zapthink
white paper

# How to Define a Business Service

## The Art & Science of Service Granularity

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

# How to Define a Business Service

## *The Art & Science of Service Granularity*

November 2007

*Analyst: Jason Bloomberg*

**Abstract**

How to define a Business service is a pervasive and critical question that is essential to the success of any Service Oriented Architecture (SOA) initiative. SOA projects can consist of too many moving parts causing confusion on what to use, reuse, or leverage. It then becomes the challenge of the architect to reduce this confusion by identifying the proper best practices for defining Services properly to meet the business goals set out for them.

Such practices include a variety of approaches. Architects can define a Service:

- in the context of business processes
- in the context of existing assets leveraging the business, information that the Service is expected to send or receive
- in an iterative fashion that delivers business value and decreases risk.

Combined with SOA governance processes that alleviate the challenges with managing too many moving parts, the architect will have the necessary best practices to manage the art and science of Service definition and accelerate SOA adoption.

All Contents Copyright © 2007 ZapThink, LLC. All rights reserved. The information contained herein has been obtained from sources believed to be reliable. ZapThink disclaims all warranties as to the accuracy, completeness or adequacy of such information. ZapThink shall have no liability for errors, omissions or inadequacies in the information contained herein or for interpretations thereof. The reader assumes sole responsibility for the selection of these materials to achieve its intended results. The opinions expressed herein are subject to change without notice. All trademarks, service marks, and trade names are trademarked by their respective owners and ZapThink makes no claims to these names. LEGO® is a trademark of the LEGO Group of companies which does not sponsor, authorize or endorse this content.

# I. Solving the Service Granularity Challenge

Since Service-Oriented Architecture (SOA) is an architectural approach to organizing IT resources to meet the changing needs of the business, there will never be a simple recipe for how to implement SOA properly. There are simply too many variables—the cultural and political environment, the existing technology infrastructure, the business challenges the organization faces, and more. And yet, architects persist in looking for the secret recipe, the perfect combination of technology and best practices that leads to perfect Services.

There will never be such a recipe, of course. Instead, there are a broad set of best practices currently available that detail how to go about building Services in the context of SOA. Such practices are often pragmatic, in that they focus on results more than formality, and heuristic, in that the practices don't assume a particular result, but rather advise the architect as they proceed. A central example of this pragmatic, heuristic best-practice approach to Services is how to build Services at the proper level of granularity.

*Granularity is a relative measure of how broad a required piece of functionality must be in order to address the need at hand.*

*Granularity* is a relative measure of how broad a required piece of functionality must be in order to address the need at hand. Fine-grained Services address small units of functionality or exchange small amounts of data. Consequently, to build complex business processes, companies would have to orchestrate large numbers of such Services to effectively automate the process: a difficult, often Herculean task.

Coarse-grained Services, however, encapsulate larger chunks of capability within a single abstracted interface, reducing the number of Service requests necessary to accomplish a task, but on the downside, they might return excessive quantities of data, be difficult to reuse, or it might be difficult to change them to meet new requirements. As a result, an architect must craft the right balance of fine-grained and coarse-grained Services to meet the ongoing needs of the business. This balance, of course, is part of the art of SOA.

### What Constitutes a Well-Defined Service?

The first topic to consider when trying to understand how to craft Services of the right granularity is to understand how well-defined a Service interface must be. Fundamentally, a well-defined interface is a contracted interface that a computer can understand. When the Services are Web Services, this interface definition depends upon open standards. Other Services go beyond Web Services and leverage internally defined formats within the enterprise. Either way, it doesn't matter which specific standards you use to define a Service interface, as long as the interface has a contract that provides enough information for a loosely-coupled exchange.

While it is desirable for contract metadata to be in the form of machine-processable standards, simply having the specifications available for the user

Thank you for reading ZapThink research! ZapThink is an IT advisory and analysis firm that provides trusted advice and critical insight into the architectural and organizational changes brought about by the movement to XML, Web Services, and Service Orientation. We provide our three target audiences of IT vendors, service providers and end-users a clear roadmap for standards-based, loosely coupled distributed computing – a vision of IT meeting the needs of the agile business.

Earn rewards for reading ZapThink research! Visit www.zapthink.com/credit and enter the code **BIZSVC.** We'll reward you with ZapCredits that you can use to obtain free research, ZapGear, and more! For more information about ZapThink products and services, please call us at +1-781-207-0203, or drop us an email at info@zapthink.com.

