SOA Made Simple: Service Design

Ronald van Luttikhuizen
Vennster
Utrecht, The Netherlands

Keywords:

SOA, Services, architecture, service design, design, service identification, identification, design guidelines

Introduction

This manuscript and accompanying presentation provides a summary of the Service Design chapter of the upcoming SOA Made Simple book by Lonneke Dikmans and Ronald van Luttikhuizen; see http://www.packtpub.com/service-oriented-architecture-made-simple/book.

Services are a key aspect of any Service-Oriented Architecture (SOA). Once you have identified what services your clients require, you can start designing these services by designing their operations, and the input and output of these operations. Service design principles indicate what qualities a service needs to have in order to be a usable building block in the architecture you are trying to achieve. When services are poorly designed or poorly implemented, your solution architecture will probably have little value for the business as well. What we need are sound design principles that help us as a service provider to create (re)useable services, and help us as service consumer to judge if the services that we use (or want to use) are well designed ones.

This manuscript and accompanying presentation includes several service design principles and quality-of-service aspects that can be used as checklist when creating, buying, or reviewing services. Such design principles include isolation and idempotency. Most important is that services need to be easy to use, must provide value, and that they can be trusted by (future) consumers.

What is a Service?

A service can be described as something useful that a provider does for a consumer. A service is an economic term to describe the goods and services that organizations and people produce, sell to one another, and buy from each other. Services are nothing new and have been around as long as mankind has been. Consider a land worker in Medieval England who harvests the crop (the service) in exchange for a place to live.

What is new nowadays is using the notion of services in the domain of (enterprise) architecture and IT. OASIS describes a service as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description."

OASIS (Organization for the Advancement of Structured Information Standards) is a not-for-profit consortium that drives the development, convergence and adoption of open standards for the global information society. See: http://www.oasis-open.org/org.

Contract, Interface, and Implementation

For every service you need to define certain characteristics in order for the services to be well-defined and usable. If the service is not well-defined it might not be clear for all consumers what added-value the service exactly offers, how much it costs to use it, or how to use the service at all. This will lead to customer confusion and can even cause consumers not to use the service at all. Three important characteristics of services are:

- Contract;
- Interface;
- Implementation.

A contract specifies what consumers can expect from a service based on their predicted needs, and what a service provider needs to offer. The interface defines how we can make use of the service and access it, while the implementation is about the realization of the service. The contract and interface are visible to the outside world. The implementation is more of a "hidden feature" or "black-box" to consumers. Consumers generally don't care about its implementation.

We can think of it in the following way. Suppose you go to a diner to have breakfast. The diner offers a menu that lists all their services. One of the services offered by the diner is the diner's famous classic breakfast consisting of toast with eggs and some side dishes.

- Contract. Quality, price, availability, order time, and so on are all examples of important aspects of the breakfast that we care about as consumers. While the diner will probably not define a formal contract that both we and the diner sign, it can be considered as the contract of the breakfast service nonetheless.
- Interface. The interface is the way you interact with a service. To order breakfast you among others interact with Jane, our waitress, and the menu which is in English and Spanish. These are aspects of the interface that are important for (quite literally) consuming our service.
- Implementation. We might not be that interested in the implementation of the breakfast; you probably do not need to know how the diner prepares their meals. Whether the chef is named John, bought all the ingredients himself and uses a Japanese stainless-steel knife, or whether the ingredients were delivered by a wholesaler, the chef is named Maria and uses a German knife does not really matter to you, as long as the quality is as expected.

Services in the domain of (enterprise) architecture and IT are also composed of a contract (e.g. SLA), interface (e.g. WSDL and XSD), and implementation (e.g. Java).

Service Identification

The next step is to find out what services you actually need based on the requirements of clients. This process is called service identification. Once you have identified what services your clients require, you can start designing these services by designing their operations, and the input and output of these operations.

