Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method

Olaf Zimmermann
IBM Research GmbH
Zurich Research Laboratory
Rüschlikon, Switzerland
olz@zurich.ibm.com

Uwe Zdun
Information Systems Institute
Vienna University of Technology
Vienna, Austria
zdun@infosys.tuwien.ac.at

Thomas Gschwind
IBM Research GmbH
Zurich Research Laboratory
Rüschlikon, Switzerland
thg@zurich.ibm.com

Frank Leymann
Institute of Architecture
of Application Systems
University of Stuttgart, Germany
leymann@iaas.uni-stuttgart.de

Abstract

When constructing software systems, software architects must identify and evaluate many competing design options and document the rationale behind any selections made. Two supporting concepts are pattern languages and architectural decision models. Unfortunately, both concepts only provide partial support: Extensive upfront education is needed for practitioners to be in command of the full pattern literature relevant in their field; retrospective architectural decision modeling is viewed as a painful extra responsibility without immediate gains. In this paper, we combine pattern languages and reusable architectural decision models into a design method that is both comprehensive and comprehensible. Our design method identifies the required decisions in requirements models systematically, gives domain-specific pattern selection advice, and provides traceability from platform-independent patterns to platform-specific decisions. We validate our approach by applying it to enterprise applications as an exemplary application genre and a SOA case study from the finance industry.

1 Introduction

When constructing software-intense systems such as enterprise applications, software architects encounter numerous design decision points. Both the "big picture" and many technical details influence their decision making. Numerous competing requirements must be balanced, tradeoffs occur. The high number of competing design options and tacit dependencies between them are two additional sources of complexity. Furthermore, a continuous stream of conceptual, technology, and product innovations must be evaluated. A lot of time is spent on basic orientation tasks repeatedly, decisions must be made under high time pressure, and the rationale is often lost. Consequently, constructing opti-

mal or "good enough" systems is a challenge. Software engineering research has come up with many supporting concepts, for instance *pattern languages* [5, 9, 13] and, more recently, *architectural decision models* [14, 17, 21].

Unfortunately, neither pattern languages nor architectural decision models solve all design and documentation problems in practical projects. By definition, patterns document reusable design knowledge, and hence they do not serve as a complete documentation of an individual system. Most practitioners only know a few patterns well, such as the Gang of Four design patterns [9], although the pattern community has documented patterns for many other domains, such as enterprise applications and Service-Oriented Architecture (SOA). Hence, extensive upfront education is required to use a pattern language well.

Retrospective architectural decision modeling is often seen as painful additional responsibility without many gains. It is still a research issue how to capture architectural decisions without introducing efforts that outweigh the benefits and make the gathered knowledge reusable [21]. Techniques, text templates and tool support have been proposed, but failed to become broadly adopted in practice so far.

This paper combines pattern languages and architectural decision models to their mutual benefit: Section 2 introduces the two concepts, and Section 3 presents a case study. In Section 4, an analysis of usage scenarios and characteristics of patterns and decisions leads us to a proposal how to combine them into a comprehensive and comprehensible design method. This method supports the systematic identification of the required architectural decisions in requirements models, allows giving domain-specific pattern selection advice, and provides traceability from platform-independent patterns to platform-specific decisions. Section 5 validates the method in the case study; the remaining sections discuss our approach, related work, and conclusions.

IEEE computer society

2 Review of Existing Work

Patterns and Pattern Languages. The pattern movement is a software engineering success story. In 1995, the Gang of Four published their seminal Design Patterns book [9]. Many different types of patterns have been published since then, for example Patterns of Software Architecture (POSA) [5], domain analysis patterns, and even patterns for non-IT topics. Examples for recent contributions are Patterns of Enterprise Application Architecture (PoEAA) [8], messaging [13], remoting [22], and SOA [12, 24].

A *pattern* is a proven *solution* to a *problem* in a *context*, resolving a set of *forces*. In more detail, the context refers to a recurring set of situations in which the pattern applies. The problem refers to a set of goals and constraints that typically occur in this context and influence the pattern's solution, called the forces of the pattern. To systematically explain how to apply a number of patterns in combination, many pattern authors document patterns as part of larger *pattern languages*, containing rich pattern *relationships* and extensive *examples* and *known uses* sections.

Patterns in a pattern language are applied in an incremental refinement process. The decision making in this process is based on the pattern's forces. In the architectural realm, these forces include Nonfunctional Requirements (NFRs) and software quality attributes [2]. Usually, not all forces can be resolved optimally; a compromise has to be found. The pattern describes how the forces are balanced in the proposed solution, and why they have been balanced in the proposed way. In addition, the advantages and disadvantages of such a solution are described as *consequences*.

Applying patterns during software design requires a broad view on how to select from a large body of patterns possibly eligible for a particular domain. Patterns do not focus on a single, domain-specific solution in a particular business context, but on generic design knowledge. For instance, the Invoker pattern [22] describes how a middleware invokes remote objects in general. The pattern applies to all kinds of middleware, but does not explain the specifics of Invokers in a particular application context such as a specific SOA middleware implementation.

Architectural Decision Capturing and Modeling. *Architectural decision capturing* [14, 17, 21] is an approach to software documentation that focuses on the expert knowledge motivating certain designs rather than the resulting designs themselves. Decisions capture selected design options with their strengths and weaknesses, as well as justifications for the selections. If a model-driven approach to architectural decision capturing is followed, we refer to the involved models as *Architectural Decision Models (ADMs)*.