Copyright © 2007, ZapThink, LLC 

does not give any clues on how to define the Service interfaces and address the granularity challenge at hand. It is important for a computer to be able to understand a given Service, but that doesn't make that Service valuable. Indeed, other forms of architecture have also depended on standards-based interfaces, and yet they have not produced the sort of loosely-coupled Services enterprises desire.

One prerequisite to such loose coupling is that Service contracts must contain both functional and non-functional requirements that specify the expectations of both the Service consumer and provider. As such, at the very least, a Service contract must provide unambiguous information about what the Service does. In other words, a Service should clearly say what it means and mean what it says. Users shouldn't be left scratching their heads as to what will happen when they provide the required input to a Service. So, a better argument for the well-defined Service question is that a Service is well-defined not only if a computer can understand it, but also if it is unambiguous as to what the Service will provide. Basically, a human can also understand the Service contract without having to consult additional resources or information.

### The Role of Process Decomposition

*Services are not fundamentally a technology concept, but rather an abstract representation of value the business wants to extract from its technology.*

Creating a Service contract that adequately describes a Service is an important part of the SOA story, but more important is how organizations compose Services to implement flexible business processes. It's important to remember that the notion of Services is not fundamentally a technology concept, but rather an abstract representation of value the business wants to extract from its technology. As such, it's important to focus on defining Services from the business point of view, that is to say, the business process point of view, since business processes fundamentally define the business.

One approach to specifying the right Services is to start with some business process and decompose it into increasingly smaller subprocesses until you can go no further. The resulting subprocesses then become candidate Services for implementation. The more processes that a company decomposes in this way, the more they can identify commonality across their subprocesses and thus have a chance at building an appropriate set of reusable Services.

However, this top-down process decomposition approach has a critical flaw in that the organization might end up defining Services that are impossible or impractical to implement. Therefore, it's important to simultaneously go through a bottom-up exercise of taking existing business logic and exposing it as Services which themselves become candidate Services that specify not the overall business process, but rather the mechanism for implementing the process.

### The Iterative Approach to Service Granularity

Companies going through this Service definition exercise should make sure not to fall into the common, yet fatal trap of thinking that once they've defined their Services, they are done. SOA, by its nature, demands constant evolution. Even if the Services a company develops are perfect for the business at one time, the business will continue to undergo constant change, requiring new Services as well as new compositions of Services. What might have been the right level of granularity for a Service on day one might be inappropriate just a few weeks later. Implementing governance processes for SOA implementations can help simplify taking an iterative approach to Service definition. Through visibility and control of Services, an architect will have the appropriate insight and tools to increase the value of their Services.

Copyright © 2007, ZapThink, LLC


ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

As a result, it makes no sense to try to cast the level of granularity for a Service in concrete. Companies must approach Service design iteratively, building well-defined Service interfaces at a range of granularities and then establishing and using those Services that are appropriate at the time. Service-oriented architects will spend a good amount of their time tweaking Service interfaces such that they realize the optimal combination of fine vs. coarse-grained and single-purpose vs. multiple-use given the amount of knowledge they have about the business at that point in time.

*Building Services is not the goal of SOA; it's building an architecture that allows businesses to continuously evolve their set of Services that they can leverage despite ongoing change.*

As a result, developers and architects should resist the urge to "get the right Services." Indeed, getting it right doesn't even matter, since what's right today will surely be wrong tomorrow. After all, building Services is not the goal of SOA; it's building an architecture that allows businesses to continuously evolve their set of useful Services that the business wants and can leverage despite ongoing change. Building such useful Services is all about striking the balance between general-purpose and domain-specific Services as well as between overly-defined and overly-ambiguous Service interfaces. There's no cut and dried answer to what those Services should be for any particular company, but there certainly are good approaches to making those Services a reality. Spurring and encouraging this ongoing debate in the industry will only serve to make SOA more useful to the business.

### The Role of Data

The combination top-down, bottom-up approach to specifying Services lowers the risks inherent in building such Services, and can lead to the proper granularity for each Service over the course of a few iterations. Augmenting this iterative approach to Service definition is what might be termed the middle-out approach—begin with the business context of a particular Service and leverage the characteristics of the information that Service consumes or provides. For example, if an architect is specifying a customer information Service, the quantity, format, and structure of the customer information that the organization has on the one hand and that users require on the other can help guide the definition of the Service.

