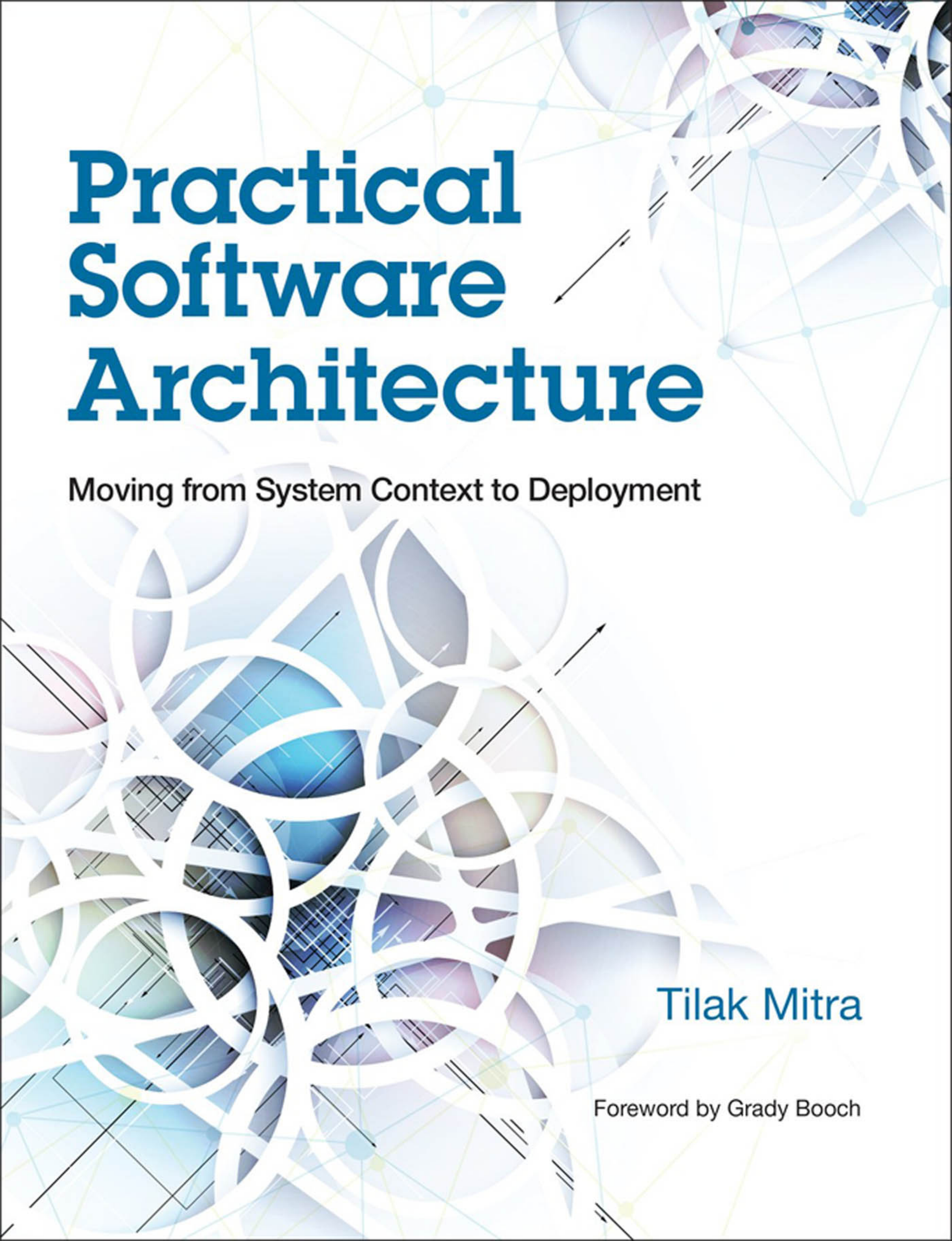
Practical Software Architecture

Moving from System Context to Deployment



Tilak Mitra

Foreword by Grady Booch

decision-making process. In the rest of this section, I focus on each of the qualitative attributes, what they mean, and how to address them.

Subject Area—Describes a specific domain of the IT System. The domains, also called subject areas, help classify the problems and challenges, which are architectural in nature. Examples of such subject areas could be Systems Management, Security, User Interface, and so on. One way to make things easy is to align the nomenclature of the subject areas with that of the architecture layers (refer to Chapter 5, "The Architecture Overview"). You can always refine them as and when they start to take shape and form.

ID (abbreviated form of **identification**)—Represents a unique number for each of the architecture decisions; for example, AD04, AD16, or AD23. Numbering helps in traceability between related architecture decisions and also may work as shorthand to refer to a particular architecture decision. Program teams often get used to referring to architecture decisions by their ID; team members know all about the decision by just referring to their ID, and when that happens, you are assured of its adoption.

