contexts and design drivers naturally makes full automation impossible; heuristic solutions are required. Matching of the requirements contexts and decision drivers is important when reusing architectural knowledge. In many circumstances, it is imperative to deviate from generic

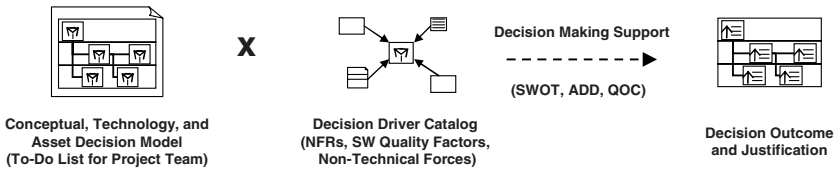

Fig. 4. Decision models, decision drivers and techniques for decision making

recommendations. Hence, the decision making support in our approach empowers the architects to make informed decisions based on collective insight.

To give an example, using DEEPLY NESTED XML SCHEMA TYPES as MESSAGE PARAMETER GRANULARITY was considered an anti-pattern in early Web services literature. Confronted with a rich core banking domain model, we still decided for this alternative in one of our projects [33]. We did so after having conducted a proof-of-technology to mitigate interoperability and performance concerns, which we had identified as key decision drivers. This decision justification became a reusable architectural recommendation at a later stage, due to the positive experience gained.

3.5 Step 3: Decision Enforcement

State of the art. Traditional software engineering processes like RUP [19] address decision enforcement through stepwise design refinement down to code. The agile community [4] emphasizes the importance of face-to-face communication. Maturity models such as the Capability Maturity Model Integration (CMMI) [25] and domain-specific governance models [13] also can be used to ensure that ADOutcomes find their way into running code. At build and deployment time, concepts such as code aspects and configuration policies can be used to express architectural intent explicitly. However, complexity and maturity concerns have limited a broad adoption of these two concepts so far.

Project reality. Coaching, architectural templates, and code reviews are the dominating decision enforcement approaches today. All of them are perfectly valid. However, applying these approaches takes time and depends on the coding and leadership skills of the decision makers. Personal architectural knowledge that remains tacit often is lost during the maintenance phase of the application lifecycle, e.g., when the team setup changes. Codifying architectural knowledge in design models is an additional option when following Model-Driven Architecture (MDA) principles. However, a key limitation of standard MDA is that model transformations often are not configurable and therefore hard to adjust to project-specific architectural decisions [32]. For example, many BPM-to-BPEL tools allow the user to make simple decisions, e.g., regarding activity naming, but use fixed values for key aspects, e.g., system transaction management settings. Consequently, development resources have to be invested for changing the default values to the settings required in the particular requirements context. Such disconnects and reconciliation problems between architecture and development tools and artifacts can be observed frequently.

Our approach. The existing practices work fine for many decisions, particularly those pertaining to micro design. As an additional option in our framework, machine-readable decision models can be interpreted by model transformations and code

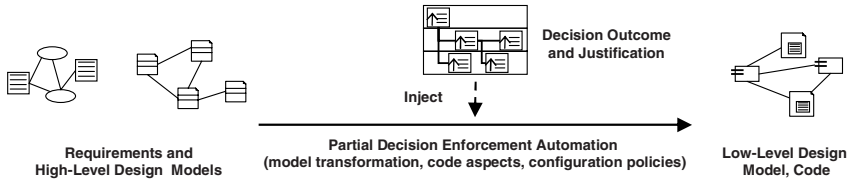

Fig. 5. Decision enforcement via injection into model transformations and code generation

generators. Figure 5 illustrates this *decision injection* concept, which can help to reduce unnecessary development efforts and ensure architectural consistency:

We have built a demonstrator for such an approach that uses Eclipse JET templates to codify key architectural decisions dealing with non-functional concerns regarding the implementation of executable business processes. For example, the demonstrator injects ADOutcome for TRANSACTIONAL POLICIES such as REQUIRESOWN and PARTICIPATES into the BPEL code generated by a BPM tool used to capture business requirements. In this example, the decision drivers are the logical business transaction boundaries, the physical resource protection needs, and the capabilities of the involved legacy systems. The BPM tool user, typically a domain expert (business analyst), can and should not be responsible for this architectural decision.