Taking the data-centric, or middle out approach to Service definition in isolation, however, has its perils as well. Considering information without the context of the business processes that leverage that information can lead to inflexibility. Similarly, a focus on the information requirements of a Service without the underlying context of the existing persistence infrastructure often results in poorly performing Services.

### The Role of Governance

Even considering a combination top-down, bottom-up, and middle-out approach to Service definition is still insufficient, because it doesn't take into account the organization's policies for Service definition, discovery, use, reuse, and versioning across the lifecycle of the Services. SOA means building for change, and as such, Services and compositions of Services must also change, as well. Architects shouldn't take a snapshot view of each Service, but instead must take a full lifecycle view that incorporates the reality of change.

The secret to achieving this full lifecycle view is governance. By building a governance framework as a precursor to building any Services in a SOA initiative, architects should work out their organization's policies regarding the lifecycle of Services: who species Services? How should developers publish Services? What are the policies for Service contracts? What are the organization's reuse,

Copyright © 2007, ZapThink, LLC 

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

versioning, and Service deprecation policies? It's vital for each enterprise to work out the answers to these questions as part of the Service definition exercise.

## II. Should All Services Be Reusable?

On first glance, the question as to whether all Services should be reusable has a straightforward answer. After all, since one of the goals of SOA is to build reusable Services, then why shouldn't all of them be reusable? But upon further reflection, answering this question properly requires greater subtlety as well as deeper architectural thinking. In fact, understanding when Services should be reusable and how to optimize reuse of such Services leads to critical Service design best practices.

### Focusing on Reuse

We still have not fully answered the question as to how to determine the proper level of granularity for a particular Service, since after all, we can build fine-grained, well-defined Services and coarse-grained, well-defined Services. It's vital to understand whether or not a particular Service is single-use or multiple-use. You could say that the best Services are the most reusable ones, which is true, up to a point. After all, having several redundant, fine-grained Services leads to tremendous overhead and inefficiency. Clearly having a small collection of coarser-grained Services that are usable in multiple scenarios is a better option. However, developers could theoretically take this principle to an extreme and try to build a single Service called, say, "DoSomething" that can meet every single need. DoSomething would have a simple interface that would support some arbitrary Service function requirement, and it would produce a corresponding Service result.

The problem with this DoSomething Service is quite obvious to anybody who has ever tried to implement such a thing. In essence, DoSomething is no longer a usable Service at all, since we've basically just passed the buck. Instead of the Service itself determining its own semantics, we've just shuffled that determination to some lower-level piece of code. In essence, we've treated SOA as just some sort of routing protocol or messaging system with no inherent functional capabilities. Such Services, however, are clearly unable to satisfy the broader business requirements of SOA.

So, if general-purpose Services are a red herring, what about single-purpose Services? The answer to this question is a bit of a draw. Some single-purpose Services, even though they might be very fine-grained and accomplish only one particular task, might be exceptionally reusable. That is to say, architects might be able to compose such Services into many different process scenarios. In contrast, domain-specific Services might only be applicable in certain scenarios, but the fact that they are specific to a particular problem or set of problems is what makes them useful to the business. And that is where we get our first clue for trying to solve the Service granularity issue: focus not on an individual Service, but rather on overall business processes and how Services might meet the needs of multiple processes in the business. The more a company can leverage a Service for multiple processes, the more useful it is. Correspondingly, if it's impossible to leverage a Service within several different processes, then you should wonder whether or not it is at the proper level of granularity.

*The more a company can leverage a Service for multiple processes, the more useful it is.*

### Top-Down Thinking and Service Reuse

The question as to whether all Services should be reusable depends upon whether you are taking a top-down or bottom-up approach to specifying those

Services. As discussed above, the top-down approach to SOA starts with a picture of all the business processes within an organization, and then looks to decompose those processes with an eye toward identifying areas of redundancy that indicate likely Service candidates. Since this top-down approach focuses on meeting business objectives and increasing efficiency through reducing redundancy, and hence increasing reusability, the key question architects should ask is whether maximizing reusability is always the most important best practice, or whether there are other priorities that the architect should consider as well.

