SOA PATTERNS

Arnon Rotem-Gal-Oz

SAMPLE CHAPTER

MANNING

SOA Patterns

by Arnon Rotem-Gal-Oz

Chapter 1

brief contents

1 Solving SOA pains with patterns

In this chapter

- What is software architecture
- What SOA is and isn't
- Pattern structure

How do you write a book on service-oriented architecture (SOA) patterns? As I pondered this question, it led to many others. Should I explain the context for SOA, or explain the background that's needed to understand what SOA is? Should I mention distributed systems? Should I discuss when an SOA is needed, and when it isn't? After much thought, it became apparent to me: a book on SOA patterns should be a practitioner's book. If you're faced with the challenge of designing and building an SOA-based system, this book is for you.

You might not even agree with an SOA-based approach, but are forced into using it based on someone else's decision. Alternatively, you may think that SOA is the greatest thing since sliced bread. Either way, the fact that you're here, reading this, means you recognize that building an enterprise-class SOA-based system is challenging. There are indeed challenges, and they cut across many areas, such as security, availability, service composition, reporting, business intelligence, and performance.

To be clear, I won't be lecturing you on the merits of some wondrous solution set I've devised. True to the profession of the architect, my goal is to act as a mentor. I intend to provide you with patterns that will help you make the right decisions for the particular challenges and requirements you'll face in *your* SOA projects, and enable you to succeed.

Before we begin our journey into the world of SOA patterns, there are three things we need to discuss:

- *What is software architecture?* The "A" in SOA stands for *architecture*, so we need to define this clearly.
- *What is a SOA?* This is an important question because SOA is an overhyped and overloaded term. We need to clearly define the term that sets the foundation for this book.
- *How will each pattern be presented in the book?* I've used a consistent structure to explain each of the patterns in this book. We'll take a quick look at this structure so you know what to expect in the discussion of each pattern.

Let's get started with the first question—what is software architecture?

1.1 Defining software architecture

There are many opinions as to what *software architecture* is. One of the more accepted ones is IEEE's description of software architecture as the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" (IEEE 42010). My definition agrees with this one, but is a bit more descriptive:

> **DEFINITION** Software architecture is the collection of fundamental decisions about a software product or solution designed to meet the project's quality attributes (the architectural requirements). The architecture includes the main components, their main attributes, and their collaborations (their interactions and behavior) to meet the quality attributes. Architecture can, and usually should, be expressed in several levels of abstraction, where the number of levels depends on the project's size and complexity.

Looking at this definition, we can draw some conclusions about software architecture:

- *Architecture occurs early.* It should represent the set of earliest design decisions that are both hardest to change and most critical to get right.
- *Architecture is an attribute of every system.* Whether or not its design was intentional, every system has an architecture.
- *Architecture breaks a system into components and sets boundaries.* It doesn't need to describe all the components, but the architecture usually deals with the major components of the solution and their interfaces.
- *Architecture is about relationships and component interactions.* We're interested in the behaviors of the individual components as they can be discerned from

other components interacting with them. The architecture doesn't have to describe the complete characteristics of the components; it mainly deals with their interfaces and other interactions.

- *Architecture explains the rationale behind the choices.* It's important to understand the reasoning as well as the implications of the decisions made in the architecture because their impact on the project is large. Also, it can be beneficial to understand what alternatives were weighed and abandoned. This may be important for future reference, if and when things need to be reconsidered, and for anyone new to the project who needs to understand the situation.
- *There isn't a single structure that is the architecture.* We need to look at the architecture from different directions or viewpoints to fully understand it. One diagram, or even a handful, isn't enough to be considered an architecture.

For a software system's architecture to be intentional, rather than accidental, it should be communicated. Architecture is communicated from multiple viewpoints to cater to the needs of the stakeholders. The Software Engineering Institute (SEI) defines an architectural style as a description of component types and their topology, together with a set of constraints on how they can be used.

1.2 Service-oriented architecture

The term *SOA* was first used in 1996 when Roy Schulte and Yeffim V. Natiz from Gartner defined it as "a style of multitier computing that helps organizations share logic and data among multiple applications and usage modes."[1] Now, SOA is finally at the forefront of IT architectures and systems. But on the uphill and rocky road to stardom, SOA has become a loaded term filled with misconceptions and hype. As in the game of "telephone," the definition of SOA has morphed as it was passed along in informal conversations. For the purposes of this book (and my view of SOA), we'll use the following definition:

> **DEFINITION** *Service-oriented architecture* (SOA) is an architectural style for building systems based on interactions of loosely coupled, coarse-grained, and autonomous components called *services*. Each service exposes processes and behavior through *contracts*, which are composed of *messages* at discoverable addresses called *endpoints*. A service's behavior is governed by policies that are external to the service itself. The contracts and messages are used by external components called *service consumers*.

