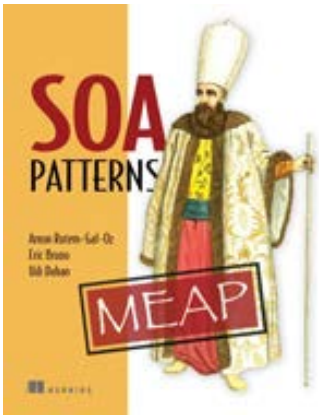
MANNING PUBLICATIONS



SOA Patterns
By Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan

You might not even agree with an SOA-based approach, but are perhaps forced into using it based on someone else's decision. Alternatively, you may think that SOA is the greatest thing since sliced bread. This green paper from SOA Patterns explains patterns that will help you make the right decisions for the particular challenges and requirements you'll face in your SOA projects.

The World of SOA Patterns

You might *not* even agree with an SOA-based approach, but are perhaps 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, performance, and so on.

To be clear, our goal here is not to lecture you on the merits of some wondrous solution set we've devised. True to the profession of the architect, our goal is to act as a mentor. We 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. However, before we begin our journey into the world of SOA patterns, there are three things we need to discuss first:

1. *What is software architecture?* Since the "A" in SOA stands for architecture, we need to define this clearly.
2. *What is a service-oriented architecture (SOA)?* This is an important question because SOA is an over-hyped and overloaded term.
3. *Explain the structure used to present each pattern in the book.* This will help them be more usable and readable to you as you move through the book.

Software architecture

There are many opinions of what software architecture is. In fact, the IEEE describes software architecture as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution (IEEE1471). Our definition agrees with this one but is a bit more descriptive:

DEFINITION: ARCHITECTURE & ARCHITECTURE STYLE.

Software architecture is the collection of fundamental decisions about a software product/solution designed to meet the project's quality attributes, or architectural requirements. The architecture includes the main components, their main attributes, and their collaboration (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.

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 international, every system has an architecture.
- *Architecture breaks a system into components and sets boundaries*. It doesn't need to describe all the components, but it usually deals with the major components of the solution, and their interfaces.
- *Architecture is about relationships and component interaction.* We are interested in the behaviors of the component as it can be discerned from other components interacting with it. Also, it doesn't have to describe the complete characteristics of components, but it mainly deals with their interfaces and other interactions.
- *Architecture explains the rationale behind the choices*. It is important to understand the reasoning as well as the implications of the decisions made in the architecture since their impact on the project is large. Also, it may be beneficial to understand what alternatives were weighted and abandoned. This may be important for future reference, if and when things needs to be reconsidered and for anyone new to the project that needs to understand the situation.
- *There isn't a single structure that is the architecture*. There's a need to look at the architecture from different directions or viewpoints to fully understand it. One or even a handful diagrams is not enough to be called architecture.

In order for a software system's architecture to be intentional, rather than accidental, it should be communicated. Architecture is communicated from multiple viewpoints to cater the needs of the different 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.

Service-oriented architecture

Arguably, the term SOA was first used in 1996 when 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." Now, many years later, SOA is finally at the forefront of IT architectures and systems. However, 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. Our definition of SOA is as follows:

DEFINITION: SERVICE ORIENTED ARCHITECTURE

Service Oriented Architecture (SOA) is an architectural style for building systems based on orchestrating 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.

Before we dive deeper into this definition, let's take a brief look at some of the misconceptions and see why they're *not* SOA. Then we'll go back and expand some more on this definition and its benefits both architecturally and business-wise.

What SOA is, and is not

Many popular terms go through what Martin Fowler calls "semantic diffusion." For instance, as a term becomes more popular, people try to make them stick to whatever it is they're doing. This occurs either because they don't understand it precisely or due to other, political, reasons. Additionally, the hype, or buzz, that a new term receives results in a lot of discussion around it. If the people discussing it don't understand it completely, the results are misconceptions and inaccurate descriptions.