Topic of Interest—Defines a topic of interest within the subject area. Although there is no hard and fast rule on a rigid set of topics, architects typically use topic elements such as efficiency, reliability, scalability, resilience, extensibility, and usability, as good starting points for categorizing the topics of interest.

Architecture Decision—Provides a descriptive name to the architecture decision under consideration. The intent is to be able to identify the architecture decision by its short descriptive name. A combination of the subject area, topic, and name typically serves to provide a quick overview of the problem at hand. As an example, the Security subject area may have a topic on Federated Identity Management with a brief problem statement entitled "Supporting user authentication in a distributed deployment topology."

Problem Statement—Provides a detailed description of the problem statement; it expands on the descriptive name captured earlier. This statement can be as descriptive as is pertinent but usually is kept to a couple of paragraphs.

Assumptions—Describe the constraints and boundary conditions that the resolution to the problem needs to adhere to. The pre-conditions and post-conditions (describing the state of the system before the problem is encountered and the state of the system after the problem is addressed, respectively) may also be stated as a measure of the architectural integrity of the overall solution that needs to be maintained with the problem resolution.

Motivations—Describe one or more incentives to address the specific problem at hand. Examples of motivations may be *to reduce complexity*, *to avoid an inordinate increase in compute with increasing workload*, *to reduce system redundancy*, and so on.

Alternatives—Illustrate the various resolution alternatives that have been considered with the objective of solving the problem under consideration. (They are possibly the most important aspect of any architecture decision.) Each alternative is described in detail along with its pros and cons, or advantages and disadvantages, in addressing the problem. The pros and cons could be in the form of technical ease or complexity, process ease or complexity, cost and time implications, among other factors. Keep in mind that it is not mandatory for all decisions to have multiple alternatives. It is okay if some architecture problems have only one alternative and that is the one chosen as the solution! The advice, though, is to consider multiple alternatives, if applicable.

Decision—Finalizes the decision by choosing the best possible solution, among the alternatives, as the resolution to the problem statement.

Justification—Describes the rationale behind choosing the solution among the various alternatives, substantiated by a list of architecture principles that the solution complies with, along with a potential list of principles that may be in noncompliance (substantiated by an explanation for the deviations).

Implications—Illustrate the consequences that the decision may have on the overall program. An implication can be limited only to the technical aspects if the decision has ramifications on the choice of tool, technology, or platform. The implication may also have consequences on program cost and timelines based on the solution characteristics; for example, implementation complexity, need for different tools or technology or platform, and so on. This element of the architecture decision template can be made optional if the decision does not have too many implications and keeps the solution well within the known constraints, boundaries, and scope.

Derived Requirements—Itemize additional requirements that may be generated by the chosen solution for problem resolution. An example of such a requirement may be the *need to add a second firewall* if the decision is to avoid placing enterprise systems in the demilitarized zone (DMZ). Similar to the implications element, this entity is also optional if no additional requirements are derived from the architecture decision. (*Note:* You don't need to rattle your brain if no additional requirements can be identified; if they exist, they would naturally surface.)

Related Decisions—Describes the set of additional architecture decisions that may be related. Including this attribute helps in decision traceability and linkage.

While looking at the attributes of an architecture decision, I have often felt that either I may miss a few of them or fail to correlate them to get a holistic view. To address this issue, I have always found that having a tabular view of the attributes provides me a more compact representation of the various characterizations of the architecture decision. To that effect, I am sharing the tabular format in Table 6.1, which you may find useful.

Table 6.1 A Tabular Format to Capture Architecture Decisions

Subject Area		ID	Topic of Interest
Architecture Decision			
Issue or Problem			
Assumptions			
Motivations			
Alternatives			
Decision			
Justification			
Implications			
Derived Requirements			
Related Decisions			

Often, I have seen that consultants have a tendency to tinker around with any template they are handed and declare, "I customized it to fit my needs!" I'm sure you have either experienced the same or have done it yourself. Now let me play the role of such a consultant.

Table 6.2 shows a customized version of the template I shared in Table 6.1 and have used in some instances. Remember: a template is only a guideline; fit it to your needs!

Table 6.2 An Example of an Architecture Decision (with a Customized Version of the Suggested Template)

Subject Area	Service Design	ID	Topic of Interest
Architecture Decision	Messaging Style for Web Services	AD007	
Issue or Problem	The impact of using RPC versus document-style encoding to the Web Services architecture of the XYZ system.		
Guiding Principles	• Maximize delivery of business capability within time and money constraints. • Minimize impact of change to Reservation System and existing Point of Sale (POS). • Minimize technology churn, system integration, and host development risk. • Need to support OTA XML.		
Motivation	Minimize performance overhead.		

