

The SOA Magazine

Feature Article



Ten Ways to Identify Services

by Jan-Willem Hubbers, Art Ligthart, Linda Terlouw

Published: December 10, 2007

(SOA Magazine Issue XIII: December 2007, Copyright © 2007)

[Download this article as a PDF document.](#)

Abstract: SOA is increasingly becoming an unavoidable part of project delivery for many organizations. It is therefore high time that practitioners avoid the dangerous practice of creating ad-hoc services and begin following proven industry principles and methods. In this article we discuss ten approaches for identifying services. The intention is for this list to become an effective starting point for service definition, alleviating project teams from the longwinded discussions about granularity, feeling, intuition and craftsmanship that have historically been part of initial service delivery phases. This collection of methods is the result of research. Therefore, it is important to understand that even though some have become established industry practices, there are still pitfalls and trade-offs that need to be taken into consideration.

Introduction: Service Fundamentals

In a service-oriented IT landscape that has been shaped according to the principles of service-orientation, functionality and data are provided by means of services. These services can be defined in a uniform fashion based on international standards (XML, SOAP, WS-*), or they may be based on more traditional and proprietary mediums.

Modern integration technologies, such as the ESB, enable consumption of services regardless of location, platform or programming language. By orchestrating services in the desired order, business processes can be optimally supported with functionality and data from systems both internal and external to the IT enterprise.

Along with the rise in popularity of SOA began a community-wide discussion about how to best identify services. When is a service "too big" or "too small", when is it "too specific" or "too generic", or exactly "right"? Before getting into the most common approaches we need to establish some fundamentals.

First and foremost, generally accepted architectural principles for services exist. These design principles should always play a guiding role when identifying services. The eight established service-orientation principles [REF-1] are: Standardized Service Contract, Service Loose Coupling, Service Abstraction, Service Reusability, Service Autonomy, Service Statelessness, Service Discoverability and Service Composability.

The golden rule to successful service identification is that services should adhere to these principles.

Secondly, there are various ways of typing services using service classifications or service models. Examples include presentation services, process services, business services, application services and data services [REF-2]. Naturally there are approaches that are particularly suitable for each type of service. In this article we primarily focus on methods for business and application-centric services. When desired, these two service types can be further subdivided (e.g. into create, read, update, delete, transform, generate, select, value, validation and calculation services [REF-3]).

Finally, what determines if a service is well-defined depends on perspective: an administrator will have different requirements than a process designer or a tester. Ultimately, the success of a service is measured by the value it

provides to the organization over time.

Ten Common Methods for Service Identification and Definition

Even though these methods are listed individually, in practice they can be combined to shape unique approaches and methodologies.

Method 1: Business Process Decomposition

One of the most common approaches for identifying and deriving services uses business processes as a starting point. The business process is subdivided into sub-processes or decomposed into granular activities and tasks. The lowest level tasks can consist of small, cohesive "logical units of work" that are supported by the functionality offered by distinct services. This results in services that are very "demand-driven".

A great benefit of this approach is that the resulting services have a guaranteed fit with an organization's functional needs. This method is also very intuitive, allowing project teams to use it for proof-of-concepts and pilot projects.

The strong focus on the demand side can have its challenges. A (too large) gap between business process and applications may result if the services are only modeled according to business process definition specifications and without taking implementation considerations into account.

In addition, unless the modeling effort involves iterating through multiple business processes, services can be tailored too specifically to the tasks and activities of one business process (resulting in services that may not be reusable).

Even when deriving services from multiple processes, several activities might require similar functions. On-going coordination is required to avoid unintended redundancy across services defined by project teams working in parallel.

Method 2: Business Functions

As just mentioned, a crucial point in the identification of services based on a single business process is the tight coupling between the process and these services. This is in stark contrast to the underlying idea that services should provide the means to decouple business logic from IT facilities.

A possible solution to this problem is to start from a business function model. This abstracts from the way the business processes have been implemented. Analogous to the business process approach a step-wise refinement towards services is used. In this case, however, the most detailed business functions in the functional decomposition are translated into services.