For instance, in the late 1980's, 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. Therefore, 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 in the same stage with SOA; it has garnered many misconceptions and incomplete definitions. Table 1 details the most prevalent ones and explains why they are what we've said they are—misconceptions:

Table 1 SOA, like many other popular terms, has created many misconceptions. As a term gets popular, people are more likely to brand whatever it is they are doing with it—whether it's an accurate description or not.

Popular SOA misconception	Why it's not SOA
SOA is a way to align IT and the business team	No, that's not true. Better IT and business alignment is something we want to achieve using SOA, but it isn't what SOA is. However, 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 is not necessarily true. To begin, 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.ogsi.org). Furthermore, exposing a method as a web-service can also be used to create procedural-like RPC, which is very far from SOA concepts and direction.
SOA is a set of technologies (SOAP, REST, WS-I, and so on)	This is a general case of the previous misconception. Still, while some technologies are identified with SOA, or make a good fit when implementing them, SOA is an architectural approach. Remember, an SOA is technology-independent.
SOA is a reuse strategy	This 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 is not a product you can buy—it is a way to architect distributed systems. Perhaps you can resell the resulting service but that's only a convenient artifact of a good design.

After looking at these misconceptions, let's now re-examine the SOA definition provided above. SOA is an architectural style. This means that SOA defines components, relationships, and constraints about each component's usage and interactions. Following our definition above, the SOA style defines the following components: service, end point, message, contract, policy, and service consumer. SOA also defines certain interactions that the components can have. Figure 1 lists SOA's components and their relations.

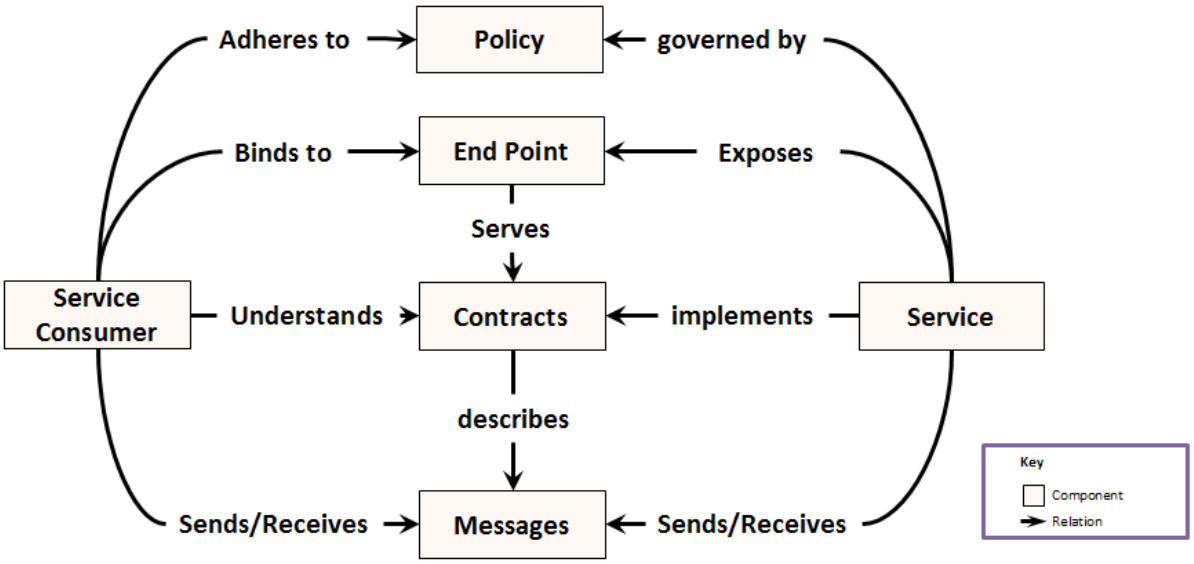


Figure 1 Apart from the obvious component (the service), SOA has several other components such as the contract that the service implements; end points where the service can be contacted; messages that are moved back and forth between the service and its consumers; policies that the service adheres to; and consumers that interact with the service.