In fact, while reusability is one of the most important criteria for determining which Services to build, there are other criteria that architects must consider. In particular, the *evolvability* of Services is a criterion every architect should consider. There may be Services that realistically have only one application, but may nevertheless experience ongoing requirement change. In other words, there are situations where agility rather than reusability is the driving force for Service design. While a particular Service might only have one consuming application, the fact that it is subject to continuously changing requirements dictates the design of the Service more so than its reusability. In such cases, it may be cost-effective to put in place the overhead necessary to ensure the required loose coupling for such Services, even though they may each only have one consumer.

*There are situations where agility rather than reusability is the driving force for Service design.*

Another example where reusability may not be the top priority for Service design is where loose coupling due to the contracted nature of the interface itself is the priority, especially in a business-to-business (B2B) situation. Two business partners trying to improve their automated interactions may be struggling with implementation-specific limitations (getting a Java-based application at one company to interact with a mainframe-based app at the other, for example). In such a case, building Services with contracted interfaces that abstract the underlying implementation may ease such interactions, even if such Services are only valuable for that single, point-to-point interaction.

### Bottom-Up Thinking and Service Reuse

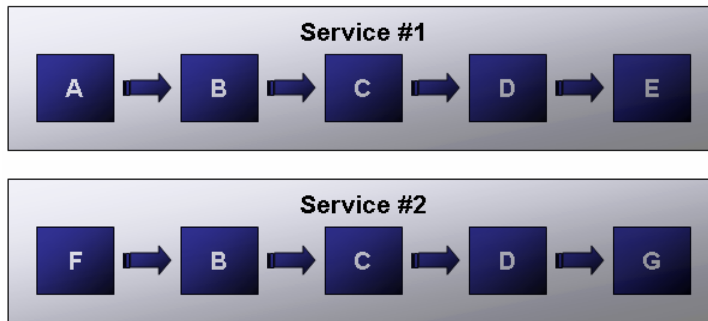
Taking the opposite tack to top-down thinking, the bottom-up approach to SOA starts by looking at existing IT capabilities in the enterprise as the starting point for developing Services that expose the existing functionality an organization already depends on. One of the key challenges that architects face in this scenario is how much of the existing functionality they should expose as Services, and at what level of granularity. If reusability is the primary driver for identifying areas of Service enablement, then it follows that some existing IT functionality may never end up as a Service. The challenge then is to determine which functionality should be exposed as Services, and more generally, prioritizing the Service enablement of existing functionality.

One way to get at solving the above challenge is to leverage a useful rule of thumb: the 80/20 rule, which states that 20% of the existing functionality in any given system will be used 80% of the time. The remaining 80% of the functionality handles special cases, exceptions, and other low-use scenarios. And while the greatest use doesn’t necessarily mean the greatest reuse, Service-enabling that 20% is likely to lead to the greatest level of reuse as a rule of thumb. Clearly, if an architect can identify the 20% of existing IT systems with the heaviest use, Service-enabling that portion of your functionality will provide the business its biggest bang for the buck, partly by easing access to the functionality, but even more significantly, by enabling broader reuse of that functionality as well. There may be reasons to Service-enable other IT assets, to be sure, but it’s less likely that reusability will be the primary driver for that follow-on enablement. In other words, functionality that is potentially less reusable is a lower priority for Service enablement.

Copyright © 2007, ZapThink, LLC 

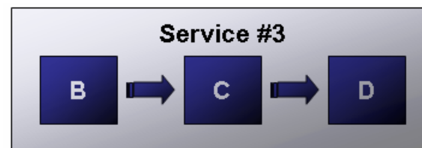
### Architecting for Proper Granularity

Let's work through an example of iterative SOA that also sheds additional light on how to deal with the fact that since neither the top-down, bottom-up, or middle-out approach alone is adequate, a core SOA best practice is to take an iterative approach that switches from top-down to bottom-up and back again, improving both the architecture and the Services with each iteration. Let's say we've defined (or already created) Services 1 and 2 that are compositions of individual Services, as illustrated in the figure below:



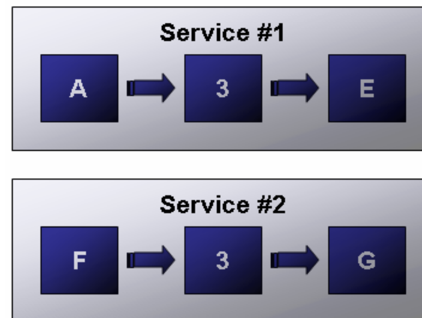
Source: ZapThink LLC

As an architect you're wearing your top-down hat, so you notice that Services 1 and 2 each share the composition of fine-grained Services B, C, and D. So in the next iteration you specify the coarser-grained Service 3 as follows:



Source: ZapThink LLC

Which enables you to redesign 1 and 2 as follows:



Source: ZapThink LLC

You've followed the familiar agile principle of refactoring out redundancy, yielding simpler, easier to maintain definitions for Services 1 and 2. But here's the key question: is Service 3 more or less reusable than Services B, C, and D? In fact, Service 3 will generally be less reusable than B, C, or D individually, because in the general case, these Services will be finer grained than Service 3, and hence

Copyright © 2007, ZapThink, LLC 

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

reusable in a broader set of situations than the coarser-grained, composite Service 3, because it's possible to consume B, C, and D separate from one another, while consumers of Service 3 are expecting B, C, and D together.

*Coarse-grained Services are more likely to be business-oriented than fine-grained ones.*

Nevertheless, it will make sense to leverage Service 3 in this way, when Service 3 is more *business-oriented* than B, C, and D separately. Coarse-grained Services are more likely to be business-oriented than fine-grained ones, and finding that Service 3 has a clearer business value than B, C, and D is sufficient motivation for building Service 3. After all, Services B, C, and D are still around, and if someone needs them in the future, they're reusable as well. But because Service 3 is coarse-grained and business-oriented, it may actually be more composable than Services B, C, and D.

Also keep in mind that Service 3 is still reusable, just not as reusable as some other Services. And remember, you're not simply architecting for today's Services, but you're architecting to enable the business to recompose Services as necessary to meet tomorrow's requirements as well. The fundamental architectural principle to keep in mind here is that coarse granularity can be more important than reusability when the level of granularity improves the composability of the Service.

### Use vs. Reuse

Not only have we added shades of subtlety to the question of whether Services should be reusable, we've also put a fine point on the definition of reusability itself by distinguishing it from composability. This distinction is subtle because, after all, composition of Services can be a form of reuse whenever it's possible to compose one Service into multiple processes. Understanding the difference between the two concepts, however, goes right at the heart of understanding SOA itself.

A Service would clearly be very reusable if there were a million Service consumers out there making regular use of the Service—and that would be a good thing, to be sure. But that Service may not be composable if business users aren't able to find a way to incorporate it into composite applications that implement Service-oriented business processes (known as Service-Oriented Business Applications, or SOBAs). If a Service is truly composable, therefore, then the business should be able to find many ways to compose that Service into different SOBAs. Such composability is where much of the true business value of SOA lies.

## III. The Service Granularity Matrix

Just as object orientation brought the issue of encapsulation to the fore, Service orientation highlights the challenge of granularity. The concept of granularity is a relative measure of how broad the interaction between a Service consumer and provider must be in order to address the need at hand. However, there is no single measure for fine granularity or coarse granularity. Rather, the measure applies in relation to the Services available and the number of interactions required to accomplish a specific goal.

The concept of granularity is incredibly important to the architect because it has a direct impact on two major goals of SOA: the composability of loosely-coupled Services, and the reusability of individual Services in different contexts. The architect might design a particular Service at a fine level of granularity to give it the greatest amount of reuse across multiple process scenarios. Alternatively, the architect might realize that a particular business context could require a

Copyright © 2007, ZapThink, LLC

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

coarser level of granularity. In essence, architects have to balance flexibility against business value to help determine granularity.

### Atomic vs. Composite Services

An important SOA best practice is the ability to expose a composition of Services as a Service. For example, a telco might wish to expose a cell phone provisioning Service, where that Service actually abstracts a multi-step business process that they have in turn implemented as a composition of Services. Clearly, however, not all Services abstract compositions, and in fact, many SOA implementations have no such Services.

Most SOA implementations, in fact, consist solely of atomic Services, which abstract implementations that are not themselves composed of Services. However, this definition of atomic presents some issues. What might seem to be atomic in the eyes of a Service consumer since it appropriately has no visibility into the implementation of that Service might actually be composite from the perspective of the Service provider. Indeed, to all Service consumers, the Service appears as atomic even if it might be composite. The only way to definitively know if a Service is composite is for a Service consumer to inspect some declarative logic that identifies specifically how a Service is composed. Of course, for a Service consumer to be required to have that visibility violates the rule of loose coupling. No Service consumer should be required to know how a Service provider is implemented.