Let's take a look at common misconceptions about SOA and see why they're not SOA. Then we'll come back and expand on this definition, and SOA's benefits both architecturally and business-wise.

---

[1] Roy W. Schulte and Yefim V. Natis, SPA-401-068: "'Service Oriented' Architectures, Part 1" (report for Gartner, 1996).

1.2.1 What SOA is, and is not

Many popular terms go through what Martin Fowler calls "semantic diffusion."[2] As a term becomes more popular, people try to make it stick to whatever they're doing. Additionally, the hype, or buzz, that a new term receives generates a lot of discussion around it. If the people using the term don't understand it completely, or if they're using the term in hopes that its popularity rubs off on their product, the results are misconceptions and inaccurate descriptions.

For instance, in the late 1980s, object-oriented programming (OOP) was the hot new topic. As a result, developers referred to everything in their design, and their code, as *objects* simply because they wanted to say they were using object-oriented design and development techniques. The truth was, because the methodology was so new and the hype was so great, their descriptions were, in most cases, inaccurate. It took several years for OOP to take root and for the development world to agree upon what it truly was.

One can argue that we're at the same stage with SOA; it has garnered many misconceptions and incomplete definitions. Table 1.1 outlines the most prevalent misconceptions and explains why they are, in fact, misconceptions.

Table 1.1 Common misconceptions about SOA

Misconception	Why it's not SOA
SOA is a way to align IT and the business team.	That's not true. Better IT and business alignment is something we want to achieve using SOA, but it isn't what SOA is. Nevertheless, the loosely coupled systems that result from a good SOA solution enable the agility needed to truly align IT and the business team.
SOA is an application that has a "web service" interface.	This isn't necessarily true. To begin with, we can implement SOA with other technologies. A nice example is the Open Services Gateway initiative (OSGi), which defines a Java-based service platform (see www.osgi.org). Furthermore, by exposing a method as a web service, we can create procedural-like RPCs, which is far from the SOA concepts and direction (see also the Nanoservice antipattern in chapter 8).
SOA is a set of technologies (SOAP, REST, WS-I, and so on).	This is a general case of the previous misconception. Although some technologies are identified with SOA, or fit in well with SOA, SOA is an architectural approach. Remember, SOA is technology-independent.
SOA is a reuse strategy.	This is not always true. Reuse certainly sounds like a tempting reason to use SOA, but the larger the granularity of a component, the harder it is to reuse it. Nevertheless, SOA will allow your services to evolve over time and adapt, so that you don't need to start from scratch every time.
SOA is an off-the-shelf solution.	SOA isn't a product you can buy—it's a way to architect distributed systems. Perhaps you can resell the resulting service, but that's only a convenient artifact of a good design.

[2] Martin Fowler, "Semantic Diffusion," http://martinfowler.com/bliki/SemanticDiffusion.html.

Now that we've looked at some misconceptions, let's reexamine the SOA definition provided earlier. SOA is an architectural style. This means that SOA defines components, relationships, and constraints about each component's usage and interactions. As mentioned in the definition, the SOA style defines the following components: service, endpoint, message, contract, policy, and service consumer. SOA also defines certain interactions that the components can have. Figure 1.1 illustrates SOA's components and their relationships:

Let's take a deeper look at each of the six components of SOA.

SERVICE

The central pillar of SOA is the *service*. Merriam-Webster's dictionary has eleven different definitions for the word service; the most appropriate here is "a facility supplying some public demand."[3]

In my opinion, a service should provide a distinct business function, and it should be a coarse-grained piece of logic. Additionally, a service should implement all of the functionality promised by the contracts it exposes. One of the characteristics of services is *service autonomy*, which means the service should be mainly self-sufficient.

CONTRACT

The collection of all the messages supported by the service is known as the service's *contract*. The contract can be unilateral, meaning it provides a closed set of messages that flow in one direction. Alternatively, a contract might be bilateral, with the service exchanging messages with a predefined group of components. A service's contract is analogous to the interface of an object in object-oriented design.

