SYS-CON

THE WORLD'S LEADING MAGAZINE DEDICATED TO WEB SERVICES TECHNOLOGIES

# SOA WORLD MAGAZINE

www.SOA.SYS-CON.com

JUNE 2008 / VOLUME: 8 ISSUE 6

## Introducing SOA Design Patterns

THOMAS ERL PAGE 2





$6.99US $7.99CAN

06

0 71486 03420 9

# Introducing SOA Design Patterns

## The SOA community collaborates to produce a master pattern catalog dedicated to SOA

BY THOMAS ERL

Originally inspired by techniques used to design buildings and cities, and popularized by the Gang of Four during the mainstream emergence of object-orientation, design patterns have seen us through the various shifts in architecture, technology, and, of course, design.

Pattern catalogs have periodically emerged, one building on the other, and each revealing a set of problem-solving techniques and providing invaluable insights as to how and when those techniques should be used to help us attain our design goals.

SOA has its own history, having risen out of a haze of ambiguity to establish itself as the basis of a distinct and maturing distributed computing platform with a distinct and ambitious design paradigm in its own right.

And now, finally, these worlds converge. SOA and service orientation (and surrounding technology platforms) have matured to the extent that proven design practices have surfaced for use by the masses. Subsequent to years of research, reviews, and validation, this body of work has been formally documented as a comprehensive collection of over 90 SOA design patterns.

One of the most intriguing aspects of the SOA design pattern catalog is its breadth. We have patterns providing design techniques that range from adjusting minute validation logic in a service contract to design strategies that help us structure pools of services across an entire enterprise.

This scope is indicative of the enterprise-centric focus of service-oriented computing in general. When carrying out an SOA initiative, we need to pay attention to many design details with every service we deliver, while always keeping the big picture in our sights. Design patterns support us in maintaining this balance by helping us overcome common obstacles that have historically inhibited or even derailed SOA project plans.

Each pattern is like a piece of wisdom resulting from the trials and errors of pioneers and the sweat and tears (and therapy) that sometimes accompanied those early SOA project experiences. So, please, a moment of silence for those who have suffered so that we can now benefit...

All right then, without further ado, let's introduce the SOA design patterns.

### Patterns in a Service-Oriented World

Service-oriented computing has a specific set of strategic goals and benefits associated with it. Most of these goals, such as increasing agility and ROI, are well known, as is the fact that to attain these goals, you need to design your solutions by following service orientation, a distinct design approach tailored to support service-oriented computing.

There's a close relationship between the service-oriented architectural model and the service-orientation design paradigm. It is through the application of service-orientation design principles that you end up creating software programs that are legitimately "service-oriented." When you implement SOA as a technology architecture you establish an environment that is conducive not just to enabling the creation of effective service-oriented solutions, but also to

enabling the effective long-term governance and evolution of the individual services that can be composed and recomposed to comprise these solutions.

## Design Patterns and Architecture Types

Each SOA design pattern provides a design solution in support of successfully applying service orientation and establishing a quality service-oriented architecture. Therefore, to better understand how and to what extent individual SOA design patterns can be applied, SOA as an architectural model itself needs to be broken down into the following types, each of which represent a common "scope of implementation":

- **Service Architecture** – The architecture of a single service.
- **Service Composition Architecture** – The architecture of a set of services assembled into a service composition.
- **Service Inventory Architecture** – The architecture that supports a collection of related services that are independently standardized and governed.
- **Service-Oriented Enterprise Architecture** – The architecture of the enterprise itself, to whatever extent it is service-oriented.

In a typical enterprise, these architecture types are very much interrelated, yet each requires individual design attention and documentation.

## About the Patterns

This article is roughly organized according to these architecture types and related patterns.

SOA design patterns collectively form a master pattern language that allows patterns to be applied in different combinations and sequences. There are also compound patterns that are comprised of multiple individual design patterns. (For example, the Enterprise Service Bus and orchestration both represent compound design patterns.)