*The concept of atomic vs. composite Services is incredibly important to the enterprise architect.*

However, the concept of atomic vs. composite Services is incredibly important to the enterprise architect. In particular, SOBAs are composed of Services and can in turn expose Service interfaces. As such, when architects define their business Services in a top-down manner what they are actually doing is defining which processes to implement as composite Services, and which to implement as atomic Services, at which point decomposition activity stops and Service implementation begins.

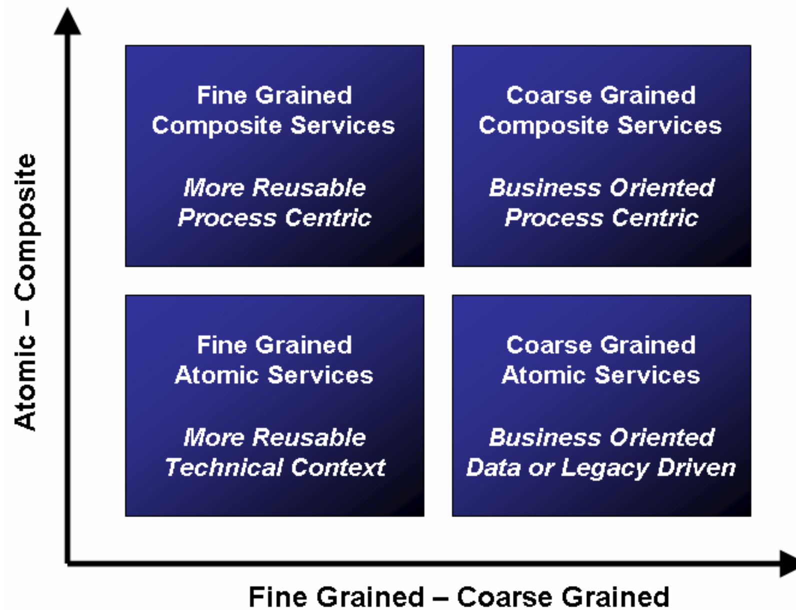
### Evaluating the Service Granularity Matrix

It's important to discuss the notions of atomic and composite Services at this point, because while it might seem that all atomic Services are fine-grained and composite Services coarse-grained, this is not necessarily the case. Since granularity is a measure of interaction and business context, and atomicity is a measure of process decomposition, it is quite possible to have coarse-grained atomic Services and fine-grained composite Services. As a way of explaining how this might come about, consider the Service Granularity Matrix in the figure below:

Copyright © 2007, ZapThink, LLC


ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

The Service Granularity Matrix



Source: ZapThink LLC

As the figure above points out, coarse-grained Services that could easily be atomic because there's no way to decompose them further. A common example of a coarse-grained atomic Service would be a Service that exposes a single database query that returns a large block of information. Or perhaps the Service abstracts a mainframe-based application where the lowest level of granularity you can get is still coarse, because of the closed nature of the legacy application.

Likewise, consider a case where architects have specified a Service to provide a small part of a large process. This individual Service may not send or receive much information, as its role is more as a step in a process than as a data Service. Similarly, the resulting composition may also be fine-grained, because it focuses more on the process functionality than on sending or receiving information. As a result, the composition itself is fine-grained. This situation might appear in situations where the Services are supporting business processes in multiple contexts, such as fine-grained Services that are themselves composed of atomic Services exposed from legacy assets.

## IV. The Business Service Abstraction

Up to this point this paper has been talking about Services both as interfaces and as abstractions—and fundamentally, Services fill both these roles. In a fully implemented SOA, however, there are typically different levels of abstraction, as how the business perceives and utilizes a business Service goes beyond accessing a particular Service interface. Therefore, clearly distinguishing among abstracted Services, Service interfaces, and also the underlying Service implementations is a critical capability of any SOA architect.