ENDPOINT

An *endpoint* is a universal resource identifier (URI), such as an address or a specific place, where the service can be found. A specific contract can be exposed at a specific endpoint.

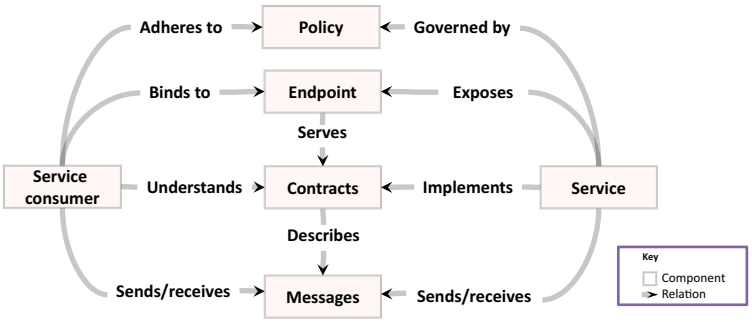

Figure 1.1 Apart from the obvious component (the service), SOA has several other components, such as the contract that the service implements, endpoints where the service can be contacted, messages that are moved back and forth between the service and its consumers, policies that the service adheres to, and consumers that interact with the service.

[3] Merriam-Webster, "service," http://www.merriam-webster.com/dictionary/service.

MESSAGE

The unit of communication in SOA is the *message*. Messages can come in many different forms, such as these:

- HTTP `GET` messages (in the representational state transfer (REST) style)
- Simple Object Access Protocol (SOAP) messages
- Java Message Services (JMS) messages
- Simple Mail Transfer Protocol (SMTP) messages

The difference between a message and other forms of communication, such as a remote procedure call (RPC), is subtle. An RPC often requires the caller to have intimate knowledge of the other system's implementation details. With messaging, this isn't the case. Messages have both a header and a body (the *payload*). The header is usually generic and can be understood by infrastructure and framework components without knowing implementation details. This reduces dependencies and coupling. The existence of the header allows for infrastructure components to route reply messages (for example, the routing of messages in the Saga pattern in chapter 5) or implement security transparently (see the Service Firewall pattern in chapter 4).

Messages are a very important part of SOA, and they've been thoroughly covered by other books, such as *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf (Addison-Wesley Professional, 2004). Nonetheless, this book also explores some messaging patterns where the SOA perspective enhances the more generic perspective used in Hohpe and Woolf's book. As an example, see the Request/Reply pattern in chapter 5.

POLICY

One important differentiator between SOA and object-oriented design (or even component-oriented design) is the existence of *policies*. Just as an interface or contract separates specifications from implementations, policies separate dynamic specifications from static or semantic specifications.

A policy defines the terms and conditions for making a service available for service consumers. The unique aspects of policies are that they can be updated at runtime and they're externalized from the business logic. A policy specifies dynamic properties, such as security (encryption, authentication, authorization), auditing, service-level agreements (SLAs), and so on.

SERVICE CONSUMER

A service is only meaningful if another piece of software uses it. *Service consumers* are the software components that interact with a service via messaging. Consumers can be either client applications or other services; the only requirement is that they adhere to an SOA contract themselves.

1.2.2 SOA architectural benefits

By definition, SOA brings many architectural benefits to a distributed software system. Many quality attributes are addressed, such as these:

- *Reusability*—This isn't reusability in the sense of "write once integrate anywhere," but rather in the sense that you "don't throw everything out when you need different functionality."
- *Adaptability*—Isolating the internal structure of a service from the rest of the world lets you make changes more easily. You only need to adhere to the contracts you publish.
- *Maintainability*—Services can be maintained by dedicated, smaller teams and can be tested this way as well. Robert L. Glass has said, "software maintenance is a solution, not a problem".[4] SOA greatly helps make this a reality.

These benefits exist because SOA removes the dependency issues related to point-to-point integration.

Many enterprises have grown isolated systems to solve particular business needs. These are sometimes referred to as *stovepipe systems*. As time passes and business needs change, there's often a need to share data between systems. Each time such a need is identified, a new relationship is formed between these systems. The result, as seen in figure 1.2, is an integration mess that becomes very hard to maintain and evolve over time.