It's important to start with service identification before going into the actual details of service design. Don't start with the design until you have a business case and a real project with real consumers, otherwise you might end up investing a lot of time and money in the design and implementation of services and operations that are useless. If you are not sure whether particular operations are needed,

leave them out and only design and implement the ones you need. Later on you can always expand services and add operations, should the need for them arise.

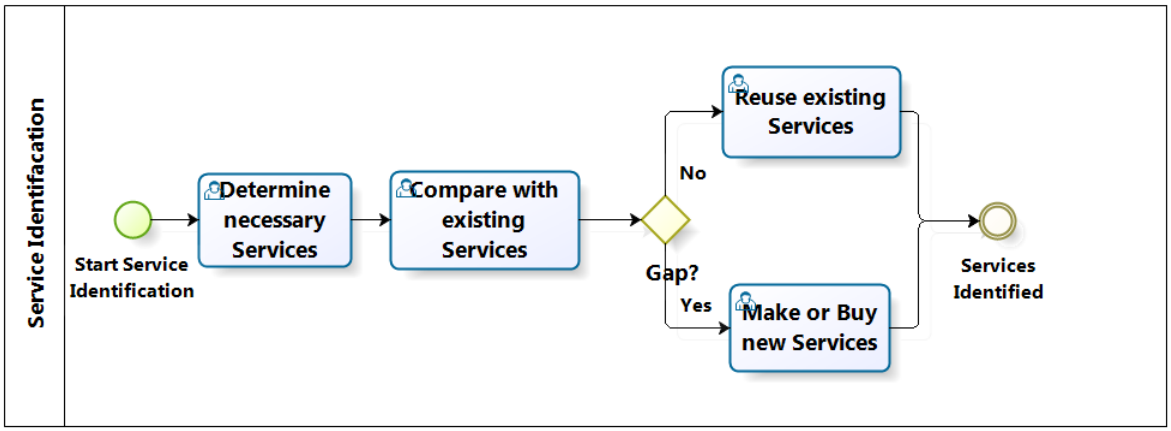
Service identification is not only about finding services that are already there, but also about identifying services that your organization needs but do not yet exist. These services can then be bought or developed. Identification can also result in the need to modify existing services, for example adding additional operations, or 'promoting' existing IT assets into services. The latter is the case when the implementation is already there, but not yet exposed as a service: the service interface and possibly the contract are missing.

Service identification is an iterative process, meaning that you can't identify all the services in one large project all at once. Start small and realize the initially identified services before gradually expanding your service landscape in subsequent iterations.

There are generally two approaches to identifying services: top-down and bottom-up. Top-down service identification typically happens when executing enterprise architecture or solution architecture activities. Bottom up service identification occurs when you investigate your existing IT assets, and derive services from there. Services that are identified in a bottom-up fashion are not less valuable than top-down identified services. Top-down services have the risk of being too abstract so nobody wants them or knows how to use them. Bottom-up services are there because there is a concrete need for them, but might need to be modified before they are usable for other departments or IT systems in the organization. So, do we need to follow a top-down or bottom-up approach? As with many things the truth is in the middle: we need to do both. The 'roadmap' chapter in the book covers this subject by explaining how to set up a roadmap for the realization of your SOA that includes service identification.

Gap-analysis based on Service Identification

Analyzing the differences between two scenarios or plans, such as the as-is and to-be architecture is called a gap-analysis. In other words you analyze what the differences, or 'gaps', between two scenarios are. When the differences between the as-is and to-be architecture are clear, you can identify the concrete activities that need to be executed to 'bridge the gap' between the as-is and to-be architecture. These activities are then planned in a roadmap based on priority, costs, benefits, and so on, and are executed.



Service Design

Service design principles indicate what qualities a service needs to have in order to be a usable building block in the architecture we are trying to achieve. When services are poorly designed or poorly implemented, our solution architecture will probably have little value for the business as well. What we need are sound design principles that help us as a service provider to create (re)useable services, and help us as service consumer to judge if the services that we use (or want to use) are well designed.