SOA design patterns are not specific to any particular vendor platform or business industry; they are simply design techniques that help overcome common obstacles to achieving the strategic goals and benefits associated with SOA and service-oriented computing.

The remainder of this article highlights key design patterns while referencing a cross-section of others. Not all mentioned patterns are explained, but descriptions are freely available at the SOA patterns community site (www.soapatterns.org).

### Service Inventories

When building services as part of an SOA project, there is an emphasis on developing them as standalone programs that are expected to be flexible and robust so that they can be readily reused and composed. Service-oriented design has therefore been heavily influenced by commercial product design techniques to the extent that a service is delivered as a black box somewhat comparable to a software product. Inspired by commercial terminology, a collection of services for a given segment of an enterprise is referred to as a "service inventory." And, similarly, the technology architecture that supports this collection of services is referred to as the "service inventory architecture."

What's the difference between a service inventory and a service catalog? The same manner in which an inventory of products is documented with a product catalog, an inventory of services is documented with a service catalog. It's therefore still appropriate to refer to a collection of services as the service catalog; however, when applying design patterns and defining the actual concrete architecture, terms like "service inventory" (or even "service pools") tend to work better.

## Patterns for Collections of Services

A service inventory represents a collection of independently standardized and governed services. As shown in Figure 1, the services you deliver for a given service inventory are standardized and designed according to service orientation so that they become intrinsically interoperable. This then allows you to draw from this pool of services to assemble and augment service compositions repeatedly.

## Inventory Boundary Patterns

One of the biggest decisions a project team faces when starting an SOA initiative is determining the appropriate scope of a service inventory. The Enterprise Inventory and Domain Inventory design patterns help address this decision point by providing alternative approaches.

The goal of the Enterprise Inventory pattern is to establish an enterprise-wide service inventory. The end result of achieving this pattern is considered desirable because it enables you to build all of your services according to the same design conventions to ensure consistent and widespread inter-service compatibility. It further guarantees that all of the services will be owned and evolved by the same group or department, which is the ultimate in centralized governance.

Although ideal, this approach is often not realistic, especially for larger organizations. It can raise various issues, including time and budget constraints, cultural and political concerns, and order of magnitude considerations (especially in relation to the long-term growth and governance of the inventory). These issues can introduce risks and problems that outweigh the benefit potential of applying this pattern.

This is the reason the Domain Inventory pattern has become so popular. It advocates an approach whereby the enterprise is divided into segments (domains), each of which represents a meaningful cross-silo scope. Often, the boundary of a domain inventory architecture is aligned with a business domain (such as accounting or claims). Services delivered into this architectural boundary are subject to the same design standards and governance practices, allowing them to be evolved independently from neighboring domain inventories in the same enterprise.

Although the use of this pattern can introduce the need for cross-domain data model and protocol conversion (as per the Schema Transformation and Protocol Bridging patterns), there are additional patterns (such as Cross-Domain Utility Layer, Dual Protocols, and Inventory Endpoint) that help reduce this impact.

## Inventory Structure Patterns

Regardless of its scope, within the boundary of a service inventory, certain design patterns are applied to ensure a consistent structure in support of service orientation. For example, Logic Centralization positions reusable services as the sole or primary contact points for the logic they represent. This is further supported by the Service Normalization pattern that fosters service autonomy by reducing the amount of functional overlap between individual service boundaries to establish more of a "normalized" inventory.

The Service Layers pattern (and related, specialized layer patterns) can be used to further organize a service inventory into a set of logical layers, each of which is based on a different classification of service.

**Note:** *Just a reminder that all of these structural patterns are only applied within the boundary of an inventory architecture. This means that, if you are working within the confines of a domain inventory, these patterns will not be applied on an enterprise-wide basis.*

To support and extend the structure of a service inventory archi-

tecture, various other design patterns can be applied. Some (like Canonical Schema and Canonical Transport Protocol) help standardize the services within the inventory boundary to foster native interoperability and composability, while others (like Process Centralization and Rules Centralization) can be selectively used to leverage established product platforms that support the centralized management of business process logic and business rules, respectively.