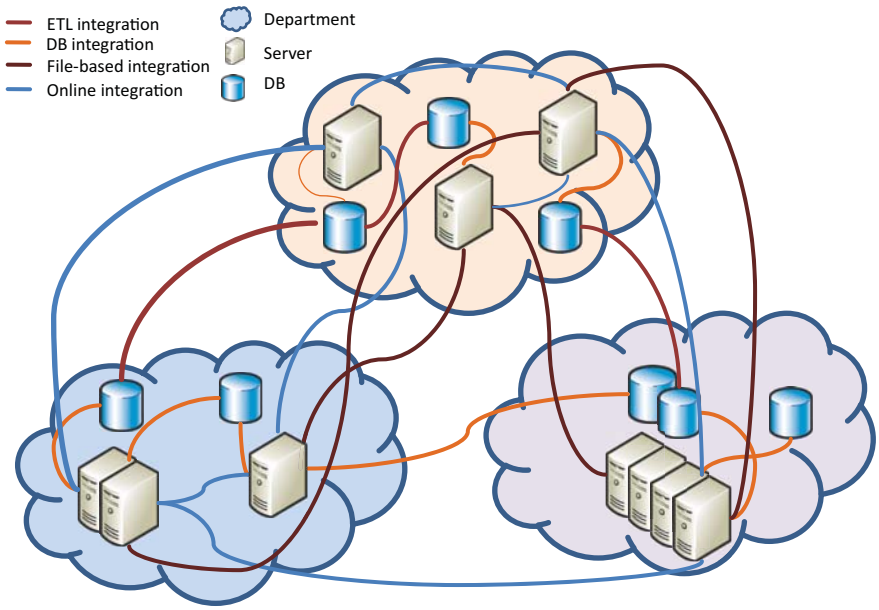

Figure 1.2 Typical integration spaghetti in enterprise systems. Each department builds its own systems, and as people use the systems, they find they need information from other systems. Point-to-point integration emerges.

[4] Robert L. Glass, *Software Conflict 2.0: The Art and Science of Software Engineering* (Developer.* Books, 2006), 61–65.

The diagram shows four types of point-to-point integrations:

- *ETL (extract, transform, load)*—Database-to-database integration or other ETL-based integration
- *Online integration*—Application-to-application integration based on HTTP or TCP
- *File-based integration*—Application-to-application integration based on the filesystems and the exchange of files (such as comma-delimited files)
- *Direct database connection*—Application-to-database integration

NOTE The preceding list isn't exhaustive. There are additional relationships such as replication, message-based relationships, and others that aren't expressed in figure 1.2.

In a well-defined SOA, the interfaces aren't designed to be point-to-point but are instead more generalized to serve many anonymous consumers. SOA eliminates this spaghetti and introduces more disciplined communication. Fewer connectors means less maintenance and fewer assumptions. Fewer connectors also result in increased flexibility, as shown in figure 1.3.

For enterprises that support a heterogeneous environment, with multiple operating systems (OSs) and platforms, SOA provides standards-based contracts that are platform-independent. In fact, SOA enables transparent interoperability among services and applications across platforms.

Policy-based communications also greatly enhance the maintainability and adaptability of SOA-based solutions because key aspects, like security and monitoring, are configurable. This moves some of the responsibility from the development team to the IT staff and makes life easier for both parties.

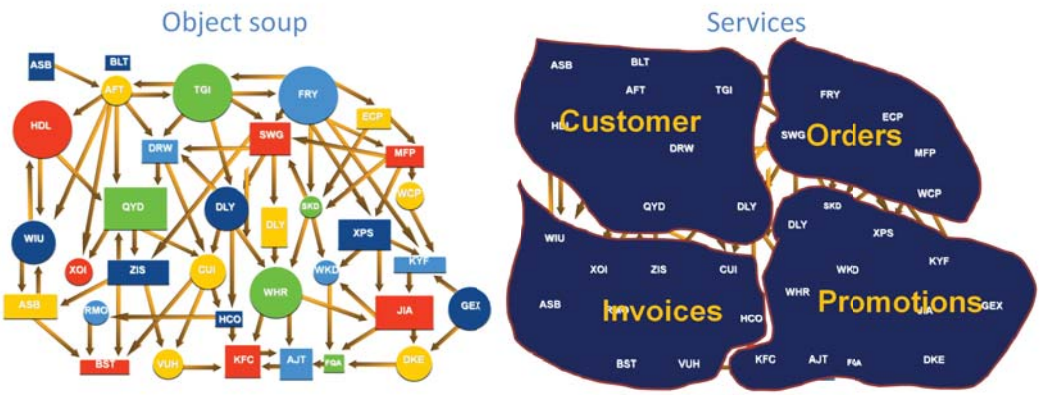

Figure 1.3 From object soup to well-formed services; one of the ideas behind SOA is to set explicit boundaries between larger chunks of logic, where each chunk represents a high-cohesion business area. This is an improvement on the more traditional approach, which more often than not results in an unintelligible object soup.