In our earlier work, we refactored the decision template used in [21] and extended it into a meta model supporting decision maker collaboration and knowledge reuse [26]. We

defined architectural decisions as "conscious design decisions concerning a software system as a whole, or one or more of its core components. These decisions determine the non-functional characteristics and quality factors of the system". The main elements of our extended meta-model are: *Architectural Decision (AD)*, *AD alternative*, *AD outcome*, *AD topic*, and *AD level*. Architectural Decision (AD) is the core entity describing a single, concrete design issue with the following attributes: Decision *name* and unique *identifier*; *problem statement*; *scope*, linking the AD to design model element types; *decision drivers*, listing types of NFRs and quality attributes driving the design; *references* to literature such as overviews, tutorials, and other decision support material; *dependency relationships*; *lifecycle information* such as decision owner and modification log.

AD alternatives enumerate design options for ADs along with their *pros*, *cons*, and *known uses*. One or more AD outcome instances record the selection of an AD alternative for a particular AD and the *justification* for the decision. AD topics group related ADs; AD topic hierarchies can be formed. We separated platform-independent from platform-specific concerns and introduced four AD levels: an *executive*, a *conceptual*, a *technology*, and a *vendor asset* level. All AD topics reside on one of these AD levels [26].

Architectural decision capturing and modeling entail some potential problems. If positioned as retrospective documentation tasks, they do not have immediate benefits for the original project team. A major problem of such an approach is that it is not possible to reuse architectural knowledge gained on previous projects or earlier project stages. As a consequence, decisions are often captured only in a rudimentary fashion, or not at all. Many decisions are made ad hoc, often biased and based on personal preferences and unqualified recommendations ("gut feel" and "best practices"). In such situations, sound architectural choices with respect to project goals and NFRs can hardly be made. Acceptance and quality problems result, as well as disconnects between architecture and code.

3 Case Study: Application Integration

Enterprise application development is an application genre that is both common and complex. Therefore, we use a case study from this genre to illustrate the requirements for and characteristics of our design method.

The case study [27] reports about the service-oriented integration of enterprise application components in the finance industry, connecting 237 independent retail banks with a shared core banking backend provider. The value proposition for introducing SOA concepts was to resolve technology mismatches between the teller application front ends running in the banks (e.g., Java and scripting languages, Microsoft .NET on Windows) and the shared core

banking backend (COBOL, IBM z/OS mainframe systems hosting a CICS transaction monitor and a DB2 database). This heterogeneity motivated project goals such as the need to improve the developer experience on a number of frontend platforms while keeping the integration platform simple and maintainable. These goals justified the introduction of a single middleware following the BROKER pattern [5].

To design the software architecture in this situation, many decisions were required. For instance, pages 111 to 116 in [5] suggest six steps to implement the BROKER pattern: "(1) Define an object model. (2) Decide which type of component interoperability the system should offer, binary or Interface Description Language (IDL). (3) Specify the APIs the broker component provides for collaborating with clients and servers. (4) Use proxy objects to hide implementation details from clients and servers. (5) Design the broker component (6) Develop IDL compilers." Step (5) has nine sub steps: "(5.1) On-the-wire protocol, (5.2) Local broker, (5.3) Direct communication variant, (5.4) (Un)marshalling, (5.5) Message buffers, (5.6) Directory service, (5.7) Name service, (5.8) Dynamic method invocation, (5.9) The case in which something fails [...]". Several of these steps are trivial if a commercial broker product or a standardized, openly specified technology implementing the BROKER pattern is chosen; others lead to substantial design efforts:

- Regarding interoperability (2) and the on-the-wire protocol (5.1), a conceptual decision for an integration technology such as Web services [28] was required.

- Decision points (5.2) to (5.8) were addressed by the middleware technology and products. A remaining technical decision was XML messaging with SOAP/HTTP vs. REpresentational State Transfer (REST) [7], both proposing certain message exchange formats, protocol primitives, and addressing schemes.

- As Web services technology had been chosen, the IDL compiler design issue (6) was resolved trivially, as a standardized Web Services Description Language (WSDL) and a WSDL-to-Java generator were available. Many other solution elements also were mandated by the Web services technology, e.g., the Java XML API for RPC (JAX-RPC) defined server-side FACTORIES [9] and INVOKERS [22] as well as CLIENT PROXIES [22] for design issues (3) and (4).

- However, as a consequence of deciding for Web services, other patterns became eligible. For example, for the object model definition (1), DATA TRANSFER OBJECT [1] and flat strings had to be considered; remote objects were not an option.

- Detailed design work was required before any selected pattern could be implemented, including a pat-

tern adoption to the project-specific context and vendor asset selections (commercial products, open source frameworks). The design of error handling procedures (5.9) was an example of such detailed design work.

Many concrete NFRs in areas of architectural concern such as performance, scalability and interoperability steered the design. Furthermore, legacy constraints had to be taken into consideration, e.g., existing network capacity and back-end interfaces already implementing certain patterns for user and session management. These issues are not covered by the BROKER pattern, but elsewhere, e.g., by Alur et al. [1]. More recent SOA patterns literature such as Keen et al. [15] presents solutions to some of these issues. However, even such SOA-specific guidance does not solve the general problem of having to structure the domain-specific architectural design work further than pattern languages do.

4 Pattern- and Decision-Centric Design

In the case study, both the business requirements and the existing system landscape steered the decision making; there were many dependencies between the related ADs. Many architectural and design patterns existed in a large number of variants. Even with information about pattern selection forces and quality attribute scoring [2] it remained hard to decide for a particular variant of a pattern. Some ADs were not related to pattern selection at all, e.g., product selections and team organization issues. These observations (also made in other enterprise application projects [24, 25]) lead to several requirements for a design method, which we now use to analyze where and how pattern languages and ADMs fit in, and how they relate to each other.