Yet another dimension to inventory architecture design is the centralization of service contract-related logic. The creation of redundant schema and policy content can be addressed by the Schema Centralization and Policy Centralization patterns, each of which establishes a separate data representation layer (one layer for data models, the other for global and domain-level policies) that supports the primary service contract layer.

There are many more specialized patterns that are applied to an inventory architecture to solve common problems related to resource management, state management, quality of service, security, and communication.

## Patterns for Service Design

Each service exists as a standalone software program, autonomous yet still fully geared to participate in larger service aggregations. When designing a service architecture, numerous challenges can arise, especially when shaping this architecture according to service-orientation design principles, such as Service Statelessness and Service Loose Coupling. Figure 2 provides an abstract glimpse of service architecture design patterns that are applied at the service architecture level.

### Agnostic Logic and Non-Agnostic Logic

The term "agnostic" originates from the Greek word "agnostos," which means "without knowledge." Therefore, logic that is sufficiently generic so that it is not specific to (has no knowledge of) a particular parent task is classified as agnostic logic. Because knowledge specific to single-purpose tasks is intentionally omitted, agnostic logic is considered multi-purpose. On the flipside, logic that is specific to (contains knowledge of) a single-purpose task is labeled as non-agnostic logic.

Another way of thinking about agnostic and non-agnostic logic is to focus on the extent to which the logic can be re-purposed. Because agnostic logic is expected to be multi-purpose it is subject to the Service Reusability principle with the intention of turning it into highly reusable logic. Once reusable, this logic is truly multi-purpose in that it, as a single software program (or service), can be used to automate multiple business processes.

Non-agnostic logic does not have these types of expectations. It is deliberately designed as a single-purpose software program (or service) and therefore has different design priorities.

## Fundamental Service Design

At the most basic level of service design, the established "separation of concerns" theory needs to be applied as part of a process whereby a larger problem is decomposed in a way that we can clearly identify how corresponding solution logic should be partitioned into services. To accomplish this, a series of base patterns have emerged to form a fundamental pattern language that can serve as the basis of a primitive design process.

Examples of these basic patterns are Functional Decomposition and Service Encapsulation, both of which help determine what type of logic does and does not belong in a service. Additional patterns, like Agnostic Context and Non-Agnostic Context provide criteria that help determine whether certain kinds of logic are deemed sufficiently agnostic to be put into a reusable or multi-purpose service (see Figure 3).

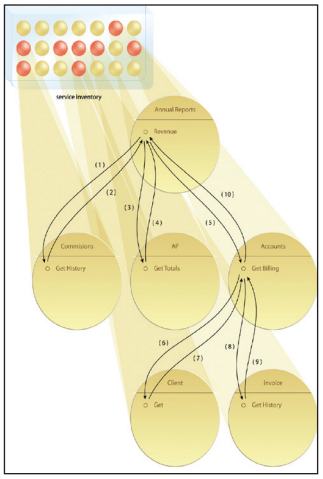


Figure 1

## Service Implementation Design and Governance

A key pattern frequently applied to the initial design of service architecture is the Service Façade. Inspired by the Façade pattern created by the Gang of Four, this pattern wedges a component between the service contract and the core service logic to establish a point of abstraction. This is really nothing new, but it does establish an architectural foundation that can be leveraged by other more specialized SOA patterns.

For example, Service Decomposition (one of the service governance patterns) allows a coarse-grained service to be split up subsequent to its deployment. Additional governance patterns, such as Proxy Capability and Distributed Capability (Figure 4), help ensure that this decomposition of one service into two or more does not impact the contract (technical interface) of the original service, thereby also avoiding impact to that service's consumer programs.

Other patterns, such as Legacy Wrapper, Redundant Implementation, Service Refactoring, and Service Data Replication, can be selectively used to address various implementation and scalability-related requirements, while maintaining the overall flexibility required for services to be repeatedly composed and extended, as required.

