

The SOA Magazine

Feature Article



The Building Blocks of SOA

by Sihyung Park

Published: February 5, 2010 (SOA Magazine Issue XXXVI: February 2010)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

Abstract: Although service oriented architecture has been around for years, confusion still remains on what "SOA" truly is. For many, embarking upon learning SOA often results in too much upfront detail and information to sort through finding themselves immersed in an overwhelming spell of acronyms and related technologies. Unfortunately, the road to understanding SOA leads many into ambiguity and mental fatigue.

I've received many requests to provide a high level "forest from the trees" review of SOA. One that provides a non-technical manager (or SOA beginner) with a presentation of the fundamentals without the technical implementation details and related technologies that would otherwise confuse those who just want to learn the basics.

In this article, I will use the analogy of comparing SOA to a toy that most people (as youngsters) grew up playing with. While I could use the name of the real product I've been advised to not use the name of the product (for legal reasons) for which all credit belongs to. For the sake of this article, I will use the product name BlocGos. BlocGos is not a real product, and the name was only conjured up for the sake of creating a name for the purpose of this article and nothing beyond. The actual product that I'm speaking of, if not immediately obvious, will become so as one continues to read the article.

While this analogy is far from perfect, there are many compelling similarities between the two which can be used to help those seeking to understand the initially unfamiliar concepts of SOA in relation to the familiar design principles and concepts behind the ubiquitous memory and knowledge of BlocGos.

Introduction

In order to learn SOA one must first obtain a good understanding of a service. This article is not an attempt to completely define Services and SOA, as this is done quite effectively and comprehensively in many publications such as "Service Oriented Architecture, Concepts, Technology, and Design"[REF-1] and "SOA Principles: An Introduction to the Service Orientation Paradigm"[REF-2]. The purpose of this article is to provide an introduction to many of the fundamental concepts of SOA by using a simple analogy and association with a toy that most people of the world understand and are very familiar with.

This article will focus on a comparison of the "Service Orientation Design Principles"[REF-3] as identified by Thomas Erl with the design principles of a set of BlocGos. Then the analogy is used to compare other SOA concepts such as UDDI (Universal Description Discovery and Integration) and Service Inventory[REF-4] to their counterpart BlocGo concepts. This article is based upon a brown bag presentation that I have delivered many times within my firm, which I have found to be an effective way to help beginners to understand the basics of SOA.

The "Principles of Service Design"

For most people who grew up during the last 50 years the toys that they enjoyed playing with almost always

included a venerable box of BlocGos. The process for most children did not begin with reading a specification and instruction sheet indicating the size, shape, and description of each BlocGo. It would typically start with the child connecting two pieces, then three, then twenty, then potentially hundreds of pieces together to create toy houses, boats, trains, and skyscrapers.



Figure 1: Ancient Component Architecture

However, the process was very different for the inventor of BlocGos. The true inventor of BlocGos had an engineering challenge on his hands. His objective was to create a set of components that were combinable, standardized, abstract, loosely coupled (I'll explain later) and had other characteristics providing children a set of small building blocks with the flexibility to combine them into new creations bound only by their imagination. Thus began the process that resulted in the very first draft of the BlocGos specification! The engineering effort that the inventor of BlocGos embarked upon was not very different from the processes and challenges that other engineers faced throughout history in similar efforts. For instance, the engineers of the Egyptian pyramids also created a specification that involved many of the common design principles that underpin a set of BlocGos. Without digressing too far, the engineering principles which have continuously emerged throughout history also naturally apply to service oriented architecture. Like the BlocGo building blocks children use to create new shapes and toys, software engineers also use services in a similar fashion to create new applications. The difference in this case is that the limitations that apply to BlocGo bricks do not apply to the software realm which are not constrained by the same physical limitations as the plastic bricks (such as deconstruction of an application is not necessary just for the purpose of scavenging components).

Creating Your Box of SOA Building Blocks

Does your company have a box of services? To begin creating new applications using these services, one must first identify and create them. Similar to the box of BlocGo bricks, SOA engineers create an inventory of services that are combined into applications and made available throughout their organization. In the SOA world this is called a "Service Inventory", which is an inventory of Services that can be freely used and combined to create applications. Where does one begin and what considerations are there when embarking upon creating a Service Inventory that will maximize the well advertised benefits of a service oriented architecture (Goals and Benefits of Service Oriented Computing [REF-6])? Fortunately, a set of guidelines have already been established to help software engineers create a Service Inventory called the "Service Orientation Design Principles" [REF-7].