Just like the process-based approach, the function-based method is demand-driven and carries the same risks. Using business functions as a starting point mitigates the risk of redundancy of services and functional overlap is eliminated via the business function model.

Method 3: Business Entity Objects

The purpose behind most services is to process business information. By modeling services according to business object models, business entity-based services can be identified commonly requiring CRUD-type functions. This approach relies on the use of canonical data models (CDMs) that standardize information exchanged between services. Therefore, this approach can be considered supply-driven.

Canonical services rely on technology resources that use their own particular data models. Data consistency is achieved by mapping between the applications' data models and the CDM. The data elements that comprise a single CDM object can hence be managed in different applications. Nearly all current ESB products offer support for CDMs. The real challenge lies in achieving consensus with regard to the exact definitions of common objects.

A strong point of this approach is that the semantics of services receive attention in early modeling stages, thereby reducing the amount of undesirable design changes required when projects get closer to production phases.

The main pitfall of this method is the need for standardized data models. Depending on the scope of the SOA project, this requirement can result in "analysis paralysis", despite the fact that only the business objects that play a role in

exchanging data need to be modeled. A domain-based roll-out of services can help overcome concerns about having to establish global data models.

Method 4: Ownership and Responsibility

This approach is not a "true" method, but it is recommended for making choices about which services should be offered. SOA requires a well-defined structure for decision-making, where roles and responsibilities for processes and services are clearly allocated and assigned. When identifying services, the party that carries the responsibility to make available the required functionality determines which services will ultimately be offered.

Naturally methodical design approaches play a role in this process, but there are several other concerns that must be taken into account when settling on the final choice: development costs, maintenance costs, lifecycle management of underlying applications and platforms, priorities, availability of human resources, and so on.

The biggest advantage is that it is always clear who owns a service. In other methods this issue can become a point of debate and can even result in political consequences. On the downside, services from different domains may end up overlapping because of their required ownership structure. Additionally, the organization must have a functioning governance platform in place, which implies a level of maturity that is not yet commonplace.

Method 5: Goal-Driven

With this approach a project team decomposes a company's goals down to the level of services. In this context, a service is regarded as a goal that can be executed through automated support [REF-5]. For example, a goal such as "increase customer retention" can result in a service called "register customer for loyalty program".

The advantage is the strong relationship forged between services and company strategy. However, there are two distinct problems to this method: goals tend to be subjective, and a fair amount of IT cannot be directly aligned to business goals.

Subjectivity may well cause two business goals to be decomposed into two distinct services, even though the desired functionality is identical (which means that using a single service would have been preferable). Also, because many IT capabilities cannot be directly related to business goals, there is a constant risk that many potentially useful services will simply be overlooked.

Method 6: Component-Based

The essence of using components is to divide IT-functionality into units with maximal internal cohesion and minimal external coupling. Components are truly self-contained units of functionality. Various methods to identify components have already been introduced in the realm of component-based development. A guiding principle in these approaches is that each component has exactly one owner and that the responsibilities of each component have to be defined as precisely as possible. These responsibilities can be used as a starting point for identifying services.

In theory, component-based development results in a functionally organized IT enterprise. Components can be custom-coded or purchased off-the-shelf. Additionally, a need arises to compose services offered by components into composite services. Currently suppliers of large monolithic applications (such as ERP and CRM systems) tend to organize their applications in a more modular fashion and to make them available through services. These modules correspond roughly with components.

The benefit to basing services on components is that the service identification process is greatly simplified. The bulk of the analysis work has already been carried out as part of the component-based development method. However, in reality, this can lead to several problems. Modern-day services and traditional components rarely share the same goals, requirements, and expectations. Creating a series of fine-grained services that mirror underlying components can severely inhibit an SOA initiative from attaining strategic goals that were never taken into consideration when the components were first designed.