We can take all of these architectural benefits and translate them to business benefits, as discussed in the next section.

1.2.3 SOA for the enterprise

There are a lot of business-oriented aspects of SOA as well. SOA is described as a way to "increase the alignment of IT and the business." Essentially, increased alignment means that IT can adapt more easily to the changing business processes, and thus increase your business's agility.

To avoid overloading the term SOA, I'd like to refer to these aspects of SOA as "SOA initiatives." Table 1.2 points out some of these business benefits.

Table 1.2 SOA technical benefits and the business benefits they provide

SOA characteristic	Business benefit
Easier maintenance and replacement of components	Easier replacement of existing business components Better adaptability to accommodate changing business processes Faster time to market for new business functionality
Standards-based service interfaces (contracts)	Reduced effort to connect new systems Easier partner integration Enables automation of business process
Service autonomy	Reduced downtime and lower operational costs
Externalized policies	Ability to set service-level agreements Easier integration

In general, it's best to take an incremental approach to adopting SOA—your business can't afford to halt and wait for an SOA initiative to finish. You need to plan for SOA-like highway intersections; detours need to be created to enable business to continue while the new system is being developed.

Many SOA books cover the business aspects of the SOA initiatives, and this book isn't one of them. This book's scope is the software architecture aspects of SOA and technological implications of these aspects, not business analysis and related methods. One of the best ways to express these software architecture concerns and provide a better understanding of the architectural solutions is through the use of patterns (best practices) and antipatterns (lessons learned and mistakes to avoid).

1.3 Solving SOA challenges with patterns

Given all its benefits, why would anyone choose *not* to build with SOA? The truth is, building with SOA isn't easy. Even though SOA is designed to face the challenges of distributed systems design, there are still many issues you need to take care of and solve when you design viable solutions.

One set of problems is the quality attributes not inherently addressed by SOA, like availability, security, scalability, performance, and so on. Real projects have to deal

with requirements like *five-nines availability* (99.999 percent uptime), which is no more than about five minutes of downtime per year.

Another set of problems has to do with the challenges of designing and building SOA. How do you gain a centralized view of business data in an architectural style that encourages encapsulation and privacy? What does it mean to aggregate services? How do you tie your services to a UI?

It would be nice if there were a few best practices already defined that could tell us how to cope with all of these issues. The truth is that there are no silver bullets in software design and development. Every system has its own set of prerequisites, hidden costs, one-off requirements, and special case exceptions. This is exactly why the use of patterns is so appealing as a medium to convey solutions. Patterns aren't defined to be perfect solutions. Instead, they give the context for where the solution works. To achieve this, patterns describe both the solution *and* the problem they solve, and any caveats associated with that solution.

The following section explains the pattern structure used in this book and demonstrates how to apply the patterns to your own set of design challenges.

1.3.1 Pattern structure

Patterns in this book mostly take after what is called the *Alexandrian form*, which is named after the style Christopher Alexander and his coauthors used in their book, *A Pattern Language*.[5] In this form, pattern descriptions are narrative with a few headings for readability, and they serve as a vocabulary for both designers and architects.

To start, each pattern has a descriptive name that's easy to remember and recall. The name is followed by a short narrative passage to introduce the problem, which is the first subsection. The other subsections in the pattern's description are solution, technology mapping, and quality attributes.

Let's examine the pattern form, and each of the subsections, in more detail now.

PROBLEM

The problem section, as its name implies, details the problem the pattern aims to solve. It includes a problem statement that summarizes the essence of the problem. More complex problems have an additional passage, prior to the problem statement, that details the problem's context. For instance, some patterns contain an example to help illustrate the problem.

Following the problem statement, the section often continues with a discussion of other related options—often a discussion of alternative solutions and why they fail to solve this particular problem (though these alternative solutions may still be applicable in other circumstances).

---

[5] Christopher Alexander, Sara Ishikawa, and Murray Silverstein, *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977).