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

SERVICE

The central pillar of SOA is the *service*. *The Merriam-Webster Unabridged Dictionary* has eleven different definitions for the word service; the best is "a facility supplying some public demand." In our opinion, a service should provide a distinct business function, and 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 collectively 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, sending messages within a predefined group of components. A service's contract is analogous to the interface of an object in object-oriented design.

END-POINT

An *end-point* 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 end-point.

MESSAGE

The unit of communication in SOA is the *message*. Messages can come in many different forms. For instance, they can be:

- HTTP GET messages, part of the Representational State Transfer (REST) style.
- Simple Object Access Protocol (SOAP) messages.
- Java Message Services (JMS) messages.
- Simple Mail Transfer Protocol (SMTP) messages.

The differentiator 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 is not the case. For instance, messages have both a header and a body (the payload). The header is usually more generic and can be understood by infrastructure and framework components without knowing implementation details. As a result, it reduces dependencies and coupling. The existence of the header allows for infrastructure components to route reply messages (for example, the *Correlated Messages* pattern), or implement security transparently (the *Firewall* pattern).

POLICY

One important differentiator between SOA and object-oriented design (or even component-oriented design) is the existence of policy. Just as an interface or contract separates specification from implementation, a policy separates dynamic specification from static/semantic specification. A policy represents the conditions for the semantic specification availability for service consumers. The unique aspects of a policy are that it can be updated in runtime and that it is externalized from the business logic. The policy specifies dynamic properties, such as security (encryption, authentication, authorization), auditing, service-level agreements (SLA), and so on.

SERVICE CONSUMER

A service is only meaningful if there is another piece of software that uses it. Therefore, we define service consumers as 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.

SOA architectural benefits

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

- Reusability—not in the sense of "write once integrate anywhere" but rather in the sense that "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 once said, “Software maintenance is a solution, not a problem.” SOA greatly helps to make this a reality

These benefits exist because SOA removes the dependency issues related to point-to-point integration. Consider for example the situation in figure 2.