The following list includes several service design principles and quality-of-service aspects that can be used as checklist when creating, buying, or reviewing services. Note that the list is not in a particular order; the importance of each principle depends on the needs of your organization.

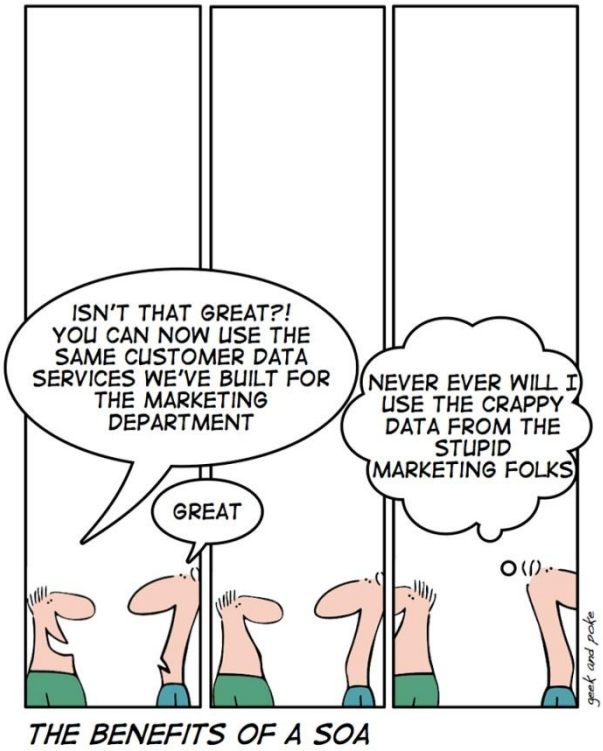
Provide value
For every service, consider if and why you need it. If a service doesn't provide value to someone or something (organization's clients, internal departments, other IT systems, etc.) then it is probably not a good service, or only part of a service and not a service in itself.

Meaningful
It should be easy for (future) consumers to use a service. Therefor the service interface needs to be meaningful to the consumer and not too abstract or complex. If a service is not meaningful, the required effort to consume a service will increase. Consumers will not be able or are reluctant to use such services since they don't understand them or it is too expensive to use and integrate them into their landscape.

Implementation hiding
When it comes to services, consumers don't care about the actual implementation behind the service; this is a black box for them. Consumers focus on the contract and interface of a service to decide whether to use it and to be able to actually consume it. In short, a service and especially its interface and contract, should be self-describing and understandable.

Hiding implementation details is a common approach in several programming paradigms, even more so for SOA in which we specifically differentiate between contract, interface, and implementation. The service interface should abstract away (or hide) the specifics of the underlying systems and organizations that do the actual work. This makes it easier to change, upgrade, or swap the implementation without breaking interoperability since the interface can stay the same. It also doesn't burden consumers who don't need to know about the specifics of the implementation.

Trust
Consumers need to have faith in the service they consume. Consumers want services to be accurate: think about the weather channel, you want it to be accurate so you can trust the forecast. Services should do what they are supposed to do, especially in a SOA where services can be outside your own span-of-control in case of external services. Think about aspects such as security, fault handling, and auditing to increase trust in your services.



THE BENEFITS OF A SOA

Source: http://geekandpoke.typepad.com

Idempotent
A service should be predictable; invoking a service operation with the same input more than once should result in the same outcome. Every time you order a soda in a bar, you expect a soda. You don't want sparkling water if somebody else ordered a soda before you, or a beer in case it rains outside.

Isolated
Services only provide flexibility and can only be easily changed if their operations are independent of other operations within the same or another service; this is called isolation. If a change to an operation results in changes to several other operations that are tightly coupled to the originally changed operation, we lose flexibility. Operations need to be separate building blocks that provide capabilities themselves.