4 Application of Conceptual Framework to SOA Design

In this section, we describe how we applied the conceptual framework from Section 3 to enterprise application development and SOA design incrementally. First, we organized the decision points encountered on our own SOA projects [30][33] according to the meta model from Section 3.2. As a second step, we factored in selected architectural knowledge from projects technically led by peers, leveraging an IBM-wide SOA and Web services practitioner community with 3500 members. To verify that the concepts are not limited to SOA as the primary architectural style, we cooperated with architects specializing on information management, who documented their know-how about information integration and data-centric architectures using our concepts. The result is a reusable SOA decision model we refer to as *SOA Design Space*.

4.1 Requirements Model and Reference Architecture for SOA Design Space

In Section 3.3, we explained that we require a machine-readable requirements model to be able to partially automate the decision identification step. When constructing SOAs, *analysis-level business process models*, optionally annotated with NFRs, are well suited for this purpose [17]. Object-oriented analysis artifacts such as use case models also work well. Our minimum requirement for such models is that they have to list the processes and activities to be realized as software services; the decision identification support can then create *realization decisions* for these high-level functional building blocks.

In the SOA case, we use the abstract *SOA reference model* from [2] as our reference architecture. It provides a conceptual, semi-strict layering scheme defining nine layers: consumer, process, service, component, resource, integration, Quality of

Service (QoS), information, and governance. It is possible to use other reference architectures, as long as these provide a layering scheme and allow associating a decision with the design model elements it pertains to. The selection of the concrete REFERENCE ARCHITECTURE is an executive-level architectural decision in its own right; making it is part of the project-specific adoption of the SOA Design Space.

If an analysis model has already been transformed into a high-level design model, e.g., with support from BPM and SOA tools, we can further improve the decision identification step because the business-level activities in the process model have already been refined into high-level design artifacts such as candidate services. Fewer decisions remain. An example for such a transformation is DATA CONTAINER ASSIGNMENT, producing typed service operations as output. Furthermore, unnecessary design points can be deleted. For example, if cycles have been removed from the business process automatically, DEALING WITH CYCLIC PROCESS MODELS is no longer relevant [17].

4.2 Organizing Principles in the SOA Design Space

To decompose the rather complex SOA design domain, we applied several proven structuring principles such as *separation of concerns* and *logical layering*. Figure 6 outlines the overall structure, resembling the ADLevel hierarchy from Figure 2:

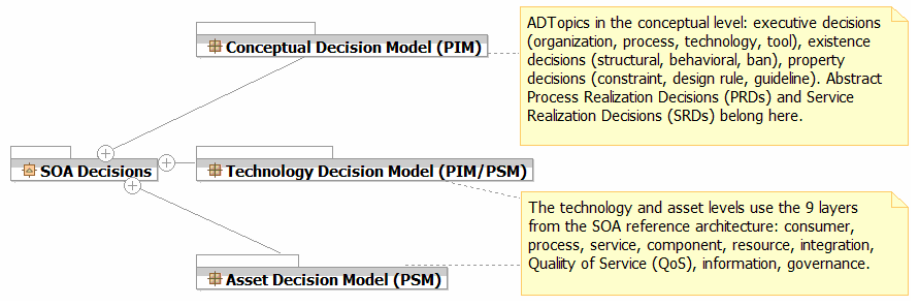

Fig. 6. UML packages for SOA Design Space and assignment to MDA levels

ADTopics are used as a fine-grained grouping mechanism on each MDA level. We aim for *high cohesion* within and *low coupling* between ADTopics. In the Conceptual Decision Model, we use the ontology from [20]. The reference architecture from [2] organizes the ADTopics. Table 1 lists selected conceptual ADTopic nodes with examples, comparing their identification, making, and enforcement characteristics:

Employing a *consistent naming style* for ADTopics, ADs, and ADAlternatives is another principle to make models comparable; all elements created according to the meta model from Section 3.2 have a unique identifier and a self-explaining short name. Our terminology takes inspiration from service modeling [2], enterprise architecture [7] and SOA patterns [29] literature. By convention, alternatives are ordered from common and recommended to exceptional; if present, fallback alternatives such as CUSTOM CODING and OTHER appear last.