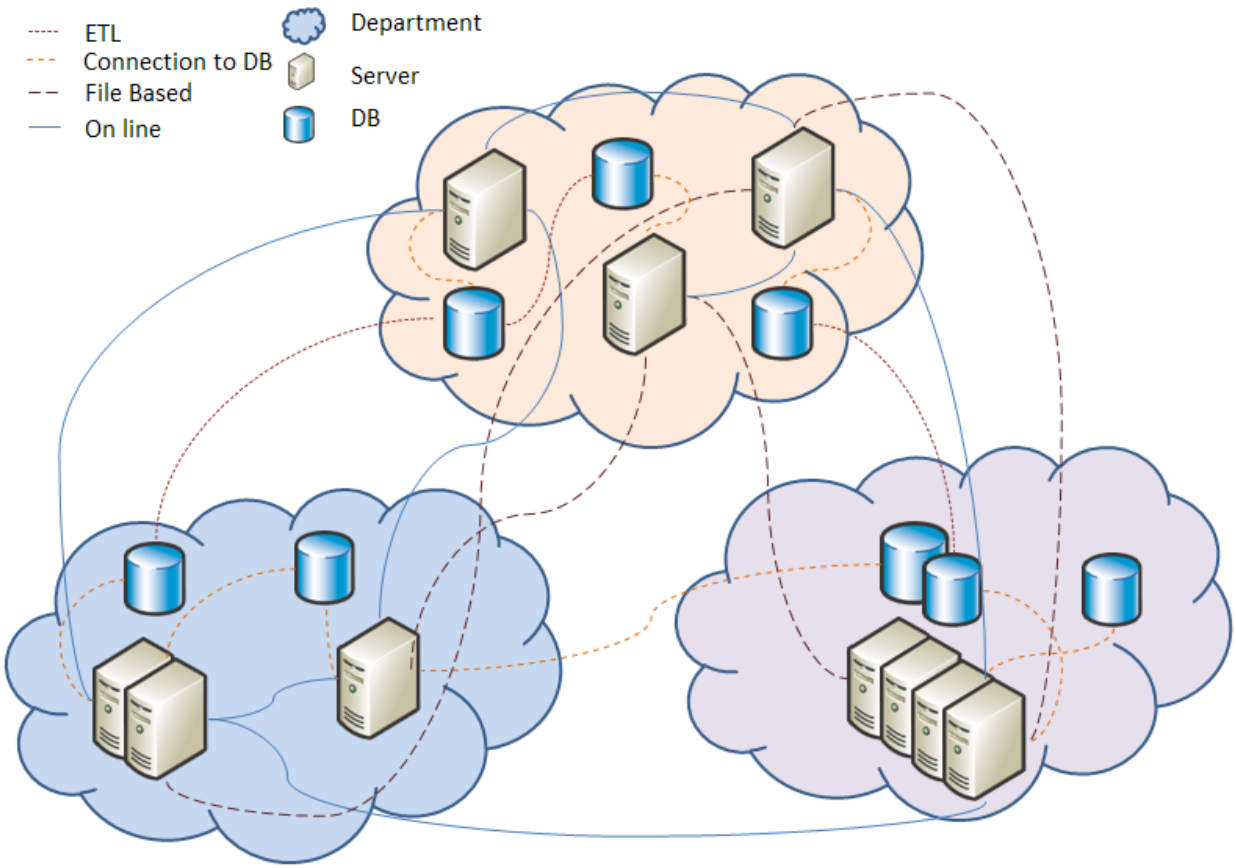


Figure 2 Typical enterprise systems integration spaghetti. Each department builds its own systems. As people use the systems, they find they need information from other systems and point-to-point integration emerges.

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 is 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 2, is an integration mess that becomes very hard to maintain and evolve over time. The diagram shows four types of point-to-point integrations:

1. Extract, Transform, Load (ETL)—A database-to-database relationship
2. On-line—An application-to-application relationship based on HTTP
3. File-based—An application-to-application relationship based on the file system
4. Direct database connection—An application-to-database relationship

Note that this is not an exhaustive list; there are additional relationships such as replication, message-based, and others that are not expressed in this diagram.

In a well-defined SOA, the interfaces are not 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 mean less maintenance and reduced assumptions and result in increased flexibility, as shown in figure 3.

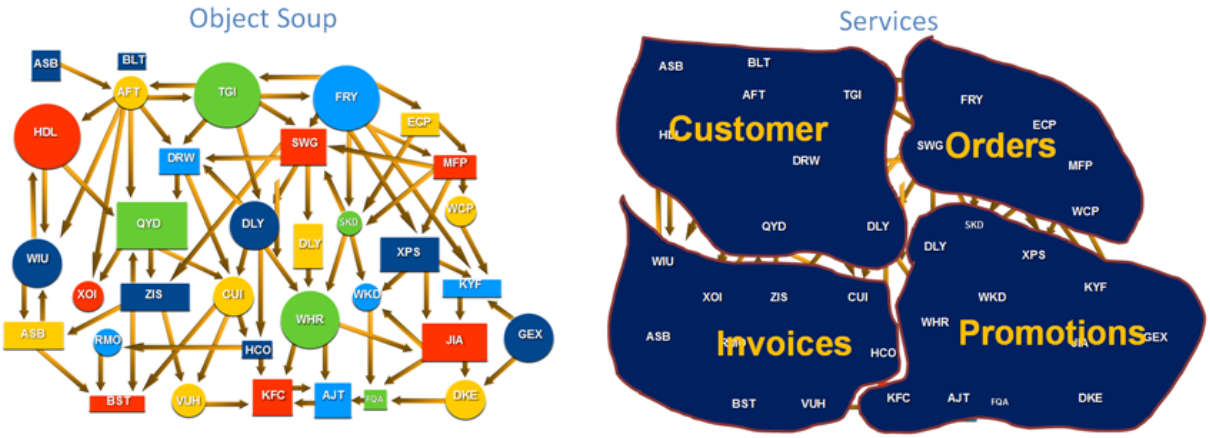


Figure 3 From object soup to well-formed services; one of the ideas behind SOA is to set explicit boundaries between larger chunks of logic, where each chunk represents a high-cohesion business area. This is an improvement on the more traditional approach, which more often than not results in an unintelligible object soup

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

Policy-based communications also greatly enhance the maintainability and adaptability of SOA-based solutions because several 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.