### Implementations, Interfaces, and Abstractions

One of the numerous challenges facing the architect is the fact that the term "Service" is overloaded even within the IT context. Even within the SOA context,

Copyright © 2007, ZapThink, LLC


ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

people still often get confused about the level of abstraction of a Service. Basically, there are three levels of abstraction we work on in the context of SOA:

1. *Service implementation* – at this level of abstraction we're talking about software. A Service implementation is made up of running code. This is where the Service Component Architecture (SCA) lives, as it deals with Service components, which are implementations can consume or provide Services (in the sense of #2 below).
2. *Service interface* – Web Services live at this level, as a Web Services Description Language (WSDL) file provides a contract for the interface, but says nothing about the underlying implementation. Web Services, however, are not the only kind of Service interface, because Service contracts are not always WSDL files. Sometimes Service interfaces are loosely coupled, but many times they're not.
3. *Abstracted Service* – A representation of a business capability or data that the organization can compose with other such Services to implement business processes. An abstracted Service is typically a business Service, but not necessarily, as there is a role for abstracted IT Services as well. However, all business Services should be abstracted Services. Such business Services are the core abstraction that underlies SOA. Abstracted Services should always be loosely coupled, although the specific coupling requirements can vary. Building loosely coupled abstracted Services thus becomes the core technical challenge of implementing SOA.

*Business Services are the core abstraction that underlies SOA.*

So far so good—but the real question here is how we make an abstracted Service actually work, when the tools at our disposal are the Service implementations and interfaces and all the infrastructure that goes along with them. It's one thing to talk about "representations of business capabilities," and quite another to string your ones and zeroes together into something that actually runs.

### Service Contracts and SOA Infrastructure

The first critical point to understanding abstracted Services is to understand that there is typically a many-to-many relationship between Services and Service contracts. Clearly, a Service implementation may support multiple contracts, each of which could correspond to a particular Service interface, for, say, a particular type of consumer. Similarly, there might be several implementations that support a single contract, and hence a single Service interface, for the purposes of scalability or fault tolerance, for instance.

With abstracted Services, however, the relationship becomes what we might call "many-to-many-to-many": a particular abstracted Service might have several contracts that represent relationships with various consumers, while also representing multiple Service interfaces that themselves might each have one or more Service implementations. This approach might sound overly complex, but it's the key to loosely coupling the abstracted Service. To illustrate this point, let's work though an example.

Let's say we have a Customer Information Service that different lines of business in a large enterprise can consume and compose to provide or update any information about their customers that the lines of business might need. From the business perspective, this is a single, coarse-grained business Service that any line of business can use as per the policies set out for that Service. From the IT perspective, however, it makes sense to implement the Customer Information Service as a set of Service interfaces with different Service contracts, either atomic or composite, in order to support the somewhat different needs for customer information that the various lines of business might have. Furthermore,



ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

each Service interface may represent several Service implementations that the SOA management infrastructure can leverage as necessary to meet the service levels set out in the contracts for both the abstracted Service as well as the Service interfaces, in addition to the policies that may apply to these Services as well as other Services in production.

In this example, the complexity beneath the Service abstraction is necessary to support the loose coupling of the abstracted Service. For example, the line of business consumers may need different formats for the customer information, or may require different data as part of the response from the Service. To loosely couple such consumers, an intermediary (or set of intermediaries) may perform a transformation that can take the output from any of the Service interfaces and put it into the format the particular consumer requires, as per the contract in place that governs the relationship between that particular consumer and the abstracted Service. Then, either the management infrastructure (or possibly the integration infrastructure) may offer content-based routing of the requests from particular Service interfaces to the underlying implementations, based upon runtime policies in effect at the time.

### Business Services, Loose Coupling, and Governance

Furthermore, a Service interface may support several contracts, for example, when one Service interface has multiple bindings. In the case of a Web Service, each WSDL file specifies a binding, so to support more than one, there should be multiple Service contracts for the Service interface. Each binding may then correspond to its own Service implementation, or in the more general case, multiple implementations may support each binding, or one implementation may support multiple bindings.

*Loose coupling means more than being able to support different consumers with different needs. It also means building for change.*

In any case, loose coupling means more than being able to support different consumers with different needs. It also means building for change. Because we have a governance and management infrastructure in place that enables this many-to-many-to-many relationship among abstracted Services, Service interfaces, and Service implementations, we are able to respond to those changes in a loosely coupled manner as requirements evolve—in other words, without breaking anything.