4.1 Analysis of Pattern Languages and ADMs

Table 1 summarizes our analysis. Pattern languages and ADMs have different purposes, but there is a strong relationship: By definition, patterns are not the documentation of an individual system, but one source of (reusable) architectural knowledge to be considered and brought to bear when architecting a system. On the other hand, ADMs offer a way of documenting and rationalizing the decisions made and alternatives considered. Applying a pattern *is* making a decision; the consequences of applying a pattern engender more decisions. Many of the issues captured in ADMs are *domain-specific*. We identified three categories of such domain-specific decision drivers (forces): (1) *Application genre and business function*, e.g., enterprise application, core banking, order management. (2) *Technical*, e.g., a specific architectural concern and set of principles such as integration and SOA. (3) *Organizational*, e.g., company-wide guidelines, project team set up, and available skills.

Table 1. Characteristics of Pattern Languages and Architectural Decision Models (ADMs)

Characteristic/Requirement	Pattern Languages	ADMs	Assessment
Intent and main use case	Capture generic architecture and design elements as reusable solutions to commonly occurring problems	Capture and record system-specific decisions justified by project-specific decision drivers	Complementary, strong relationship; pattern application must also be captured as a decision on a project
Standard description format	Several templates, e.g., Portland style	Several meta models, UML profiles	Present in both approaches, overlap
Level of detail	Comprehensive, detailed	Terse, telegram style, details elsewhere	Patterns more detailed by definition
Top-down decomposition of problem into atomic units of design work	Objective of pattern languages and context sections; approach depends on author, often informal	Modeled explicitly in our proposal [26]: AD topic groups, AD levels; AD consists of AD alternatives	Pattern languages often informal, ADMs can add project-specific concretization and modeling rigor
Bottom-up composition of atomic units of design work into solution	Not a design goal	Not a design goal of retrospective AD capturing; supported in our proposal	Additional work for project team, integration effort required
Relationships between atomic units of design work	Informally via consequences, related patterns or pattern language diagrams (design spaces emerging, see [23])	Design goal, addressed by explicit dependency management via associations in UML meta model (directed graph)	ADMs more precise, can be populated from pattern texts
Requirements management link	Context, forces sections	Decision driver attribute in AD, decision identification step	Explicit in our proposal, informal in patterns
Links to software engineering and project management methods	Informal only, through other related patterns, or out or scope	Modeled in our proposal: scope, phase, role attributes	Integration effort required
Separate platform-independent from platform-specific concerns in models, but preserve links between concerns	Most patterns are platform-independent; platform-specific aspects come in informally via known uses and examples	Dedicated vendor asset level exists in our ADM proposal, decision dependencies can model link between levels	AD levels provide explicit support, patterns can be assigned to these levels
Ease of architectural documentation (authoring)	As patterns are publications, significant effort for pattern author (reviews, writer's workshops); easy to reference	Depends on project setup; no formal authoring and review process (yet)	ADMs less rigorous, more project-specific; patterns more sustainable, long lasting (by definition)
Ease of consumption (usability)	Depends on pattern author; time intense due to in-depth education character	Search capabilities, multiple ordering dimensions to support orientation character	ADMs provide orientation, patterns in-depth coverage of single design concern

Issues from each of these three categories influence the decision making. Patterns typically capture timeless, generic design aspects in their forces and consequences, but do not cover all domain-specific aspects that might arise for a specific pattern instance adopted in a project.

Both approaches provide standardized description formats. Several concepts can be found in both approaches, so that it is straightforward to map many pattern templates to ADMs: Pattern intent and context map to the AD problem statement; pattern forces and decision drivers in ADMs discuss NFR types and related tradeoffs; pattern consequences and related pattern information lead to additional ADs to be made (AD dependency relationships).

A pattern represents established design knowledge harvested from successful software architectures from which the pattern authors have mined the pattern; an ADM is the decision log of a specific project. Patterns use an informal description format, covering a broad variety of pattern variants. The pattern descriptions usually are provided in excessive detail, using pattern templates. An ADM from an individual project by definition is concrete and domain-specific. ADs often are captured in a telegram style; short bullet lists, text templates, tables, and forms are commonly used.

With regard to composition and decomposition capabilities, pattern languages explain the piecemeal composition of patterns. AD topic group hierarchies in ADMs serve similar purposes. Both patterns and ADMs with explicit dependency modeling support virtually any kind of relationship. In comparison, the relationships between patterns are rather informal, whereas in our extended meta-model for ADMs, AD relationships are formalized. A similar assessment can

be made for the requirements management link and the software engineering/project management link. Patterns use informal forces; the decision drivers, scope, and phase attributes of ADs make the related knowledge explicit. An important consequence is that ADMs can be used as input for model transformations and code generation.

Both patterns and ADMs cause documentation and consumption efforts. ADMs authored during the architectural documentation work on concrete projects can take domain-specific decision drivers (forces) into account. Patterns are only in seldom cases documented by the users of patterns, but by pattern authors who are experts in the domain of the pattern. Due to their level and topic structure and explicit relationships, ADMs are well searchable (esp. if their creation and consumptions is supported by tools); pattern languages cover single design concerns in more depth.