Table 1. Decision types in Conceptual Decision Model of SOA Design Space

Decision type (ADTopic) with examples	Identification (role, phase)	Decision Making Support (non-exclusive list)	Enforcement (now, future)
Executive Decisions, e.g., PLATFORM SELECTION, ARCHITECTURAL STYLE, GOVERNANCE	Enterprise architect, before project starts	SWOT analysis and other consulting techniques (high number of alternatives, incomplete data)	Now and future: Governance processes, limited tool support (personal productivity software)
Enterprise Architecture Decisions (EADs), e.g., existence decisions: TRANSACTION MANAGEMENT, SESSION MANAGEMENT, LAYERING, PERSISTENCE STRATEGY [7]	Lead architects and senior developers, during early project phases (solution outline, macro design)	Literature research (e.g., patterns books, online forums) and "if-then" best practices rules (often several valid choices, decision drivers semi-concrete)	Now and future: Architectural templates, coaching Future: pattern toolkits, configurable model transformations
Process Realization Decisions (PRDs), e.g., property decisions: MACRO VS. MICRO FLOW, INSTANCE CORRELATION, SYSTEM TRANSACTION BOUNDARIES, COMPENSATION [32]	Technical architects, lead developers, platform and technology specialists, during macro and micro design	Domain analysis and design (challenging NFRs and many other decision drivers), to be supported by QOC diagrams etc. (choices can be justified by concrete decision drivers)	Now: Manual coding, hard wired in MDA model transformations and code generators Future: Decision injection into code, aspects, policies
Service Realization Decisions (SRDs), e.g., MESSAGE EXCHANGE PATTERN, SERVICE GRANULARITY [32]	Same as PRDs, but different skill set	Same as PRDs, but often less alternatives because decisions on higher levels constrain choices	Same as PRDs

The SOA Design Space implements the abstract decision scoping concept from Section 3.2, using the process and service abstractions from the selected SOA reference architecture. PRDs have to be taken per process to be realized in software, SRDs once per process activity to be implemented as a software service.

Via *decision tagging*, ADs can be annotated with keywords to express cross-cutting concerns, which then become additional dimensions in our SOA Design Space. For instance, we tagged all decisions dealing with transactionality across ADLevels and ADTopics so that they can be searched for.

There are many dependencies within and between the levels. To resume the example from Section 3.2, PROCESS AUTOMATION PARADIGM and deciding between abstract MESSAGE EXCHANGE PATTERNS such as REQUEST-REPLY and ONE WAY are architectural decisions in the Conceptual Decision Model. In the Technology Decision Model, concerns then are BPEL PROCESS DESIGN and SOAP MESSAGING VS. REPRESENTATIONAL STATE TRANSFER (REST) as MESSAGE EXCHANGE FORMAT; when integrating distributed components, the selection of a TRANSPORT PROTOCOL, e.g., HTTP or MESSAGING, is another technology decision. Vendor-specific issues appear in the Asset Decision Model. WEB SERVICE STACK SELECTION and deploy-

ment issues such as selection of an open source or commercial SOAP ENGINE (e.g., APACHE AXIS) and engine-specific BPEL configuration decisions such as LONG OR SHORT PROCESS LIFETIME and ACTIVITY TRANSACTIONALITY are examples for such decisions [31]. The dependencies between the levels are modeled explicitly.

4.3 Example: Ws-01, Service Provider Type

Figure 7 illustrates a single AD, the selection of the SERVICE PROVIDER TYPE. It is a screenshot of AD_{kwik}, a Web 2.0 collaboration front end implementing the concepts presented in this paper. We describe the user interface and knowledge engineering concepts of AD_{kwik} in detail in [24].