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

SOA an architectural style for the enterprise

There are a lot of business-oriented aspects for SOA. SOA is described as a way to "increase the alignment of IT and the business." This allows for greater adaptability of IT to the changing business processes and thus increases your business's agility. To avoid overloading the term SOA, we'd like to refer to these aspects of SOA as SOA initiatives. Table 2 points out some of these business benefits:

Table 2 Moving to SOA is not a trivial effort, but SOA's technical benefits can make for some very desirable business benefits.

SOA characteristic	Business benefit
Easier maintenance, 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 • Enable automation of business process
Service autonomy	• Reduced downtime and lower operational costs
Externalized policy	• Ability to set service-level-agreements • Easier integration

In general, it's better to take an incremental approach to adopting SOA. Your business cannot afford to halt and wait for the SOA initiative to finish, and you need to plan for SOA like highway intersections are planned; detours need to be created to enable business to continue while the new system is being developed.

One of the best ways to express software architectural aspects of SOA and technological implications of these aspects and provide a better understanding of the architectural solutions is through the use of patterns (best practices) and anti-patterns (lessons learned, and mistakes to avoid).

Solving SOA challenges with patterns

With all the benefits mentioned above, 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 actually 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% 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. For instance, 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 user interface?

It would be nice if there were a few "best practices" already defined to 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 a perfect solution. Instead, they give context where the solution works. To achieve this, patterns describe both the solution *and* the problem they solve, and 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.

Patterns structure

Patterns in this book mostly take after what is called the *Alexandrian form*, which is named after the style Christopher Alexander used in his book, *A Pattern Language*. With this form, pattern descriptions are narrative with a few headings for readability, and serve as a vocabulary for both designers and architects. To start, each pattern has a descriptive name that is easy to remember and recall when applicable. The name is then followed by a short narrative passage to introduce the problem, which is the first heading. The other headings in the patterns description are *solution*, *technology mapping*, and *quality attributes*. Let's examine the pattern form, and each heading in more detail now.

Problem

The problem section, as its name implies, details the problem the pattern aims to solve. It usually begins with a problem statement in bold 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, this section often continues with a discussion on other related. For example, there may be a discussion of alternative solutions and why they fail.

Solution

The solution begins with a solution statement (again in boldface) that summarizes the essence of the solution. A diagram, which 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, and other neutral players (see figure 1). This includes component relationships, other pattern components, attributes, and the functionality of the pattern's components. Take a look at figure 4 as an example.