4.2 ArchPad: Combining Patterns and Decisions

Based on this analysis, we propose to leverage *Reusable ADMs (RADMs)* to steer the pattern selection and other architectural decision making activities. We call the resulting architectural pattern- and decision-based design method *ArchPad*. In ArchPad, the following types of patterns appear as AD alternatives: Analysis and architectural patterns are among the AD alternatives on the executive and conceptual level, design patterns reside on the technology level, and the vendor asset level can be populated with implementation and test patterns, as well as known uses of patterns from higher levels. We demonstrated the motivation for and feasibility of the creation of RADMs in our previous
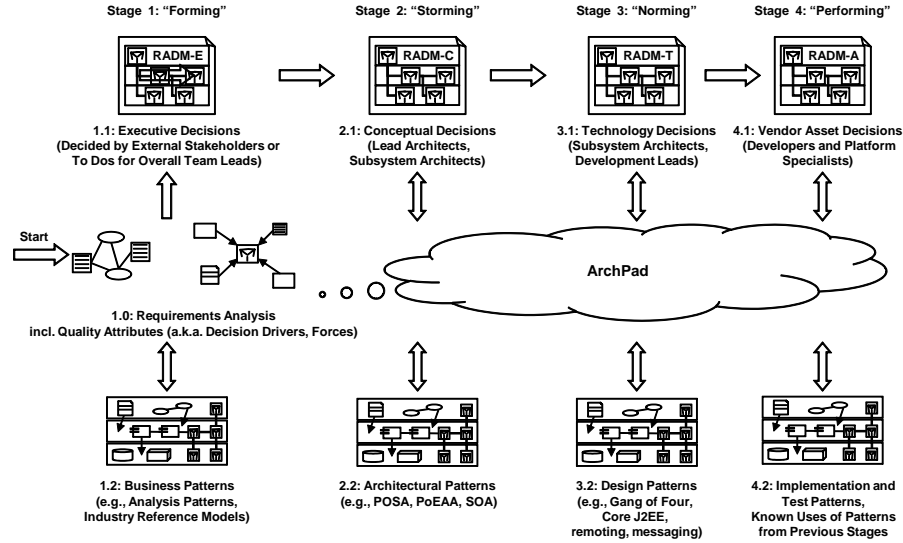
Figure 1. Refinement stages and decision/pattern correspondences in ArchPad

work [26]. A RADM that is based on patterns does not have to copy the pattern text and hence is easier to create than a self-containing one. AD outcomes can be captured in much less detail, because they only record the adoption of the patterns and can reference the patterns for further detail [10].

The relationships between RADMs and patterns in Arch-Pad are illustrated in Figure 1. The upper row represents the RADM perspective, the lower row the pattern perspective. The columns in Figure 1 represent four *refinement stages*:

- *Stage 1*: Executive decisions, requirements analysis.

- *Stage 2*: Conceptual decisions including selection of architectural patterns and key technology choices.

- *Stage 3*: Detailed technology decisions, design patterns as architecture alternatives.

- *Stage 4*: Vendor asset level decisions and selection of implementation, deployment, and test patterns.

The stages do not suggest a linear, non-invertible waterfall process, but merely information flows. The horizontal arrows in Figure 1 illustrate refinement relationships between stages; the vertical ones indicate the correspondences between architecture alternatives and patterns. Backtracking and iterating over stage boundaries is possible.

Stage 1: This stage deals with requirements analysis and executive decisions as entry points into the architecture design work. The motivation for this stage is that some non-technical analysis and planning has to happen before any technical patterns can be applied. Executive decisions reside here. The runtime platform and programming language

selection is such an executive decision. Business-level patterns and industry reference models can be utilized as background information in this stage.

Stage 2: In Stage 2, conceptual decisions are made; architectural patterns appear as AD alternatives. For instance, BROKER is an architectural pattern; deciding for or against it is a related conceptual decision.

Stage 3: In Stage 3, technological decisions are made and detailed design patterns are selected. For instance, the six implementation steps in the BROKER pattern from Section 3 fall into this stage.

Stage 4: In Stage 4, implementation and deployment related decisions are made. Discrepancies between abstract concepts and implementation reality can be discussed and documented here – e.g., vendor products often implement a conceptual pattern in a specific way, have limitations, or offer proprietary extensions. Asset-level application server, workflow engine, and other middleware selection decisions fall into this stage, e.g., to use a particular SOAP engine for XML messaging in a Web services-based BROKER.

RADMs as guides through the stages. Without reuse, the limitations of patterns and decision modeling cannot be overcome. Hence, RADMs in ArchPad are not created from scratch on every new project, but made available as a reusable asset, much like a pattern catalog or language. Leveraging experience from projects in the same domain, the RADM can be populated with patterns that already have proven to be applicable; decision paths for certain project types can be predefined. ArchPad RADMs are domain-specific companions to pattern languages and catalogs.

Like pattern texts, RADMs are "living documents": they have to be updated when new technologies emerge or pat-

terns become available. The vendor asset levels in RADMs typically face the largest amount of change over time. That is, the decisions made in Stages 1 to 3 are usually rather stable in a certain domain, but Stage 4 will have to be adapted as technology matures.

4.3 Pattern-based RADM for SOA

This section presents how we applied the concepts from Section 4.2 to the genre of enterprise applications and developed a particular *RADM for SOA*. The RADM for SOA follows the four stages introduced in Section 4.2. At present, it consists of 300 decisions covering various aspects such as Web services integration, transaction and session management, security and user management, Enterprise Service Bus (ESB) and presentation layer design; it is already in use in practitioner communities. Table 2 shows a small excerpt. The detailed information organized according to the meta model outlined in Section 2 is not shown; however, we discuss selected decision drivers, relationships, and AD alternatives informally throughout this section.

Stage 1: On this stage, executive decisions set the boundaries for the subsequent decisions. For instance, in this particular RADM, we must decide for the technologies (e.g., programming languages and runtime platforms) to be used in a project. Furthermore, we have to decide for the main architectural concepts in terms of principles, paradigms, pattern languages, and processes used. As explained earlier, these executive decisions govern which patterns are applicable in subsequent stages and which decisions have to be made in these stages. For instance, if a process-centric integration approach is selected, the SOA patterns from [12] are applicable.

Stages 2 and 3: Following these fundamental executive decisions, several conceptual decisions and then technology decisions have to be made. Often conceptual decisions are tightly coupled with related technology decisions.