Method 7: Existing Supply (Bottom-Up)

A pragmatic way of quickly defining services is to base them on immediate requirements for information and

functionality. In this case, the starting point is the functionality provided by existing legacy applications. The systems that provide the bulk of the automated support required by the primary business processes are selected. Using tools and wizards the existing interfaces, APIs, screens, transactions, queries and tables are made accessible through services.

This classic bottom-up approach is supply-driven by nature, and does not focus on reusability (or even usability!) of the identified services. Hence it is commonly necessary to cluster the functionality and remove functional overlaps by combining similar services into a single (often monolithic) service.

The main advantage of bottom-up delivery is that it requires little time to reach a first definition of services. It is an appropriate approach if the functionality of the existing applications is urgently required and perhaps also sufficient to support both current and future business processes. A potentially positive side-effect of this method is that it can be used in a context where little process or function models are available.

However, ultimately this is not a recommended approach for defining services in support of service-orientation. The Law of Conservation of Challenges will rear its ugly head in that badly designed applications that have been adapted to changing circumstances many times over (and are tightly coupled to the business processes) will make it very difficult to design reusable and future-proof services. In the end, this approach almost always leads to the creation of new application silos. It just happens that in this case, the silos are themselves services.

Method 8: Front-Office Application Usage Analysis

Charting the current demand for information and functionality is a pragmatic way of identifying services quickly. In this approach, a set of applications is selected that support the majority of the primary business processes. The functionality, as offered by the back-office applications, is then surveyed by making a list of the queries and transactions used by the set of front-office applications. These functions are clustered, redundancy is removed, and finally an optimization step combines comparable functions into a single service.

The most obvious advantage is that services can (again) be quickly identified. In a way, the applications provide a view on the business function model and the services that are found using this method can score well on immediate reusability, as long as clustering and optimization steps are properly carried out. Another side-effect is that this approach can be utilized in a context that lacks usable process or function models.

However, at the end of the day, the Law of Conservation of Challenges applies once more. Applications that suffer from bad design or that are tightly coupled to the current implementation of the business process can pose severe challenges to designing quality services that are expected to remain reusable on an on-going basis. This approach should really only be considered when you have a great deal of confidence in the quality of existing application designs.

Method 9: Infrastructure

Platform independence might well be an accepted architectural principle for services, but composite services in particular demand extra attention. This method acknowledges that services cannot always be identified independently of the technical infrastructure that is being used.

How convenient is it when a service composes two utility-centric services that run on separate platforms (e.g. a mainframe and a Unix machine)? Consider the required connectivity, execution and potential rollback of transactions, variations in availability, security and monitoring, network traffic, etc.

Note that even though it is nearly always technically possible to solve these issues (using modern middleware), a cost-benefit analysis might indicate that an alternative solution is called for. Non-functional requirements also play a part in this analysis (see Method 10).

When discussing the advantages of this approach we can be brief: it should only be used when absolutely necessary. The core idea of service-orientation is to hide and abstract the underlying application environment (and especially its supporting infrastructure layer!).

Method 10: Non-Functional Requirements

This method uses non-functional requirements as the primary input to identify services. It is not a "method" per se, but more like a set of techniques that can be used on top of other methods.

After a preliminary set of services has been identified using one of the other service identification approaches, the (future) service provider verifies the feasibility of the non-functional requirements of the (future) service consumer(s). If one or more of the non-functional requirements are deemed infeasible, but still more important than the design principles of the identification method, the services may be redesigned.

Two common non-functional requirements that can play a role here are security and performance. Consider the following example: to realise Service X, functionality is needed from Applications A and B. If Application A conforms to the security requirements imposed on Service X and Application B cannot, it may make sense to split Service X into separate secured and non-secured services.

Alternatively, if an organization chooses to implement SOA by means of Web services, some services may not be able to deliver the required performance. Choices that can be made are the merging of multiple services (reducing the amount of inter-service communication requirements) or deviating from Web services standards for that particular service.

Something to keep in mind is that if performance considerations necessitate redesigning a large number of services, further analysis is required to determine if the current IT architecture is suitable for SOA in general (or only for this part of the IT landscape).