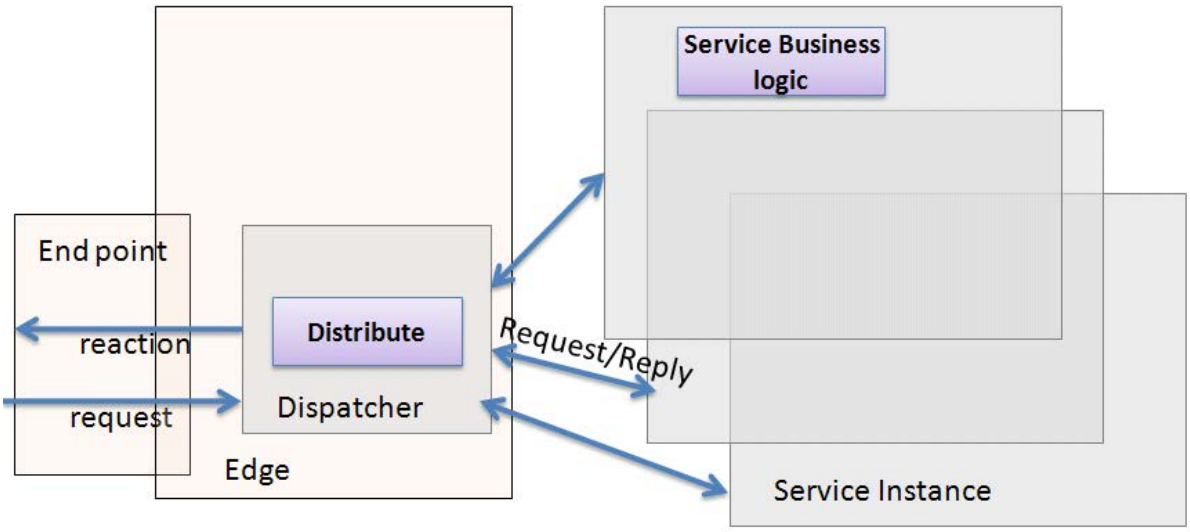


Figure 4 Sample pattern diagram: the *Service Instance* pattern. *Endpoint* and *Edge* are two neutral components (not part of the pattern).

Without getting into the details of the roles of the different components, looking in this diagram we can see that the edge and endpoints are neutral components that are not a part of the pattern. The dispatcher and the service instance are two components that are a part of the pattern. Each of the pattern's parts has one or more roles and attributes. In this case, we can see the dispatcher is responsible for distribution (of messages) and that the service instance is responsible for (running) the service business logic. *Dispatcher* and *Service Instance* are part of the pattern, while the purplish 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. For instance, requests and replies are passed back and forth between *Dispatcher* and *Service Instance*.

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 on the implications or consequences of applying the pattern as well as the relationship to other patterns and examples.

TECHNOLOGY MAPPING

This 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 actually 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 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. In figure 5, we see the various inputs the architect can use before a solution is designed. 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.