Interoperable
Services should be easy to integrate into our IT landscape. Interoperability is a measure for the amount of effort it takes to use and invoke services. Interoperability is achieved by using standards for describing, providing, and accessing services such as XML, WS-*, WADL, and WSDL. The use of standards increases the ease of consuming services since most platforms and vendors adhere to standards. Note that only the service interface needs to be interoperable; the implementation itself can be proprietary. Think about the breakfast example again: as long as you can order breakfast in the language you as a consumer speak, you don't care what language the cooks speak, as long as the breakfast is good! Using standards to provide and access services helps to mix and match different technologies such as PL/SQL, .NET, Java, and PHP.

Interoperability is not equally important in all SOA environments; it depends on several factors how important operability is for your organization. Interoperability is especially important in the following cases:

- You need to integrate services in a heterogeneous environment in which different technologies and platforms are used.
- Your services are used outside your own organization and you have no control or knowledge over the used platforms by the consumers.
- You want to integrate services with packaged software or offer services as part of packaged software.

Interoperability is less of a concern when you apply SOA within your own organization consisting of a homogeneous environment in which you have standardized on one or a few (possibly proprietary) platforms and technologies.

The principles of isolation, trust, and idempotency can be tricky and difficult to apply and are explained in more detail in the book. We conclude service design by looking at granularity and reusability since these aspects build upon the service design principles that are introduced here.

Granularity
So far we have considered all services to be of equal 'weight' or relevance. However this is not the case in a real-life SOA. A service for converting AM/PM time notations into 24hr time notations isn't comparable to a service that is used to book a holiday including hotel reservations, car rentals, and flight bookings in terms of added value. In other words, services are of different importance based on the degree of value or functionality they add. This is called granularity. When thinking about granularity, the million dollar question we need to answer is: 'How granular should a service and its operations be?' In any case a service should be:

- Big enough to provide value on its own.
- Small enough to be able to change it, without changing the whole IT landscape in your department or organization. Putting all ERP-related functionality in one big monolithic service with thousands of operations that are used by almost everyone is too big. Every change in ERP-related functionality requires a change in this one service.
- Granularity should be derived from functionality: don't mix functionalities that are unrelated into a single service, and don't split functionality that belongs together into different services.
- Granularity should go hand in hand with transaction boundaries; an operation defines a transactional unit: all activities as part of an operation are either, successfully executed and committed together, or, in case an operation fails, all activities as part of the operation are rolled back.

We need a pragmatic scheme to categorize our services into a set of different types of granularity. One way to do this is by means of a service classification. The book provides an example of a service classification.

Reusability
Reusability means a service is used by more than one consumer or is meant to be used by more than one consumer in the future. In the latter case the service is designed with reuse in mind, but not yet used by more than one consumer at the time being.

Reusability should not be confused with ease of use or usability of services. Reuse can save money. Instead of buying or building a new service, we can use existing ones; hopefully without too many modifications to the service we want to reuse. Measure the benefits of reuse not only by the current number of consumers but also the possible future consumers that don't use the service yet but are viable candidates for using it. Reuse can also speed up the creation of new artifacts (processes, products, and so on) since we assemble already existing services instead of building them from scratch.

However, there is a price to pay: the more a service is reused the less flexible it often becomes. An increase in the number of consumers results in more wishes and requirements that need to be agreed upon by all stakeholders. Reusability isn't a goal on its own and not every service is meant to be reusable. It is a pitfall to make services more generic per se to increase their reusability. It makes a service harder to use if it is too generic. First investigate if (future) reusability is necessary, before you end up with over over-generic and non-understandable services and operations.

Contact address:

Name
Vennster
Postbus 31457
6503 CL, Nijmegen, the Netherlands

Phone:	+31 (0)6 52456043
Email	ronald.van.luttikhuizen@vennster.nl
Internet:	vennster.nl
Twitter:	rluttikhuizen
LinkedIn:	http://nl.linkedin.com/in/soaarchitecture