Assessing Methods with the Service Pattern Language

Regardless of the approach you choose to identify and define services, it is essential that you understand the fundamental theory behind service design in support of service-orientation.

The Basic Service Design Pattern Language [REF-4] establishes the foundation for service identification, definition, and design by providing seven basic design patterns that form a fundamental service pattern language. This pattern language can be considered a primitive process that addresses only the most necessary steps for creating services.

You can trace all ten of the methods explored in this article back to this fundamental process. Of course, each method has its own priorities and trade-offs that can affect the extent to which any given service design pattern is supported. But by understanding this basic design pattern language, you can better evaluate these methods as to how well they support service-orientation in general.

General Pitfalls

As with the pursuit of anything worthwhile, the road to attaining a good service portfolio is lined with pitfalls. Here are some common examples that apply to service identification:

- *Services in Name Only* - The terms "SOA" and "service" are used rather loosely in many IT environments. Project teams may choose to label their applications as "service-oriented" simply because it sounds more cutting edge or due to the common misperception that the use of Web services alone constitutes an SOA. Either way, when it comes to implementing the "services" in these types of initiatives, the programmers tend to run the show. They create a plethora of (mostly technology-centric) Web services, disregarding business/IT-alignment, reusability or any other properties a service should have. The end-result is an application that uses Web services but is not itself service-oriented. Ultimately, this pitfall leads to great disappointment when the expected benefits of SOA are never realized.
- *Perfect Non-Existent Services* - On the other side of the spectrum lies the danger of having analysts and architects model wonderful services that simply cannot be built using today's technologies - or - they can only be realized via murderous costs. This pitfall can be avoided by constantly ensuring that all modeling efforts are balanced with a dose of reality.
- *And Never Shall They Meet Services* - When different project teams within the same organization commit to radically different service definition and delivery methods (such as the opposing top-down and bottom-up approaches), collections of services can be created that will simply never be compatible. These can form natural silos that will eventually impose significant integration effort when cross-silo communication is required.

- *Babel Services* - If an organization does not have a canonical data model (and therefore the definition of the service semantics is not clear), the services are automatically incompatible. The result is an environment that will depend on transformation and bridging technologies for many, many years to come. This will ultimately inhibit every aspect of service-orientation.
- *Spaghetti Services* - A problem that can occur when services are been defined on multiple levels of granularity is that technical terminology and business terminology can get so mixed up that the services themselves can become unintuitive, confusing, and sometimes just unusable.

There are some simple rules to avoiding all of these pitfalls:

1. Adhere to the principles of service-orientation. These are essential and fundamental to creating well-defined services that support the strategic goals of SOA.
2. Understand your options when it comes to service identification and definition approaches by studying the methods covered in this article.
3. Measure the effectiveness of service identification approaches you are considering by mapping proposed service design processes to the fundamental service design pattern language.

Conclusion

Using proven methods to tackle the issue of creating well-defined services is certainly recommended. It allows you to leverage the experience of those that have already been through this process many times. However, no one method is perfect. Each has its own benefits and trade-offs. It's always in your best interest to take proven methods as a starting point and then consider how they can be optimized in support of realizing the requirements that are specific to your SOA goals.

References

- [REF-1] "SOA: Principles of Service Design" by Thomas Erl (Prentice Hall, 2007)
[REF-2] "Enterprise SOA" by D. Krafzig et al (Prentice Hall, 2005)
[REF-3] "SOA, een praktische leidraad voor invoering: Socrates™" by A. Ligthart et al (SDU, 2005)
[REF-4] "SOA Design Patterns" by Thomas Erl (public review of unpublished draft at www.soapatterns.org, Prentice Hall 2007/2008)
[REF-5] "A Goal-driven Approach to Enterprise Component Identification and Specification" by K. Levi and A. Arsanjani (Communications of the ACM, 2002)

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About](#) [Legal](#)

Copyright © 2006-2008
SOA Systems Inc.