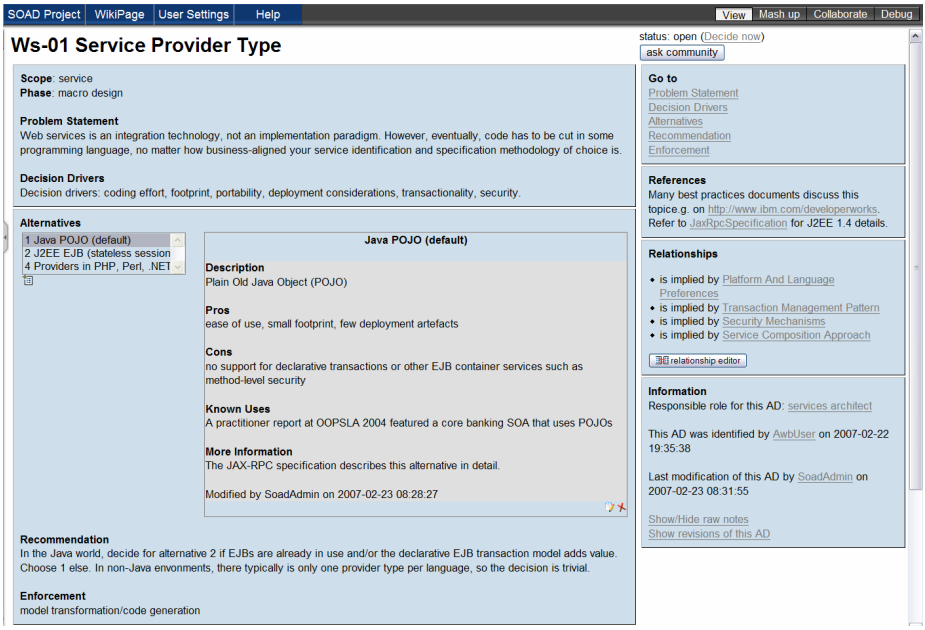

Fig. 7. Web services decision example: Ws-01, selection of SERVICE PROVIDER TYPE

The SERVICE PROVIDER TYPE decision is a SRD according to Table 1. On SOA projects, this decision has to be made for each service to be implemented, it can be identified in the analysis-level BPM model serving as input to the decision making process; therefore, this decision has a "service" scope (the scope attribute is defined in our meta model, see Figure 2). The phase attribute links the decision to a methodology. In this case, "macro design", a term from the method used by IBM Global Services, suggests that this decision should be taken during the early, overall architecture design. There is a problem statement motivating why this decision is needed. In this example, it is one paragraph paraphrasing the motivation for this decision found in the literature; in other cases, a simple question like "How to correlate incoming user requests to server-side session objects?" is more appropriate.

For this decision, the coding effort, the memory footprint, and several other general quality attributes are listed as particularly important decision drivers. The available alternatives are listed as well, along with their pros, cons and known uses. In the example, JAVA PLAIN OLD JAVA OBJECT (POJO), J2EE ENTERPRISE JAVA BEAN (EJB), and PROVIDERS IN PHP, PERL, .NET have been identified. The references field points to recommended reading, in this case two online resources. Dependencies to and from other decisions are modeled explicitly and shown as relationships. For example, the executive-level PLATFORM AND LANGUAGE PREFERENCES decision clearly has an impact: the non-Java alternatives are no longer relevant if using Java is imperative. As there are several WSDL-TO-JAVA CODE GENERATION WIZARDS, this decision then can be enforced via code generation, assuming that the selected wizard supports both POJO and EJB generation.

4.4 Initial Evaluation and Expected Benefits of SOA Design Space

As stated previously, the initial content of our SOA Design Space originates from several successful large-scale SOA development projects conducted since 2001. In the meantime, we have refactored the content and the meta model several times, which led to the fine-grained ADTopic structure outlined in Section 4.2. At present, the SOA Design Space consists of 160 reusable decision nodes.

We have already applied our SOA Design Space in the use cases specified in Section 3.1, as well as for education, coaching, and architecture review purposes. From the experience gained during this initial evaluation, we estimate that on average one third of the early project phases such as RUP inception is spent on education and identification of decision points. Some of that will always be required to give new team members an opportunity to familiarize themselves with the project context, for instance the business problem to be solved and the project logistics (tools, build environment, etc.). Still, the feedback from early SOA Design Space users suggests that much of this time can be saved with better tooling and pre-configured decision models supporting decision identification in requirements models and reference architectures.

In one case, the effort for the creation of a SOA principles deliverable decreased from eight to five person days because thirteen out of fifteen required decisions were present in the SOA Design space and could be reused. For instance, the architect on that project reused the decision node from Figure 7. The decision drivers listed in Section 4.3, particularly transactionality needs and ease of deployment, matched with the project requirements, so that our recommendation to use EJBs if leveraging the declarative EJB transaction model is adequate, and to use POJOs otherwise, was directly applicable. The architect also reported that he found several decisions in the SOA Design Space that he had not identified yet, but which turned out to be required: for instance, the decision for a SERVICE CATEGORIZATION SCHEME to distinguish technical utility services and logic-centric business services, which is described in [18] and [30], became a key element of his SOA design.

A rigorous decision making process is often seen as a prerequisite to achieve higher maturity levels, e.g., in CMMI [25]. Decision dependency modeling makes design errors visible and allows backtracking. A positive impact on software quality can be expected, for example when combinations that do not work are detected or

disabled before the mistake is even made. These positive effects are hard to quantify; however, we have observed them on projects already.