Service Orientation Design Principles

The "Design Principles of Service Orientation" are a set of guidelines that a SOA engineer can use when designing a service to determine the level of "Service Orientation" of the service being created. The intention

behind these design principles was originally for service design. However, these principles (in most cases) are very relevant when applied to a set of BlocGos. Without further delay, let's review the "Service Orientation Design Principles" as compared to the principles of BlocGo Design. While the application of many of the principles to a BlocGo brick are obvious, some take a little creative thought to understand the application and or analogy. The following image displays the eight design principles as they apply to BlocGos.

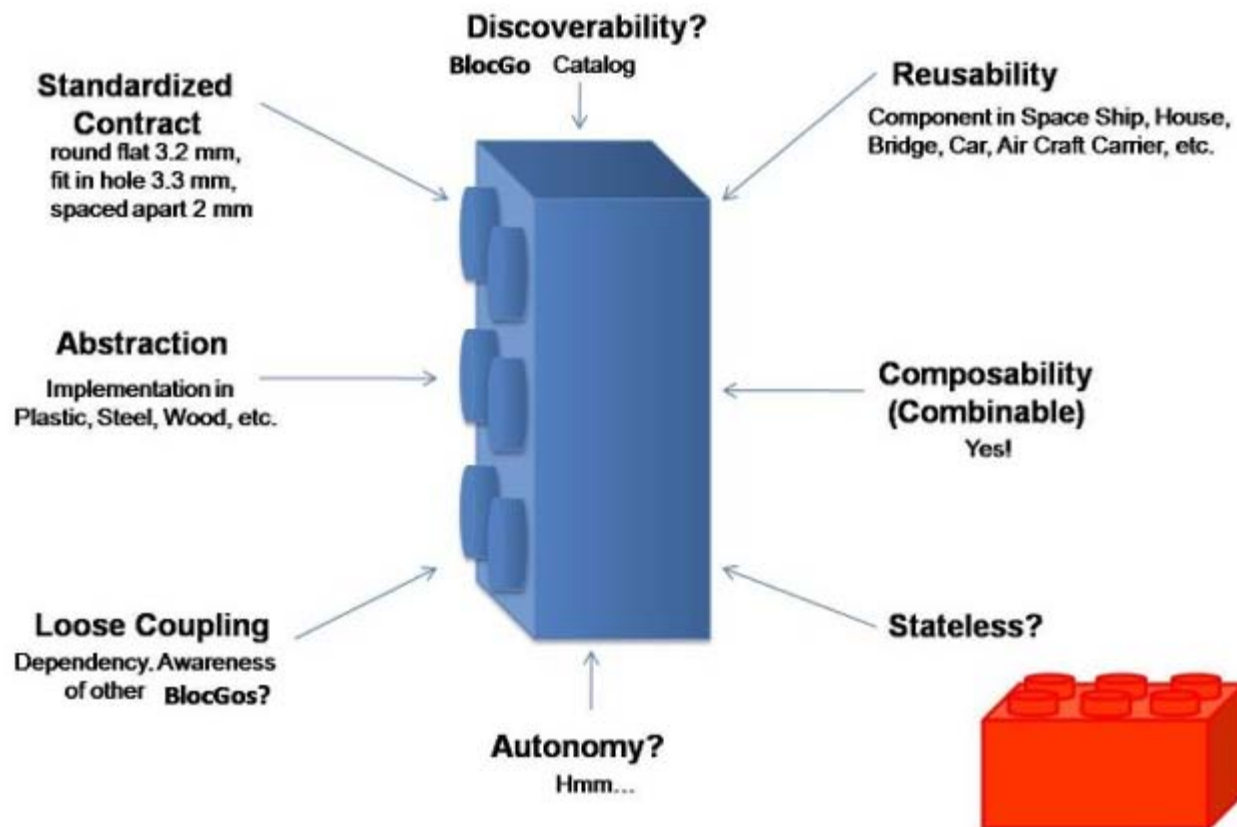


Figure 2

Standardized Service Contracts

The Standardized Service Contract refers to the set of contracts (or documented agreements) that defines and describes the functionality, data types and policies that characterize the behavior and data of a service. These contracts are created and maintained by prominent standards bodies such as OASIS, W3C, and OMG [REF-8] with members from high profile companies such as IBM and Oracle. The three primary documents that define a service, the language it speaks, and rules that apply to them are the WSDL (Web Services Definition Language), Data Model (XML Schema) files, and WS-Policy files respectively.