### "Capability" v "Operation" v "Method"

Service contract design patterns avoid the use of terms like "operation" and "method" because they are generally associated with a specific implementation platform (like components or Web Services, respectively). The term service capability is used to represent a specific function of a service that it can perform and through which it can be invoked. Service capabilities are generally expressed by the service contract. This term is also common during early service modeling stages, when service candidates are conceptualized prior to determining how they will be physically designed and developed.

## Service Contract Design and Governance

A key design pattern that helps increase the longevity of service contracts (to postpone versioning requirements) is Contract Denormalization. This pattern essentially explains how capabilities with overlapping functional scopes can be added to a service contract without negatively impacting service or inventory architectures. This results in the contract's technical interface exposing similar functions at different levels of granularity, allowing the same contract to facilitate the requirements of different types of consumers.

Alternatively, multiple groups of consumers can be accommodated by the Concurrent Contracts pattern that describes how entirely separate contracts can be created for the same underlying service logic. Security restrictions or a need to split up policy alternatives for reasons of governance can also result in the requirement to employ multiple service contracts. This pattern also benefits from the existence of a service façade that can be positioned to coordinate

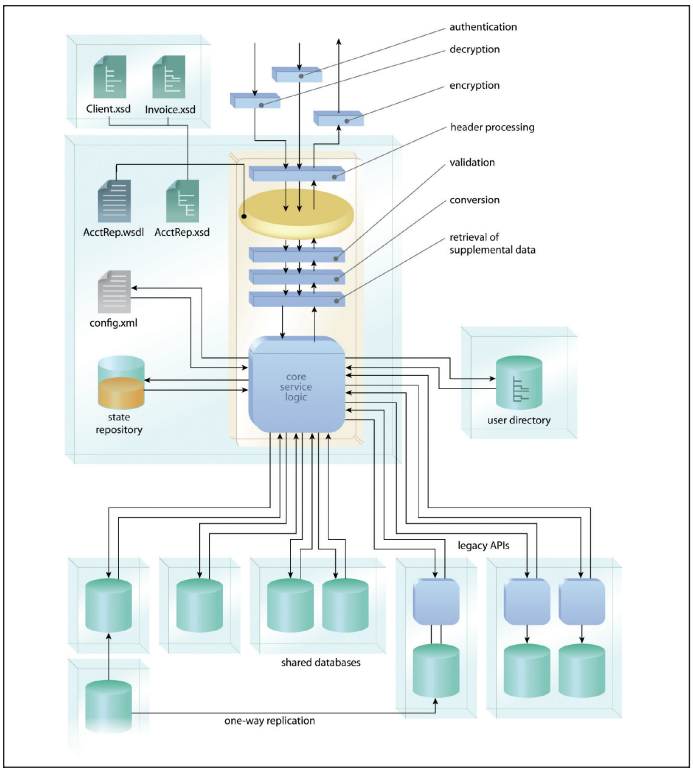


Figure 2

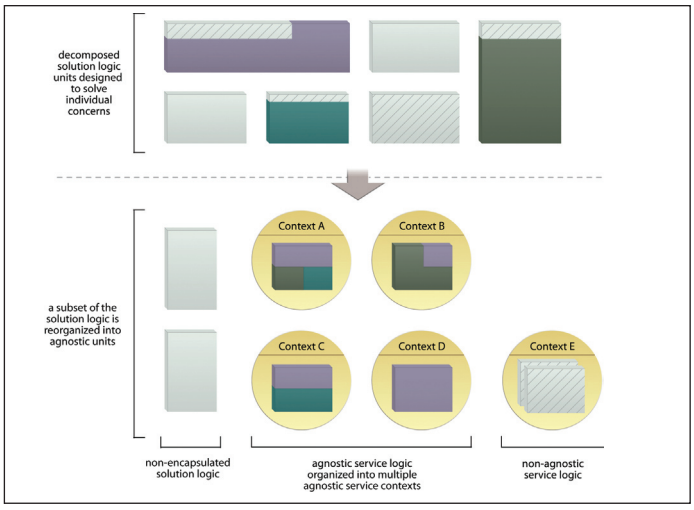


Figure 3

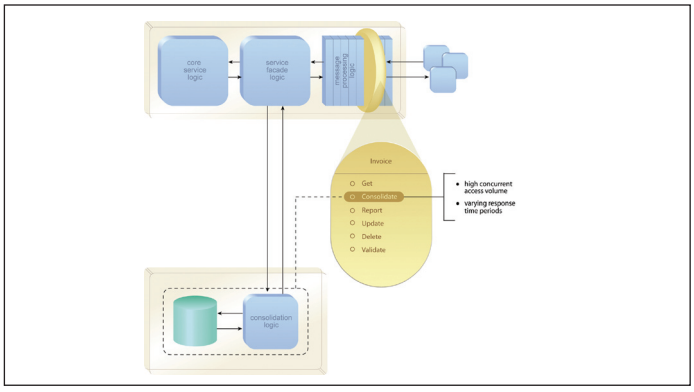


Figure 4

incoming and outgoing data exchanges across multiple contracts but with a single body of core service logic.

Part of these governance patterns includes various contract versioning design techniques to minimize the impact of having to introduce new contract versions or support multiple versions of the same contract. Some governance patterns are reactive because they help solve unexpected governance-related design issues, while others are preventative in that they can be applied prior to the initial deployment of a service in order to build in additional flexibility that allows the service to be more easily governed and extended over time.

## Patterns for Service Composition and Communication

As an aggregate set of services, a service composition establishes its own unique architecture encompassing the individual service architectures of the composition members and introducing additional design requirements focused on inter-service communication and runtime activity management. It's therefore no surprise that many design patterns have emerged to address composition-related design issues.