Within the boundaries of the executive decisions, many architectural patterns appear as alternatives of conceptual ADs. For instance, different choices for the integration style exist, such as on which layer to integrate and whether the BROKER pattern [5] or direct client/server connectivity [22] should be used. Message exchange patterns, such as synchronous request-response, POLL OBJECT, RESULT CALL-BACK, SYNC WITH SERVER, or FIRE AND FORGET have to be selected. The granularity of the in and out parameters of service invocation messages has to be defined.

For a BROKER or a direct connection, the supported message exchange technology must be selected. For instance, technologies such as Web services, plain MOM, RPC, CORBA, proprietary vendor protocols, or even custom developed protocols can be selected. If a conceptual BROKER is used, a broker technology (e.g., commercial ESB product vs. custom integration layers) must be picked. Related decisions deal with the message exchange style and format, with options like WS-* and SOAP vs. RESTful and Plain Old XML (POX) or Java Script Object Notation (JSON) vs. custom protocols and notations. Other follow-on decisions are concerned with transport protocol binding selection and service provider type definition and API.

Remoting patterns such as PROTOCOL PLUG-INS [22] to support multiple, runtime-exchangeable protocols vs. just one hard-coded protocol are now eligible. Other pattern selection decisions depend on these decisions. For instance, the use of an architecture based on the INVOCATION INTERCEPTOR and INVOCATION CONTEXT patterns (from [22]) has to be contrasted to a simpler, but less flexible architecture. Finally, these choices are also related to the design or integration of the MARSHALLER(S). All these remoting patterns from [22] are eligible as technology options for the BROKER adoption. In ArchPad, details of these architecture alternatives do not have to be documented in the RADM, a reference to the pattern source is sufficient.

Services can be composed using custom code or using a business process engine, e.g. following the MACRO-MICROFLOW pattern [12]. If this pattern is chosen, the PROCESS-BASED INTEGRATION ARCHITECTURE pattern [12] is an important option for architectural layering. A service provider component technology has to be selected; SCA, J2EE, CORBA, and .NET are some of the alternatives. This decision implies a follow-on decision, which business process language to use. Decision drivers include the maturity of languages and tools. Other patterns from [12] also have to be considered as subsequent technology decisions, e.g.: should a RULE-BASED DISPATCHER be introduced or not; how to organize the deployment and structure of MACROFLOW ENGINES and MICROFLOW ENGINES; which CONFIGURABLE ADAPTERS and CONFIGURABLE ADAPTER REPOSITORIES to use and how, etc.

Similar technology choices appear for the presentation layer, for instance plain HTML vs. portal technology vs. Web 2.0 vs. or Rich Internet Application (RIA) vs. Eclipse Rich Client Platform (RCP). A follow-on technology decision that must be made is the presentation layer organization. Here, patterns such as APPLICATION CONTROLLER [1], PAGE/FRONT CONTROLLER [1], or HUMAN TASK LIST [18] are options. If a process-layer and a presentation layer are present, the interface between presentation and process layer also has to be designed.

Session and transaction management are important architectural concerns. For conversational state, alternatives are CLIENT SESSION STATE [8], SERVER SESSION STATE [8], or no state. This leads to a technology decision for the proper representation of state using activation strategy patterns. Resource management patterns such as LEAS-ING, POOLING, LAZY ACQUISITION, and PASSIVATION