SOLUTION

The solution begins with a solution statement that summarizes the essence of the solution. A diagram that serves as a visual representation of the solution's components and their relationships follows the solution statement.

The same diagram conventions are used for all the patterns, with different visualizations for the SOA components (see figure 1.1) and other neutral players. The figures include component relationships, other pattern components, attributes, and the functionality of the pattern's components. Take a look at figure 1.4.

Without getting into the details of the roles of the different components, in this diagram you can see that edge and endpoint are neutral components that aren't part of the pattern. The dispatcher and service instance components are part of the pattern. Each of the pattern's parts has one or more roles and attributes. In this case, you can see that the dispatcher is responsible for the distribution (of messages) and that the service instance is responsible for (running) the service business logic. The dispatcher and service instance are part of the pattern, while the innermost rectangles designate roles or attributes of the pattern's components (for instance, the dispatcher distributes messages). The arrows are used to show interactions and relationships. Requests and replies are passed back and forth between the dispatcher and service instance, for example.

The pattern description then continues with more details regarding the solution, such as how the solution addresses outside forces, and so on. There may be a discussion of the implications or consequences of applying the pattern as well as the relationship to other patterns and examples.

TECHNOLOGY MAPPING

The technology mapping section of the pattern description deals with technology implications. Although a system's architecture can be technology independent, a set of technologies must be chosen to build the system. Therefore, as a practicing architect, you often need to map parts of the architecture to specific technologies.

For SOA, there are many relevant technologies, such as the WS-* protocol stack, REST-based web services, dedicated products, EDBs, and many others. The technology

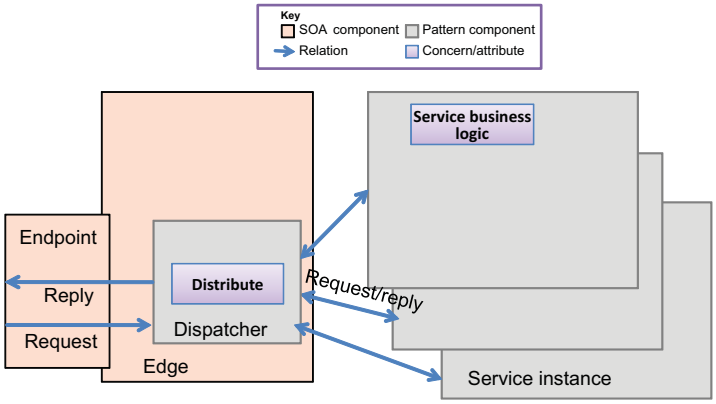

Figure 1.4 Sample pattern diagram: the Service Instance pattern. The endpoint and edge are two neutral components (not part of the pattern).

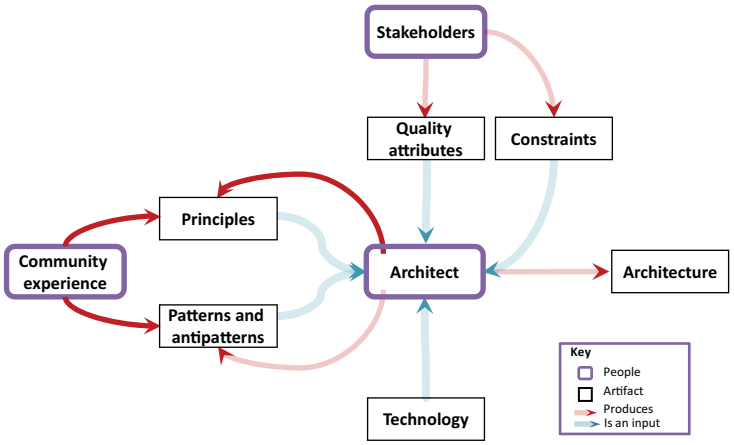

Figure 1.5
The architect uses various inputs to design the architecture.

mapping section of each pattern talks about the relevant technologies that can be used to implement the pattern or where the pattern is implemented.

QUALITY ATTRIBUTES

The final section of the pattern description has to do with identifying applicable patterns for your solution. If patterns are the solutions, then quality attributes are the requirements. The quality attributes section of each pattern talks about the architectural benefits of the pattern and provides sample scenarios that can be used to identify the pattern as relevant.

In figure 1.5, you can see the various inputs the architect can use before a solution is designed.