There are several key design patterns that establish the mechanics behind inter-service communication. Due to the popularity of building services as Web Services, several of these design patterns are based on messaging, and some are focused solely on asynchronous messaging. For example, the Asynchronous Queuing design pattern establishes a central queue to allow services to overcome availability issues and increase the overall robustness of asynchronous data exchange.

The marriage of SOA and EDA has resulted in the Event-Driven Messaging pattern that enables publish-and-subscribe functionality between services over extended periods. This can go hand-in-hand with the use of the Service Agent pattern that introduces an additional event-driven dimension into composition architecture by allowing you to defer various types of cross-cutting logic to transparent agents that intercept and forward messages at runtime (Figure 5).

The Intermediate Routing pattern takes this a step further by providing intelligent agent-based message routing that can also help increase the overall scalability of services and compositions.

Very much related to supporting service messaging and the hosting and execution of service compositions as a whole is the Enterprise Service Bus compound pattern. As shown in Figure 6, this pattern establishes an environment comprised of multiple other patterns, such as the aforementioned Intermediate Routing and Asynchronous Queuing patterns in addition to the Broker pattern (that in itself is also a compound pattern that represents a set of individual transformation patterns).

Note that there are additional design patterns associated with the Enterprise Service Bus compound pattern, several of which are classified as optional extensions to a core model. Other design patterns that tie into service composition architecture include Cross-Service Transaction, Composition Autonomy, Reliable Messaging, and Agnostic Sub-Controller.

## Working with SOA Design Patterns

Before we conclude, here are just some guidelines for getting the most out of SOA design patterns.

### *View Patterns with a Strategic Context*

One of the greatest expectations of SOA initiatives is for the IT enterprise as a whole to become a more streamlined, responsive,

and efficient part of the overall organization. Regardless of whether you are applying service-orientation design principles or SOA design patterns, it's important to keep these strategic goals in mind because they help you understand why there are certain priorities, preferences, and sacrifices that need to be made in order to deliver truly valuable service-oriented logic.