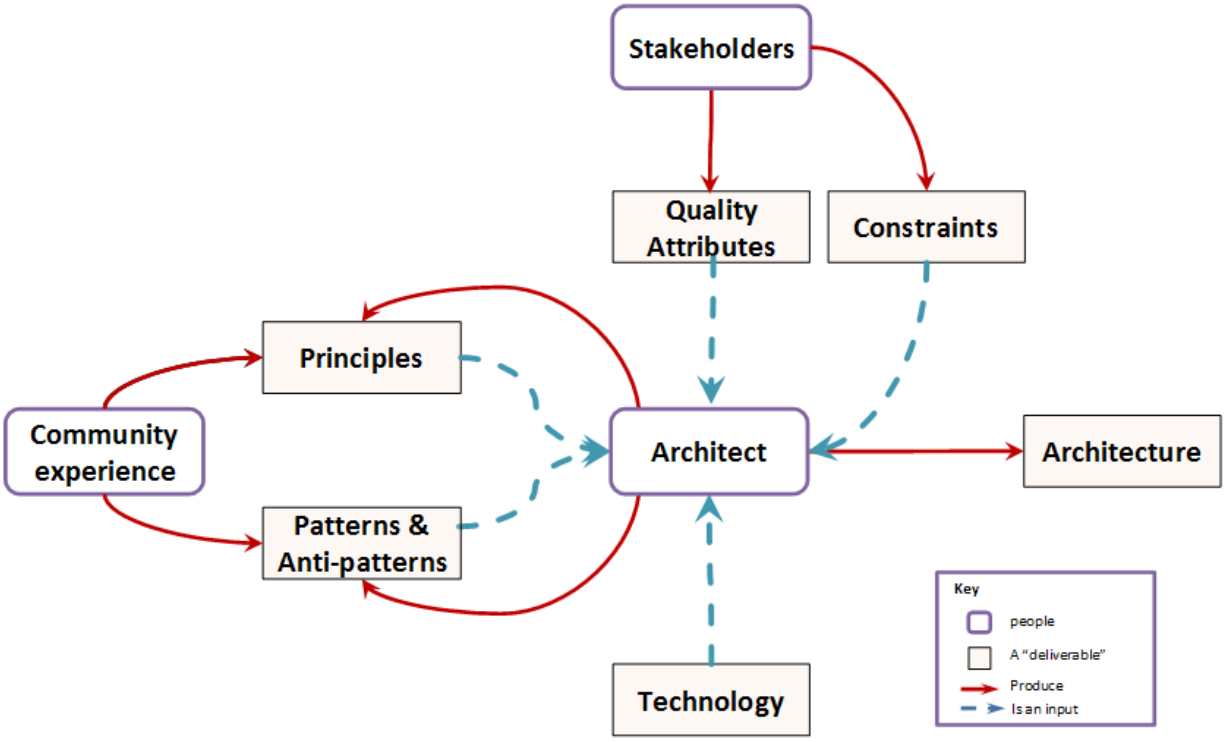


Figure 5 The architect uses various inputs to design the architecture.

First and foremost, you work with the constraints and requirements gathered from the stakeholders. These include, requirements for performance, security, scalability, interoperability, and so on. You can augment these inputs by drawing on personal and community experience to add principles, patterns, and anti-patterns. 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.

From isolated patterns to a pattern language

Each pattern on its own provides some useful information and describes a good practice. As mentioned, patterns have relationships to other patterns. For instance, 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*.

Evolving patterns into a pattern language, which shows the patterns' relationships, helps to increase the ability 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 the needed visibility you may not have had before.

Table 3 shows how the patterns are categorized.

Table 3 List of pattern categories

Category	Subcategory	Description
Service structure	Basic	Common service building blocks
	Performance, availability, and scalability	Patterns to solve scalability, availability, and performance challenges
	Security and management	Patterns for securing and managing services
Service consumer interaction	UI interaction	Interaction patterns when the consumer is a user client
	Service Interaction	Interaction patterns when the consumers are other services
Composition patterns		Making services work together and share information

Summary

This paper laid the foundation needed to understand the proposed SOA patterns and their overall context. We began with a definition of SOA and patterns in general and how patterns can be used to provide solutions to SOA challenges. The definition was followed by a short discussion on the technical and business benefits of SOA. The second part of this paper explained patterns, each pattern's structure, and provided a list of categories and descriptions.

This chapter covered a lot of issues very briefly. The goal was to create a common vocabulary for our discussion on SOA patterns. If you're interested in learning more on some of the issues discussed here, look at one or more of the resources listed in table 4.

Table 4 resources for further reading on topics covered in this chapter.

Area	Resource name	Why
Distributed systems	*IT Architecture & Middleware: Strategies for building large and integrated systems* by Chris Britton	Provides a good look at the history of distributed systems and the inherent difficulties that they inflict. It is a very thorough book; the only problem is that it ends just before the SOA era.
Fallacies of distributed computing	"Fallacies of Distributed Computing Explained" (http://www.rgoarchitects.com/Files/fallacies.pdf)	SOA is an architectural style for distributed systems. Most other styles don't have a "distributed mindshare" and they dis my paper that explains how the fallacies are still relevant today,
SOA	*Enterprise SOA: Service Oriented Architecture Best Practices* by Dirk Krafzig et. al.	One of the best books on SOA. Provides a very good introduction on the subject.
SOA	*Service-Oriented Architecture (SOA): A Planning and Implementation Guide for Business and Technology* by Eric A. Marks, Michael Bell	Takes a look at the business perspectives of SOA and provides a completely different (and complementary) angle at SOA.

Here are some other Manning titles you might be interested in:

Open Source SOA
Jeff Davis

SOA Governance in Action
Arnon Rotem-Gal-Oz, Eric Bruno, and Udi Dahan

SOA Security
Ramarao Kanneganti and Prasad A. Chodavarapu

Last updated: November 20, 2011