For example, if one consumer changed its required data format, we could introduce a new contract which might require a new transformation on the intermediary between the Service interface and the abstracted Service, but wouldn't impact the Service interface directly or any of the Service implementations. Another example might be the need to upgrade or add a new data source to support the Service. Such a change might require a new implementation of one or more Service interfaces. But if the contracts for those interfaces don't change, then the abstracted Service is unaffected, and neither are the consumers. A third example would be a policy update that would change the content-based routing behavior between the Service interfaces and their implementations. In fact, we see this application of content-based routing as more of a management challenge than an integration task because of this need to support runtime policy changes.

## V. The ZapThink Take

One of the greatest challenges of SOA is the fact that the architectural approach consists of a broad set of best practices—tools in the architect's tool belt, if you will. But just as with any set of powerful tools, a skilled practitioner must know which tools are appropriate for solving which problems. A good carpenter would

Copyright © 2007, ZapThink, LLC 

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

never select a screwdriver to hammer in a nail, and a good architect must also know how to leverage the tools of loose coupling, coarse granularity, and abstraction when specifying Services.

Among these best practices include the following:

- Taking a top-down, business process-driven approach to defining Service granularity
- Taking a bottom-up approach to Service granularity that takes into account existing application and data assets
- Taking a middle-out approach to Service granularity that begins with the business context of the information a Service is expected to send or receive
- Taking an iterative approach to Service definition that combines the three above approaches in a way that delivers business value and reduces risk
- Distinguishing between atomic and composite Services, and specifying the level of granularity and the composite nature of each Service independently, depending upon the business context of the Service
- Building business Services at a higher level of abstraction than Service interfaces, which in turn abstract the underlying Service implementations.

Architects who not only understand these best practices, but have a grasp as to when to apply them, will be in an excellent position to implement SOA to successfully address the problems the business faces.

Copyright © 2007, ZapThink, LLC

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com

## Copyright, Trademark Notice, and Statement of Opinion

All Contents Copyright © 2007 ZapThink, LLC. All rights reserved. The information contained herein has been obtained from sources believed to be reliable. ZapThink disclaims all warranties as to the accuracy, completeness or adequacy of such information. ZapThink shall have no liability for errors, omissions or inadequacies in the information contained herein or for interpretations thereof. The reader assumes sole responsibility for the selection of these materials to achieve its intended results. The opinions expressed herein are subject to change without notice. All trademarks, service marks, and trade names are trademarked by their respective owners and ZapThink makes no claims to these names.

## About ZapThink, LLC

ZapThink is an IT advisory and analysis firm that provides trusted advice and critical insight into the architectural and organizational changes brought about by the movement to XML, Web Services, and Service Orientation. We provide our three target audiences of IT vendors, service providers and end-users a clear roadmap for standards-based, loosely coupled distributed computing – a vision of IT meeting the needs of the agile business.

ZapThink helps its customers in three ways: by helping companies understand IT products and services in the context of Service-Oriented Architecture (SOA) and the vision of Service Orientation, by providing guidance into emerging best practices for Web Services and SOA adoption, and by bringing together all our audiences into a network that provides business value and expertise to each member of the network.

ZapThink provides market intelligence to IT vendors and professional services firms that offer XML and Web Services-based products and services in order to help them understand their competitive landscape, plan their product roadmaps, and communicate their value proposition to their customers within the context of Service Orientation.

ZapThink provides guidance and expertise to professional services firms to help them grow and innovate their services as well as promote their capabilities to end-users and vendors looking to grow their businesses.

ZapThink also provides implementation intelligence to IT users who are seeking guidance and clarity into the best practices for planning and implementing SOA, including how to assemble the available products and services into a coherent plan.

ZapThink's senior analysts are widely regarded as the "go to analysts" for XML, Web Services, and SOA by vendors, end-users, and the press. Respected for their candid, insightful opinions, they are in great demand as speakers, and have presented at conferences and industry events around the world. They are among the most quoted industry analysts in the IT industry. ZapThink was founded in November 2000 and is headquartered in Baltimore, Maryland.

**ZAPTHINK CONTACT:**
ZapThink, LLC
108 Woodlawn Road
Baltimore, MD 21210
Phone: +1 (781) 207 0203
Fax: +1 (815) 301 3171
info@zapthink.com

ZapThink, LLC • 108 Woodlawn Road • Baltimore, MD 21210 • info@zapthink.com • www.zapthink.com