### *Take Governance into Consideration*

Many IT professionals focus on governance as a project lifecycle phase that begins after services and service-oriented solutions have been deployed and are in use. However, governance itself is also very much a design-time consideration. Just about every decision point you face when designing services, service compositions, and service inventories will have governance-related implications. Understanding the long-term consequences of design decisions in advance will help you choose and combine design patterns wisely to minimize the governance impact of anything you end up building. This is a critical consideration because it's through reduced governance burden that you can truly achieve the strategic goals of service-oriented computing.

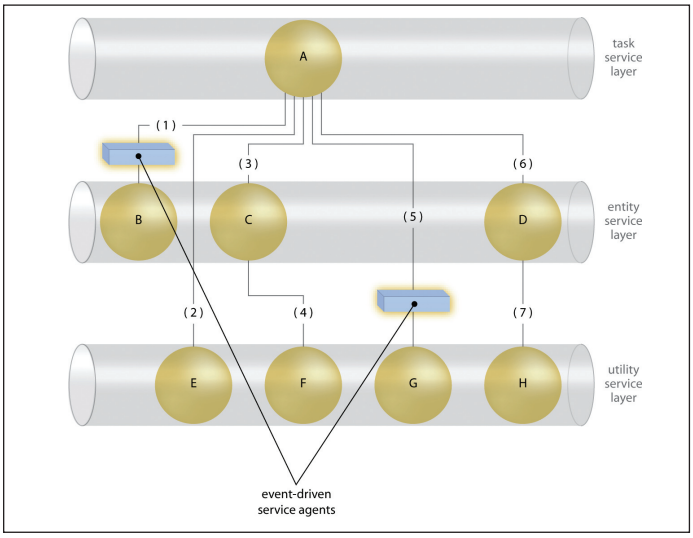


Figure 5

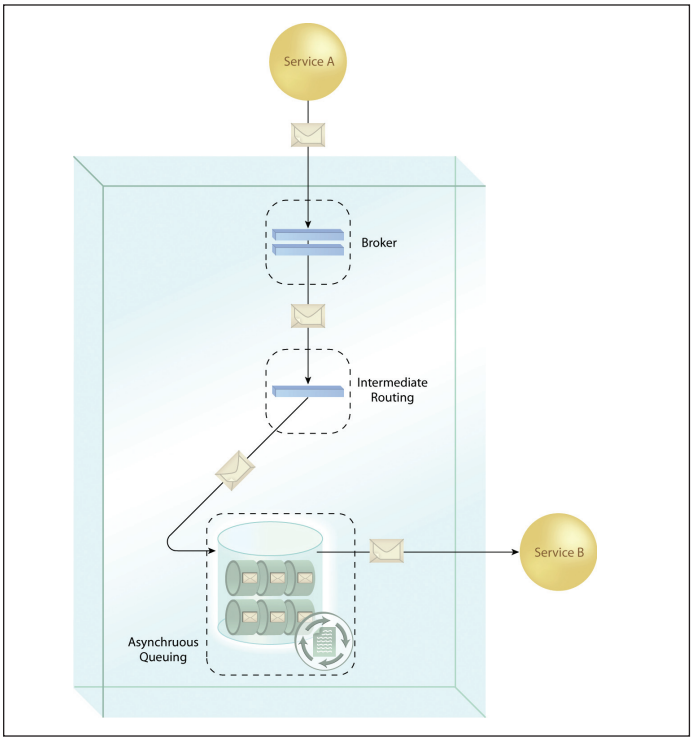


Figure 6

### *Patterns are Applied in Measures*

As with service-orientation design principles, SOA design patterns can (and often must) be applied to various extents. This means that for any given pattern, you are not faced with an all-or-nothing proposition. Some patterns need to be applied in a limited capacity, while with others it makes a whole lot of sense to maximize their application to whatever degree possible. Either way, it's always good to keep in mind that the benefit promised by a given pattern is generally tied by the measure to which it is applied.