Table 2. Excerpt from RADM for SOA

Refinement Stage	Architectural Decision (AD)	ADAlternatives (subset)
Stage 1 (RADM-E): Executive Decisions (EDs)	ED-1: Platform/Language/Tool Preferences	Technologies such as J2EE, .NET, LAMPP, Ruby on Rails
	ED-2 to ED-5: Architectural Principles, Paradigms, Patterns, Processes	Selection of reference architecture, layering approach, design method, relevant literature (many alternatives)
Stage 2 (RADM-C): Conceptual Decisions (CDs), dealing with selection of architectural patterns)	CD-1: Integration Style	Front end centric vs. integration centric vs. process centric vs. database centric [13]
	CD-2: Broker Pattern	BROKER [5] (RPC or messaging) vs. direct client/server connectivity
	CD-3: Message Exchange Pattern	Synchronous request-response, POLL OBJECT, RESULT CALLBACK, SYNC WITH SERVER, or FIRE AND FORGET [22]
	CD-4: In and Out Message Parameter Granularity	Deeply structured DOMAIN MODEL [8] elements vs. flat strings
	CD-5: Service Composition Paradigm	Business process engine (following MACRO-MICROFLOW [12] or no separation of flows) vs. custom code; PROCESS-BASED INTEGRATION ARCHITECTURE [12] or not
	CD-6: Presentation Layer Paradigm	Rich client vs. thin client vs. best of both worlds
	CD-7: Conversational State	CLIENT SESSION STATE [8] vs. SERVER SESSION STATE [8] vs. none
	CD-8: Transaction Management	SYSTEM TRANSACTIONS vs. BUSINESS TRANSACTIONS [8]
Stage 3 (RADM-T): Technology Decisions (TDs), dealing with selection of design patterns, technologies	TD-1: Message Exchange Technology a.k.a. On-The-Wire Protocol [5]	Web services vs. plain Message-Oriented Middleware (MOM) vs. plain Remote Procedure Call (RPC) vs. CORBA vs. proprietary (e.g., CICS Transaction Gateway) vs. custom protocols
	TD-2: Broker Technology	Commercial ESB product vs. custom integration layer
	TD-3: Remoting Patterns	INVOKER, CLIENT PROXY, MARSHALLER, INVOCATION INTERCEPTOR, INVOCATION CONTEXT, use of PROTOCOL PLUG-IN [22] or not
	TD-4: Message Exchange Style and Format	WS-* and SOAP vs. REST and POX/JSON vs. plain TCP/IP and custom strings vs. other
	TD-5: Transport Protocol Binding	HTTP vs. messaging
	TD-6: Service Provider Type and Application Programming Interface (API)	Enterprise Java Bean (EJB) vs. plain Java object vs. other provider in other programming language; JAX-RPC vs. JAX-WS/JAX-B vs. proprietary
	TD-7: Service Provider Component Container Technology	Service Component Architecture (SCA) vs. Java 2 Enterprise Edition (J2EE) vs. Spring vs. Common Object Request Broker Architecture (CORBA) vs. .NET vs. other
	TD-8: Business Process Language	Business Process Execution Language (BPEL), other
	TD-9: Process-based Integration Architecture Design	RULE-BASED DISPATCHER or not; Deployment and Structure of MACROFLOW ENGINES and MICROFLOW ENGINES; CONFIGURABLE ADAPTERS and REPOSITORIES; etc. (see [12])
	TD-10: Presentation Layer Technology	Plain HTML (thin client) vs. portal (thin client) vs. Web 2.0 RIA vs. Eclipse RCP
	TD-11: Presentation Layer Organization	APPLICATION CONTROLLER vs. PAGE/FRONT CONTROLLER [1] vs. HUMAN TASK LIST [18]
	TD-12: Process Layer Interface Granularity	Batch vs. conversational
	TD-13: Presentation/Process Layer Coordination	Pull (presentation leading) vs. push (process leading)
	TD-14: Presentation/Process Layer Protocol	Synchronous RPC API vs. asynchronous messaging
	TD-15: Activation Strategy Patterns	STATIC INSTANCES, PER-REQUEST INSTANCES, or CLIENT-DEPENDENT INSTANCES (all from [22])
	TD-16: Resource Management Patterns	LEASING, POOLING, LAZY ACQUISITION, and PASSIVATION (all from [22])
	TD-17: Session Management	Client (e.g., full state in cookie) vs. presentation layer (HTTP session) vs. process layer (BPEL correlation) vs. backend (database)
	TD-18: Compensation Scheme	BPEL vs. vendor-specific spheres vs. custom logic
Stage 4 (RADM-A): Vendor Asset Decisions (VDs), dealing with product selection and configuration	VD-1: ESB Product	E.g. IBM WebSphere ESB, Progress Sonic ESB, Mule
	VD-2: SOAP Message Exchange Engine	Apache Axis, Codehaus XFire, vendor engines such as IBM WebSphere engine, WSIF and Apache SOAP
	VD-3: Service Provider Container	J2EE application server with(out) EJB support, SCA container such as WebSphereProcess Server
	VD-4: Service Provider Sourcing	Make or buy; adapt or refactor existing asset
	VD-5: Business Process Engine Vendor	E.g. IBM WebSphere Process Server, Oracle BPEL Process Manager), open source (ActiveBPEL)
	VD-6: Presentation Layer Application Server	E.g. various servlet engines and portal servers, application wiki engines
	VD-7: Platform-Specific Transaction Attribute	E.g. various SCA qualifiers and EJB attributes

[22] have to be considered to optimize resource usage. The later two decisions are also related to the decision on broker technology because only in some technologies these patterns are already supported. For transaction management, either SYSTEM TRANSACTIONS [8] or compensating BUSINESS TRANSACTIONS [8] can be chosen to roll back and undo operations when handling processing errors.

Stage 4: Finally, in Stage 4, asset selection and configuration decisions are made. For instance, we have to decide for the ESB product, SOAP message exchange engine, service provider container and sourcing, business process engine, and presentation layer application server. Some alternatives for these decisions are also summarized in Table 2. Such vendor asset decisions typically refine conceptual and technology decisions made in the previous stages.

5 Case Study Walkthrough

To validate ArchPad and the RADM for SOA, we applied them to the case study. We now walk through the four steps from Section 4.2 and provide ADOutcome information for selected ADs from Table 2 as well as case study specific decision drivers and justifications from [27].

Stage 1: Already existing core banking functions merely had to be integrated; an initial Stage 1.0 analysis of the business domain therefore was not required. Other executive decisions were to use J2EE on the integration layer and to support mulitple front end platforms. The back end was decided to be based on CICS, COBOL, and DB2. The justification for these decisions can be found in the business model, operational procedures, and the project objectives.

Stage 2: The selection of the BROKER pattern was mandated by the main challenges of the case, as explained in Section 3. The invocation semantics from the consumer's perspective called for using request-reply as the message exchange pattern. To decide for the in and out message parameter granularity, the technology-specific issue of creating XML Schema (XSD) definitions for the operations defined in the WSDL contracts of the core banking components had to be taken into account. Alternatives included a deeply nested structure, representing the business DOMAIN MODEL accurately, and flat, serialized strings [27]. Decision drivers included service consumer API convenience (highly expressive, strongly typed API), acceptable message verbosity, and proven interoperability between Java, .NET, and scripting languages. In this case, the deciding factor for selecting the rich DOMAIN MODEL was that API convenience had a high priority (allowing to catch errors at development time) and the verbosity concerns could be resolved. If real-time responses or bandwidth constraints had been a top concern, a more compact textual or binary message format would have been selected.

Service composition was decided to be the responsibility of the front end application developers; therefore, no process layer was introduced.

Stage 3: On Stage 3, many technology decisions were made and design patterns selected. Aiming to reduce development and maintenance costs, Web services were chosen as a cross-platform message exchange technology. Performance and interoperability results on an early prototype had confirmed that the technology was good enough for the purposes of the project. Due to decision drivers such as licensing cost, available development skills, as well as availability of an in-house command interface on top of an adapter product (CICS Transaction Gateway), it was decided to develop a custom integration tier as BROKER technology. The Stage 4 ESB product selection decision therefore was not required. A custom integration tier introduces development and maintenance, but no licensing costs; it gives maximum control over the implementation, but has a higher technical risk than using a mature broker product.