First and foremost, you work with the constraints and requirements gathered from the stakeholders. These include requirements for performance, security, scalability, and interoperability. You can augment these inputs by drawing on personal and community experience to add principles, patterns, and antipatterns. There are also the possibilities and constraints imposed by available technologies. Finally, you must analyze, prioritize, and balance all of these inputs to produce a final architecture to suit the problem.

Appendix A includes a cross-reference from quality attributes back to pattern names (and the chapters they're discussed in), and it provides some more background on quality attributes and quality attribute scenarios.

1.3.2 *From isolated patterns to a pattern language*

Each pattern on its own provides useful information and describes a good practice. As mentioned, patterns have relationships to other patterns—sometimes another pattern is an alternative, and sometimes patterns can complement one another. There is usually value in documenting these relationships, and this structural organization is called a "pattern language."

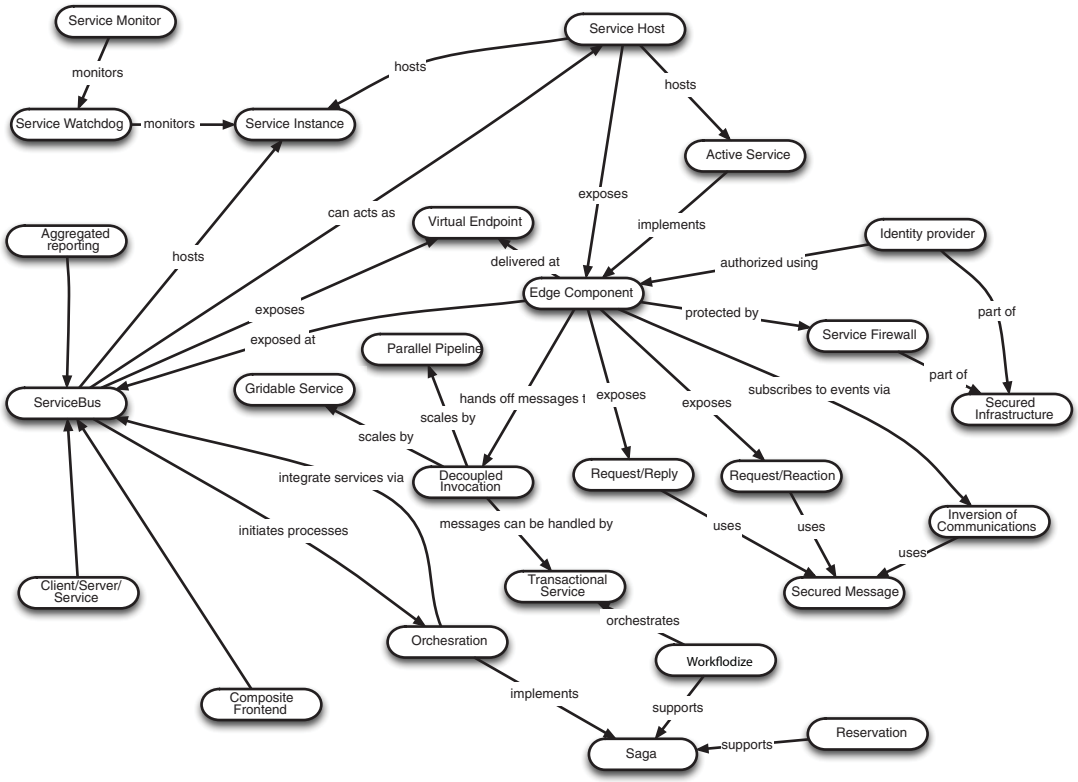

Figure 1.6 **Like any good pattern language, the SOA patterns in this book build upon each other to provide a big-picture solution.**

Evolving patterns into a pattern language that shows the patterns' relationships helps enable us to recognize related problems, and allows the architect to navigate the patterns in a logical way. In a sense, you can think of a pattern language as a logical and intuitive "mind map" of the patterns that lets you take different paths through the design process. As a result, patterns often open your mind to the bigger-picture problems that need to be solved, and provide an overview perspective you may not have had before (see figure 1.6).

Table 1.3 shows how the patterns in this book are categorized, and in which chapters they are discussed. Note that as you progress from chapter to chapter, you'll be moving outward. The first two pattern chapters (chapters 2 and 3) mostly deal with the internal structure of services. Chapter 4 focuses on the service interface, chapters 5 and 6 focus on the service consumer and its interaction with the service, and chapter 7 focuses on SOA as whole.