### *Acknowledge that Some Patterns Are Evolutionary*

For those of you with an OOD or EAI background, several of the design patterns mentioned here probably looked familiar. Service orientation owes much of its existence to past design paradigms. What makes it distinct are the parts of those paradigms that were not included, combined with new design considerations and techniques that are particular to the enterprise-centric focus of service-oriented computing.

The same goes for SOA design patterns. While some of the patterns in this catalog are based on (or even derived from) patterns in previously published pattern catalogs, they are documented in the specific context of SOA and service orientation. Other SOA design patterns introduce brand new design techniques that complement and build on the derived patterns.

### *Patterns Relate to Patterns*

Most SOA design patterns have numerous relationships with each other (see Figure 7). For example, one pattern may have a dependency on another, or its application may influence another, or perhaps it is simply part of a larger compound pattern. With an understanding of inter-pattern relationships, you can avoid potential conflicts (some of which don't reveal themselves until later). Additionally, you can fully leverage the SOA design pattern catalog as a full-fledged pattern language that supports the application of patterns into various creative pattern sequences.

### *Patterns Relate to Principles*

Each SOA design pattern affects and influences the application of one or more service-orientation design principles. By identifying these relationships, you can leverage specific patterns when working with the design principles individually. There are also adverse relationships, where the results and trade-offs of some patterns negatively impact the goals of a design principle. Again, knowing these relationships in advance will help you make educated design decisions. (For more details regarding service-orientation design principles, visit www.soaprinciples.com.)

### *Not Every Pattern Is Suitable for Everyone*

As mentioned at the beginning of this article, this collection of SOA design patterns was produced as a result of numerous (successful and failed) projects, on-going research, and documented lessons learned. The focus of each pattern is on providing a solution

to a common problem. Just because a given problem was common elsewhere does not mean it will apply to your environment. Also, you always have the option of taking ideas from these design patterns and then deriving your own distinct design techniques.

## A Final Word

We've really just skimmed the surface with this exploration of SOA design patterns. The pattern catalog has nearly 100 design patterns, the majority of which we haven't been able to mention in this article. Examples of other types of patterns that have been documented include those that address grid computing and balanced stateful service design, REST-based communication and service design, security technology, attack prevention and perimeter protection, versioning of services contracts and related schemas and policies, dynamic messaging, runtime compensation, the use of binary attachment technologies, various transformation requirements, and others as it evolves.

All of these patterns are intended to help you make a success out of an SOA initiative. How success is measured in the SOA world often comes down to how well the service-oriented environment you create represents the business of your organization – and how well it continues to stay in alignment with the business.