Our decision enforcement approach leads to less manual reconfiguration and coding needs and simplifies the model-code reconciliation, faithful to the original vision of MDA. A positive impact on team communication and climate can also be expected. Decision capturing becomes a shared responsibility; decisions that are openly created, discussed, and justified often are easier to accept than dictated ones.

5 Conclusions and Outlook

In this paper, we presented a proactive approach to modeling and reusing architectural knowledge for enterprise application development. As discussed in Section 2, our approach extends existing proposals for retrospective architectural decision capturing. It facilitates reuse of design rationale and team collaboration, two issues particularly relevant in enterprise application development. In Section 3, we defined a conceptual framework facilitating collaborative decision making supported by an extended meta model. In this framework, three steps improve decision reuse and sharing of rationale:

- Semi-automatic decision identification, speeding up early project activities. In this step, we combine requirements models with reference architectures containing reusable decision templates to create an initial to-do list.
- More informed decision making via reusable collections of decision drivers, good practices recommendations and other supporting techniques. In this step, our framework promises to improve decision making rigor and quality.
- Improved decision enforcement in MDA via decision injection into parameterized model transformations and code generation, reducing development efforts and simplifying communication, governance, and maintenance.

As demonstrated in Section 4, our approach already has proven to be practical for BPM requirement models and SOA as architectural style; we compiled a SOA Design Space with 160 reusable decision nodes. We could observe initial effort savings and quality improvements on an early adoption project. Tool support is available.

The presented approach is generally applicable if several applications are built in the same or a similar context and if full decision automation is an illusion. We require the requirements model to be reasonably structured and at minimum one reference architecture for the selected architectural style to exist. Enterprise application development and SOA meet these applicability criteria.

The complexity of the solution space and keeping the content up-to-date, consistent, and easy to locate are key challenges for a broader adoption of the presented approach. In response to these challenges, we plan to investigate the integration of architectural design and decision models even further, to involve a broader practitioner community in future content engineering, and to leverage additional results from other fields, e.g., knowledge management and architectural patterns.

We envision several advanced usage scenarios for the SOA Design Space. Project managers can use it for planning and health checking purposes. Work breakdown structures and effort estimation reports can be created from the decision model, as open decisions correspond to required activities. If there are many, frequent changes,

or many questions are still unresolved in late project phases, the project is likely to be troubled. Moreover, product-specific decision outcome can serve as input to software configuration planning. Product selection and operational modeling decisions define which software licenses are required, and on which hardware nodes the required software has to be installed. The SOA Design Space can also serve as an enterprise architecture communication vehicle; enterprise architects can maintain a company-specific instance of the SOA Design Space, consisting of a subset of decisions and alternatives to give freedom of choice to individual project teams without sacrificing overall architectural integrity. Finally, we plan to use the SOA Design Space as a prescriptive micro method for SOA construction, complementing service modeling methods.

Future research work includes exploring several advanced concepts, for example more expressive dependency modeling. Decision space pruning can rule out alternatives based on the outcome of other decisions. We also plan to investigate whether reusable architectural decision models can help improving the documentation of software products, for example packages and middleware with many variation points.

Acknowledgments. We would like to thank Davide Falessi, Jonas Grundler, Gregor Hohpe, Dirk Huppert, David Janson, Ed Kahan, Jochen Klein, Jana Koehler, Oliver Kopp, Petra Kopp, Philippe Kruchten, Einar Landre, Ralp Mietzner, Sven Milinksi, Frank Müller, Mike Papazoglou, Stefan Pappe, Cesare Pautasso, Willem-Jan van den Heuvel, Harald Wesenberg, and Uwe Zdun for their input, provided through many discussions and/or reviews of earlier versions of this paper.

References

[1] Abrams, S., Bloom, B., Keyser, P., Kimelman, D., Nelson, E., Neuberger, W., Roth, T., Simmonds, I., Tang, S., Vlissides, J.: Architectural Thinking and Modeling with the Architects' Workbench. IBM Systems Journal 45 (2006)
[2] Arsanjani, A.: Service-oriented modeling and architecture, IBM developerWorks (2004), http://www.ibm.com/developerworks/webservices/library/ws-soa-design1
[3] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison Wesley, Reading (2003)
[4] Beck, K.: Extreme Programming Explained. Addison Wesley, Reading (2000)
[5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture – a System of Patterns. Wiley, Chichester (1996)
[6] Falessi, D., Becker, M., Cantone, G.: Design Decision Rationale: Experiences and Steps Towards a more Systematic Approach. In: Workshop on Sharing and Reusing Architectural Knowledge, ACM SIGSOFT Software Engineering. Notes 31, 5 (2006)
[7] Fowler, M.: Patterns of Enterprise Application Architecture. Addison Wesley, Reading (2003)
[8] Fowler, M.: UML Distilled. Addison Wesley, Reading (2000)
[9] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
[10] Hitchins, D.: Advanced Systems Thinking, Engineering, and Management. Artech House Publishers (2003)
[11] Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison Wesley, Reading (2004)