Many design patterns and API concerns such as the decision to use JAX-RPC were mandated by the platform selected in Stage 1, Java. Due to an enterprise-wide architectural principle, the service providers were realized as plain Java objects and not as EJBs. Being supported by all service consumers in the retail banks, HTTP was selected as the transport protocol binding between the service consumers and the mid tier. Due to legacy system constraints, a proprietary protocol was chosen as the backend interface.

Session management was required due to the conversational nature of the core banking functions to be integrated (e.g., request all customer whose name starts with "Z", retrieve 10 results per request, then get the account details of customers as needed). A session management capability already existed in the backend, using it was a natural choice. This alternative simplifies the service consumer (front end) programming and minimizes the amount of data exposed on the integration channel. For transaction management, existing operational procedures mandated to handle each backend call as one atomic system transaction and implement business compensation in the front end. As a process layer had not been introduced, related patterns and business process languages did not have to be selected.

Stage 4: On the vendor asset level, several SOAP engines such as Apache Axis were evaluated. Using a standardized API was an important requirement ensuring portability and vendor independence, and the WebSphere Application Server was already in use as J2EE application server; therefore deciding for its JAX-RPC API and SOAP engine was an obvious choice because of the desire to have vendor support. This is an example for a technology decision tightly coupled with a vendor asset decision.

Evolution and reuse. In the case study, an executive decision was validated and confirmed with high level quality attribute analysis; interoperability needs, API convenience, and maintainability were important NFR types. Next, the selected architectural patterns were refined into design patterns via conceptual and technology ADs. Many new decisions originated from that refinement step, others were implied by assets strategically decided upon. However, decisions made at previous stages had to be revisited regularly. For example, when designing the second release of the solution it was decided to switch to a generic integration tier not using any deeply nested domain objects; however, during the following interoperability tests, it turned out that this design would have led to unacceptable future test and maintenance efforts. Therefore the decision was revisited and the original domain object-based design revived. This corresponds to stepping back from Stage 4 to Stage 3.

Over time, we could observe that it is possible to share and reuse the decision knowledge gained on this project: We harvested these and other ADs and produced several reusable assets ranging from an informal collection of lessons learned to a text book section, which presents 26 architectural decisions commonly required during Web services design [28].

6 Discussion and Related Work

Applicability and maturity. ArchPad originates from our practices employed on the case study project. Since then we refined it on additional projects, and evolved it into a design method also applicable to other application genres and architectural styles. Our method requires the existence of suitable patterns and decision logs harvested from several projects, which can be assembled into an RADM. This ef-

fort only pays off if many projects in a particular application domain follow the same architectural style and the specific forces and decision outcomes vary. While the four-stage design method from Section 4.2 is applicable to many project types, the RADM for SOA from Section 4.3 is specific for one particular style of enterprise application development and integration.

Pattern languages and ADMs differ in their adoption rate. Pattern usage is state-of-the-practice, whereas most projects still capture architectural decisions retrospectively and informally (if at all). When conducted in isolation, AD capturing and modeling do not produce enough immediate value to be adopted in practice. As demonstrated in Section 4.3, patterns enhance the reusability of ADMs. The value proposition of a design method in which RADMs take an active, guiding role is more appealing than that of a retrospective and therefore unwelcome documentation task.

Benefits and drawbacks. ArchPad is a rather comprehensive, but still comprehensible design method: From a pattern perspective, RADMs provide domain-specific refinements, progressing from the conceptual and technological levels to the vendor asset level (where product recommendations and limitations can be discussed). Relationships between patterns across language boundaries can be captured in them. Tool support can be provided guiding the decision maker through the decision making process. The transformation of a conceptual design to the implementation level is no longer left to developers, code generators, or off-the-shelf products. Code generators and off-the-shelf products often hard code proprietary variations of de jure or de facto standards. In contrast to so-called pattern toolkits, ArchPad offers an elaborate combination of patterns and RADMs, in which an incremental, creative process of decision making and pattern adoption is required from the project team – which can also be supported by tools. Using references to patterns, RADMs also enable a domain-specific entrance into the relevant pattern literature. Hence our approach enables the learning of relevant design knowledge in a project for new team members.

A potential drawback of our design method is that the RADMs must be combined with the existing pattern literature carefully. Content syndication and change management require architectural experience and knowledge engineering skills. Furthermore, some documentation effort for actual ADMs on projects is still required.

RADMs make it possible to rapidly identify decisions in requirements models and map the requirements to the forces in the pattern texts. This can be done semi-automatically, which also enables traceability from generic design knowledge in patterns to ADs to design models to generated code. The quality of the decision making can be expected to improve, as forces discussions in pattern descriptions can serve as additional sources of advice.

Related work. Architectural decisions and patterns are research topics that both have been covered extensively, but so far they have not been combined into a design method. Still, our work extends several recent contributions from software architecture and design decision research [14, 17, 19, 21], as well as the rich architecture knowledge captured by the patterns community [5, 8, 12, 22, 24].

Design decision research in the 1990s focused on facilitating the decision making step. Quality Options and Criteria (QOC) diagrams [19] raise a design question, which points to the available solution options; decision criteria are associated with the options. Option selection can lead to follow-on questions. In [17], Kruchten et al. extend this research by defining an ontology that describes the information needed for a decision, the types of decisions to be made, how decisions are being made, and their dependencies. In [6], Falessi et al. present the Decision, Goal, and Alternatives framework to capture design decisions. They identify why a certain approach has been chosen and which design decisions have to be updated upon changes. In our work we build on these approaches, but take their relationship with patterns into account.