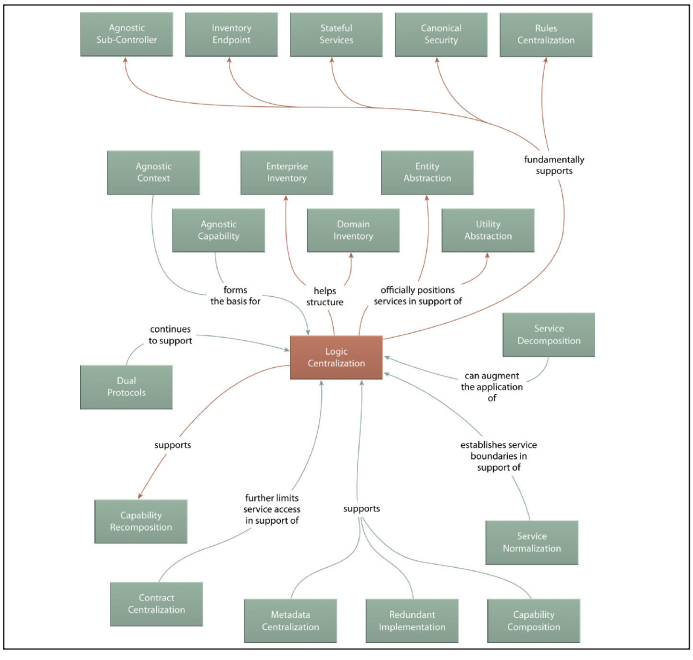


Figure 7

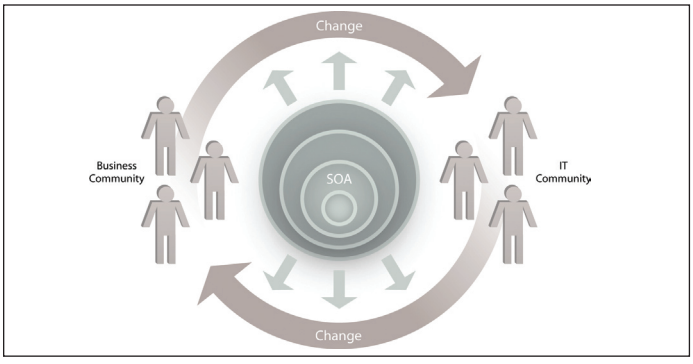


Figure 8

Figure 8 illustrates the never-ending progress cycle that continually transpires between business and IT communities. The result of this dynamic is constant change, and if we had to choose the one thing responsible for inspiring service-oriented computing in the first place, it would be that word "change."

Ultimately, service orientation and the implementation of the different service-oriented technology architecture types (inner circles in Figure 8) support the two-way dynamic between business and IT communities (outer circle in Figure 8), allowing each to repeatedly introduce or accommodate change. SOA design patterns help establish and strengthen SOA architectures in support of service-orientation, a paradigm specifically geared toward facilitating this cycle of constant change.

From the minute you design them to the months and years later when you extend or build on them, your services, your compositions, and the inventory architectures in which they all reside must be equipped to deal with change. Only this way can your technology enterprise evolve in tandem with the business.

To establish such an environment requires a different approach to design. Service-orientation provides an overarching design paradigm that defines this approach, but it's the SOA design patterns that help you deal with the nitty-gritty. Regardless of whether you're applying design principles or patterns, be sure to keep the strategic vision of SOA in your sights, because it is only by the extent that you attain and maintain this vision that the success of your SOA initiative can be truly measured.

## Acknowledgements

This design pattern catalog is very much the result of an international community effort. Experts throughout the industry have contributed patterns to this catalog, and patterns were subjected to an open community review at soapatterns.org in which hundreds of members from standards organizations, major SOA vendors, and the patterns community itself participated.

The author would like to especially thank (in alphabetical order): Mohamad Afshar, Raj Balasubramanian, Frank Buschmann, David Chappell, Martin Fowler, Richard Helm, Kelvin Henney, Jason Hogg, Gregor Hohpe, Ralph Johnson, Mark Little, Brian Loesgen, David Orchard, Chris Riley, Thomas Rischbeck, Robert Schneider, Arnaud Simon, Bobby Woolf, and Olaf Zimmermann. The author would also like to thank Prentice Hall for making advance copies of the book manuscript available for distribution during the review phase.

This article contains diagrams from the upcoming book "SOA Design Patterns", a title in the "Prentice Hall Service-Oriented Computing Series from Thomas Erl" to publish in Fall, 2008 (ISBN: 0136135161, Prentice Hall, Copyright 2008 SOA Systems, Inc.). The book is currently available as part of the Safari Books Online "Rough Cut" program. For more information, visit www.soabooks.com and www.soapatterns.com.

***About the Author***

*Thomas Erl is the world's top-selling SOA author and Series Editor of the "Prentice Hall Service-Oriented Computing Series from Thomas Erl" (www.soabooks.com). With over 90,000 copies in print worldwide, his books have become international bestsellers and have been formally endorsed by senior members of major software organizations, such as IBM, Microsoft, Oracle, BEA, Sun, Intel, SAP, Cisco and HP. His most recent title ("SOA Principles of Service Design") was released in 2007, and his fourth and fifth titles ("Web Service Contract Design & Versioning for SOA" and "SOA Design Patterns") were jointly authored with industry experts and are scheduled for publication this year. Thomas is also the founder of SOASchool.com, an organization providing industry-recognized SOA training and certification.*