[12] IBM Corporation: SOA and Web Services Best Practices, Academy of Technology Report (2004)
[13] IBM Corporation, SOA Governance and Management Method, http://www.ibm.com/software/solutions/soa/gov/method
[14] International Standards Organization (ISO), ISO/IEC 9126-1:2001, Software Quality Attributes, Software engineering – Product quality, Part 1: Quality model (2001)
[15] Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (Wicsa 2005), IEEE Computer Society Press, Los Alamitos (2005)
[16] Johnston, S.: RUP Plug-In for SOA V1.0, IBM developerWorks (2005), http://www.ibm.com/developerworks/rational/library/05/510_soaplug
[17] Koehler, J., Hauser, R., Küster, J., Ryndina, K., Vanhatalo, J., Wahler, M.: The Role of Visual Modeling and Model Transformations in Business-driven Development. In: Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques, Elsevier, Amsterdam (2006)
[18] Krafzig, D., Banke, K., Slama, D.: Enterprise SOA. Prentice-Hall, Upper Saddle River (2005)
[19] Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley, Reading (2003)
[20] Kruchten, P., Lago, P., van Vliet, H.: Building up and reasoning about architectural knowledge. In: Hofmeister, C., Crnkovic, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, Springer, Heidelberg (2006)
[21] Lee, J., Lai, K.: What's in Design Rationale?. Human-Computer Interaction 6 (3 & 4) (1991)
[22] MacLean, A., Young, R., Bellotti, V., Moran, T.: Questions, Options, and Criteria: Elements of Design Space Analysis, Human-Computer Interaction 6 (3 & 4) (1991)
[23] OASIS. Web Services Business Process Execution Language (WSBPEL), Version 1.1 (2003), http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
[24] Schuster, N., Zimmermann, O., Pautasso, C.: ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering. In: Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE 2007), KSI (2007)
[25] Software Engineering Institute, Capability Maturity Model® Integration (CMMI), http://www.sei.cmu.edu/cmmi
[26] Svahnberg, M., Wohlin, C., Lundberg, L., Mattsson, M.: A Quality-Driven Decision Support Method for Identifying Software Architecture Candidates. International Journal of Software Engineering and Knowledge Management 13(5) (2003)
[27] Tang, A., Babar, M.A., Gorton, I., Han, J.: A Survey of the Use and Documentation of Architecture Design Rationale. In: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (Wicsa 2005), IEEE Computer Society Press, Los Alamitos (2005)
[28] Tyree, J., Akerman, A.: Architecture Decisions: Demystifying Architecture. IEEE Software 22 (2005)
[29] Zdun, U., Dustdar, S.: Model-Driven and Pattern-Based Integration of Process-Driven SOA Models, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, http://drops.dagstuhl.de/opus/volltexte/2006/820
[30] Zimmermann, O., Doubrovski, V., Grundler, J., Hogg, K.: Service-Oriented Architecture and Business Process Choreography in an Order Management Scenario. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), ACM Press, New York (2005)

[31] Zimmermann, O., Grundler, J., Tai, S., Leymann, F.: Architectural Decisions and Patterns for Transactional Workflows in SOA. In: Krämer, B., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 81–93 (2007)
[32] Zimmermann, O., Koehler, J., Leymann, F.: The Role of Architectural Decisions in Model-Driven Service-Oriented Architecture Construction. In: Proceedings of the OOPSLA 2006 Workshop on Best Practices and Methodologies in Service-Oriented Architectures, Unipub (2006)
[33] Zimmermann, O., Milinski, M., Craes, M., Oellermann, F.: Second Generation Web Services-Oriented Architecture in Production in the Finance Industry. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), ACM Press, New York (2004)
[34] Zimmermann, O., Tomlinson, M., Peuser, S.: Perspectives on Web Services. Springer, Heidelberg (2003)