Subject Area	Service Design	ID	Topic of Interest
Alternatives	*Option 1:* Use Remote Procedure Calls (RPC). RPCs using SOAP messages interact with the back-end service in an RPC-like fashion. The interaction is a simple request/response, where the client sends a SOAP message that contains a call to a method. The application server receiving this request can then translate this request into the back-end object. XML is used for data format and data interchange. *Pros:* This would require very little development effort since all the mapping of messages to the back-end object has already been implemented. *Cons:* RPC is typically static, requiring changes to the client when the method signature changes, resulting in tight coupling between the client and the service provider. In addition, it cannot support OTA XML messages. *Option 2:* Use Document Style. Document-style XML "business documents" are complete and self-contained. When the service receives an XML document, it might perform some data preprocessing, execute some business logic, and construct the response. There is no direct mapping to a back-end object. It is used in conjunction with asynchronous protocols to provide reliable, loosely coupled architectures. *Pros:* • Utilizes full capabilities of XML to describe and validate a business document. • Does not require a tight contract between the client and the service provider. Rules can be less rigid. • Is better suited for asynchronous processing because it is self-contained. • OTA XML messages, because they are document-style oriented, can be supported easily. *Cons:* It is typically more difficult to implement than RPC. The developer has to do much of the work in processing and mapping XML data received, and new tools need to be learned to implement the payload transformation.		
Decision	Use both RPC and Document Style.		
Justification	A decision was made to go with both the RPC and Document Style messaging options. Document Style will be used for transactions that lend themselves to a document-style approach (for example, OTA XML messages), and RPC based for transactions that lend themselves to an RPC-based approach. It was decided that eventually RPC messaging will be replaced by Document Style messaging because the flexibility gains outweigh the implementation costs.		

Subject Area	Service Design	ID	Topic of Interest
Implications	Need to maintain two messaging styles at the onset; that is, in the first phase of the implementation. Potential rework if and when switchover to Document Style messaging is planned.		
Derived Requirements	N/A		
Related Decisions			

The purpose of this section was to give a good glimpse and guidance on how you can develop architecture decisions. Now let's move on to the case study.

Case Study: Architecture Decisions for Elixir

Now that you've learned about the various facets of the architecture decision artifact, it's time to get back to the case study of the Elixir system. The final work product for Elixir had 10 architecture decisions. For the sake of brevity, I share two of the architecture decisions to provide a sneak peak at how it is done in real-world engagements. The two that I share (see Tables 6.3 and 6.4) are also related to each other.

Table 6.3 An Architecture Decision (AD004) for Elixir

Subject Area	Recommendations Management	ID	Topic Area
Architecture Decision	The message format of the generated recommendations from the Elixir system.	AD004	Information Architecture
Issue or Problem	One of the key outputs of Elixir is a recommendation for a possible maintenance job on any equipment. Although the currently used maintenance system is SAP Plant Maintenance (SAP PM), there is a possible migration to IBM Maximo® as the system of record for equipment maintenance and work orders. The challenge is to develop the information exchange between Elixir and the maintenance system in the most optimal manner.		
Assumptions	The current SAP PM interface supports an XML-based message format for work order submissions.		
Motivation	Lessen impact to Elixir when the maintenance system of record is migrated from SAP PM to IBM Maximo.		

Subject Area	Recommendations Management	ID	Topic Area
Alternatives	*Option 1:* Use the exposed SAP PM API to submit a requisition for a work order from Elixir. *Pros:* • Well documented and easy to use. Development team already well acquainted with the API and its usage. • Quick development time frame. *Cons:* • Implementation of the Recommendations Management subsystem will be tightly coupled to the SAP PM specific work order API. • Makes Elixir less resilient to changes when the enterprise migration to IBM Maximo is planned for implementation. *Option 2:* Leverage the MIMOSA Open O&M industry standard. Create an XML message structure that is MIMOSA compliant, and leverage the MIMOSA EAM adapter to SAP PM to pass the MIMOSA-compliant XML structure for work order creation. *Pros:* • Elixir is designed to be resilient to external changes, specifically a future migration from SAP PM to IBM Maximo. • The MIMOSA EAM adapter for IBM Maximo will accept the same MIMOSA-compliant XML message structure for work order creation. This implies that the imminent change would not affect the Elixir system too much. • Adherence to industry standard for data exchange. *Cons:* • MIMOSA-based data exchange format has a steeper learning curve for the development team. • Change from the recently concluded proof of concept where the direct SAP PM API was used. • Additional time and cost for the project.		
Decision	Go with Option 2.		
Justification	Both SAP PM and IBM Maximo products are MIMOSA compliant, and hence the message structure and format for a work order would be very similar, if not identical. This change from SAP PM to IBM Maximo would introduce minimal change to Elixir. Although there is an initial learning curve, analysis reveals that the extra time taken would be much less than it may take to revamp the Recommendations Management subsystem if Option 1 was implemented.		