When you encounter a problem in your SOA implementation, you can use both the pattern diagram in figure 1.6 and the pattern categories in table 1.3 as roadmaps

Table 1.3 Pattern categories and the chapters they're discussed in

Category	Subcategory	Description	Chapter
Service structure	Foundation patterns	Common service building blocks	2
	Performance, availability, and scalability	Patterns to solve scalability, availability, and performance challenges	3
	Security and manageability	Patterns for securing and managing services	4
Integration	Message exchange patterns	Patterns for communication between services	5
	Consumer interaction	Interaction patterns for when the consumers are user clients or other services	6
Service composition		Patterns for making services work together and share information	7

to help you locate patterns that should be useful. The patterns diagram can also help you find related patterns to create more complete solutions.

1.4 Summary

We've now laid the foundation you need to understand the SOA patterns in this book and their overall context. We began with a definition of SOA and patterns in general, and we considered how patterns can be used to provide solutions to SOA challenges. We also looked at the technical and business benefits of SOA. The second part of this chapter explained what patterns are, the structure of the patterns as they'll be discussed in this book, and how to locate the patterns discussed in the book.

This chapter covered a lot of issues very briefly in order to create a common vocabulary for our discussion of SOA patterns. If you're interested in learning more about the issues discussed in this chapter, look at one or more of the resources listed in the further reading section.

Chapter 2 is our first pattern chapter, in which we'll take a look at some of the basic patterns used to build services.

1.5 Further reading

DISTRIBUTED SYSTEMS

Chris Britton, *IT Architectures and Middleware: Strategies for Building Large, Integrated Systems* (Addison-Wesley Professional, 2004).

Provides a good look at the history of distributed systems and the inherent difficulties that they inflict. It's a very thorough book—the only problem is that it ends just before the SOA era.

FALLACIES OF DISTRIBUTED COMPUTING

Arnon Rotem-Gal-Oz, "Fallacies of Distributed Computing Explained," www.rgoarchitects .com/Files/fallacies.pdf.

SOA is an architectural style for distributed systems. Most other styles don't have a distributed mindshare and so, unlike SOA, they disagree with the fallacies. This paper, which I wrote, explains how the fallacies are still relevant today.

SOA

Dirk Krafzig, Karl Banke, and Dirk Slama, *Enterprise SOA: Service-Oriented Architecture Best Practices* (Prentice Hall, 2004).

This is one of the best books on SOA, and it provides a very good introduction to the subject.

Eric A. Marks and Michael Bell, *Service-Oriented Architecture: A Planning and Implementation Guide for Business and Technology* (Wiley, 2006).

Marks and Bell take a look at the business perspectives of SOA and provide a completely different (and complementary) look at SOA, as compared to this book.

SOA/ENTERPRISE

SOA PATTERNS

Arnon Rotem-Gal-Oz

The idea of service-oriented architecture is an easy one to grasp and yet developers and enterprise architects often struggle with implementation issues. Here are some of them:

- How to get high availability and high performance?
- How to know a service has failed?
- How to create reports when data is scattered within multiple services?
- How to make loose coupling looser?
- How to solve authentication and authorization for service consumers?
- How to integrate SOA and the UI?

SOA Patterns provides detailed, technology-neutral solutions to these challenges, and many others, using plain language. You'll understand the design patterns that promote and enforce flexibility, availability, and scalability. Each of the 26 patterns uses the classic problem/solution format and a unique technology map to show where specific solutions fit into the general pattern.

Written for working developers and architects building services and service-oriented solutions. Knowledge of Java or C# is helpful but not required.

Arnon Rotem-Gal-Oz has over a decade of experience building SOA systems using Java and C#. He's a recognized authority in designing and architecting distributed systems in general and SOAs in particular.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SOAPatterns

"Documents a significant body of knowledge on SOA."

—From the Foreword by Gregor Hohpe, coauthor of *Enterprise Integration Patterns*

"An essential guide."

—Glenn Stokol
Oracle Corporation

"For people who actually have to ship code."

—Eric Farr
Marathon Data Systems

"Patterns are hard; this book makes them easy."

—Robin Anil, Google

"Brings structure to the wild SOA landscape."

—Rick Wagner, Red Hat

ISBN 13: 978-1-933988-26-9
ISBN 10: 1-933988-26-6
54999
9 781933 988269

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]