Braga and Masiero [4] propose a systematic approach to organize a pattern language. They identify hot spots in a pattern language, which can be identified from the information present in the elements of the patterns. The approach thus uses a technique for understanding a pattern language similar to ours, but in contrast to our approach it aims at reducing the complexity of object-oriented frameworks rather than general relationships between patterns and decisions.

Porter, Coplien, and Winn [20] provide an approach to compose pattern languages using pattern sequences, and introduce a formalism for representing a sequence as a single, totally ordered set of patterns. Henney extends this approach using a grammar-oriented approach for deriving sequences [11]. This approach is extended by our work in [23] with graphical pattern language grammar overviews and a subsequent design space analysis. All these approaches do not offer explicit support for decision capturing and documentation, or resolving domain-specific forces, but only support generic pattern selection.

Our approach does not replace existing general-purpose software engineering processes, such as the Rational Unified Process (RUP) [16] or agile methods such as Extreme Programming (XP) [3]. Such processes take a wide range of the software design cycle into account and are independent of the design concerns in a specific application genre. In contrast, ArchPad is a domain-specific design method "plug in" for such general-purpose processes. RADMs in ArchPad focus on a particular application genre and can therefore draw on knowledge gathered from previous projects.

7 Conclusions

In this paper, we presented an architecture design method which combines pattern languages and architectural decision models to their mutual benefit. This work originates from an analysis of our decision making and pattern selection practices on industry projects. When combined, the strengths of patterns and decision models complement each other and eliminate the weaknesses and inhibitors we observed in practice. To validate this hypothesis, we analyzed the two approaches and combined them into a four-stage design method. This method provides reusable, domain-specific decision models guiding practitioners through the pattern selection and adoption process. Architectural patterns are architecture alternatives on the conceptual level, design patterns reside on the technology level. Domain-specific refinement relationships are part of the decision models, facilitating project-specific adaptation of patterns based on decision drivers such as NFRs, legacy system constraints, software quality factors, and experience from previous projects. To validate our method, we created a reusable architectural decision model for SOA and applied it to an industry case study. The validation showed that the method is comprehensive, but still comprehensible.

References

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns*. Prentice Hall, 2003.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice 1st (2nd) Edition*. Addison Wesley, Reading, MA, USA, 1998 (2003).

[3] K. Beck. *Extreme Programming Explained*. Addison Wesley, 2000.

[4] R. Braga and P. Masiero. Finding frameworks hot spots in pattern languages. *Journal of Object Technology*, 3(1):123–142, 2004.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.

[6] D. Falessi, M. Becker, and G. Cantone. Design decicion rationale: Experiences and steps towards a more systematic approach. *SIG-SOFT Software Eng. Notes 31 – Workshop on Sharing and Reusing Architectural Knowledge*, 31(5), 2006.

[7] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[8] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

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

[10] N. Harrison, P. Avgeriou, and U. Zdun. Using patterns to capture architectural decisions. *IEEE Software*, pages 38–45, July/Aug. 2007.

[11] K. Henney. Context encapsulation – Three stories, a language, and some sequences. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPloP 2005)*, Irsee, Germany, July 2005.

[12] C. Hentrich and U. Zdun. Patterns for process-oriented integration in service-oriented architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, Irsee, Germany, July 2006.

[13] G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[14] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEE/IFP Conference on Software Architecture, WICSA*, 2005.

[15] M. Keen et al. *Implementing an SOA using an ESB*. IBM Redbook, 2004.

[16] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 2003.

[17] P. Kruchten, P. Lago, and H. Vliet. Building up and reasoning about architectural knowledge. In C. Hofmeister, editor, *QoSA 2006 (Vol. LNCS 4214)*, pages 43–58, 2006.

[18] F. Leymann and D. Roller. *Production Workflow – Concepts and Techniques*. Prentice Hall, 2000.

[19] A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, options, and criteria: Elements of design space analysis. *Human-Computer Interaction*, 6(3–4):201–250, 1991.

[20] R. Porter, J. Coplien, and T. Winn. Sequences as a basis for pattern language composition. *Science of Computer Programming*, 56(1–2), 2005.

[21] J. Tyree and A. Ackerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(19–27), 2005.

[22] M. Voelter, M. Kircher, and U. Zdun. *Remoting Patterns*. Pattern Series. John Wiley and Sons, 2004.

[23] U. Zdun. Systematic pattern selection using pattern language grammars and design space analysis. *Accepted for publication in Software: Practice & Experience*, 2007.

[24] U. Zdun, C. Hentrich, and W. van der Aalst. A survey of patterns for service-oriented architectures. *International Journal of Internet Protocol Technology*, 1(3):132–143, 2006.

[25] O. Zimmermann, V. Doubrovski, J. Grundler, and K. Hogg. Service-oriented architecture and business process choreography in an order management scenario. In *OOPSLA Conference Companion*, San Diego, CA, USA, October 2005.

[26] O. Zimmermann, T. Gschwind, J. Kuester, F. Leymann, and N. Schuster. Reusable architectural decision models for enterprise application development. In S. Overhage and C. Szyperski, editors, *Quality of Software Architecture (QoSA) 2007*, Lecture Notes in Computer Science, Boston, USA, July 2007. Springer-Verlag Berlin Heidelberg.

[27] O. Zimmermann, M. Milinski, M. Craes, and F. Oellermann. Second generation web services-oriented architecture in production in the finance industry. In *OOPSLA Conference Companion*, 2004.

[28] O. Zimmermann, M. Tomlinson, and S. Peuser. *Perspectives on Web Services*. Springer-Verlag, Heidelberg, Germany, 2003.