A BlocGo brick also has a standardized contract. Conformance to this contract is realized in the instant when one connects two bricks together. The two bricks mate together and behave as expected because all BlocGo bricks are designed with the same "standardized" contract. This contract indicates the exact size of the round male connector which perfectly mates to the back side of the other BlocGo brick's female connector. Because all BlocGo bricks follow the same contract, there is a predetermined understanding that two bricks conforming to the BlocGo specification will inherently mate together as indicated. From a SOA context, this concept is called "Intrinsic Interoperability" [REF-9]. The Standardized BlocGo Contract also indicates other dimensions that determine the shape and interface of all BlocGos to ensure interoperability. Like a BlocGo brick, a service that conforms to the "Principles of Service Design" should also inherently function and communicate with other services (Intrinsic Interoperability) because they conform to the same set of standardized contracts and rules.

Service Loose Coupling

Service Loose Coupling refers to the lack of relationship and dependency of one service to another. When a service has a high dependency with another service, environment or technology, its utilization and potential for reuse becomes limited. Like a service exhibiting loose coupling, a BlocGo brick is not dependent on another specific BlocGo brick to function. It can be freely combined ("composed") with any other type of BlocGo brick without depending on another specific one. For a services, one type of coupling to avoid is: "contract to technology coupling". This refers to the dependency of a service's contract (or interface definition) to the technology that's used to make it functional. For instance, a service's contract should not expose the fact that it is implemented using Microsoft's .NET application framework, which would be a problem if the service needed to use another technology. In the same way a BlocGo brick only needs to know that another one has the expected interface and nothing else.

Service Abstraction

The principle of Service Abstraction indicates that a service should hide as much of its implementation details as possible. The only information that a service consumer should know is what is provided in the services contract (WSDL, XML Schema, and WS-Policies). The other implementation details such as how or what is used to implement the service should not be exposed.

For BlocGo bricks, there isn't much that can be hidden. However one way to apply the Principle of Abstraction is in how the BlocGo brick contract is implemented. In this case as long as a brick is created in conformance to the contract, it doesn't matter what technology or material is used. The principle of "Abstraction" states that the implementation details of a Service should not be exposed to other services. As opposed to a service, it would be very difficult to hide the implementation details of a BlocGo brick. However, a BlocGo would function and maintain "Intrinsic Interoperability" whether it's created using plastic, aluminum, or wood. If an advanced paint were used to hide the material to create the BlocGo brick, the material (or implementation technology for a Service) used to create the BlocGo brick would be hidden. In many ways it is feasible to think that the inventor of BlocGos (possibly being a carpenter by trade) created his first BlocGo prototype using wood.

Service Reusability

The principle of Service Reusability refers to the number of ways a service can be used. High reuse is an essential characteristic for services. Reusability is also a primary factor in determining ROI (return on investment) of a service. A service that is highly reusable needs to be generic ("agnostic" [REF-10]) enough to be combined into a multitude of combinations. Like services, the principle of "Reusability" applies to BlocGos. For instance a child may use a BlocGo brick for the roof of a house but then use that exact same brick in the body of a car, in the head of a robot, or the toe of a dinosaur. The more applications that a BlocGo brick can be used for, the higher its reusability. The degree of reusability of a service determines in part the value returned by a service. If a service is not used more than once it does not need to exist.

Figure 3: The Red BlocGo brick possesses many Service Design Principles: abstractness, loose coupling, reusable, composable, etc



Service Autonomy

Service Autonomy refers to the degree of control that a service has over the resources that it uses. For instance, if a service uses a database and it shares this database with other applications, then the degree of Service Autonomy is low. However, if this service has exclusive access to the resources it uses (i.e. a database) then the degree of Service Autonomy is high. A low degree of Service Autonomy results in low consistency and/or reliability, because the existing dependencies that a service has cannot be controlled. How does this apply to BlocGos? If the factory, that makes them is also used to manufacture other toys and, if the production of both toys compete to get access to the factory then the level of "Autonomy" of BlocGos set production is lower and less predictable. The main point behind this principle is to express the idea that a service should have control of its underlying resources to maximize consistency and reliability of the service.

Service Statelessness

The principle of Service Statelessness recommends minimizing the amount of state held within a service. Excessive state held within a service can adversely impact the availability and reliability of that service. In concept, a service should not hold state because it only contributes a portion of the service's functionality. The holding and coordination of state should be conducted in the invocation between Services. When a service consumes excessive memory, the scalability of the service and the server it runs on may be adversely impacted.

Applying this principle to a BlocGo brick is not straight forward. However, another way to view state within the context of BlocGo bricks is to consider the collective identity of one brick vs. a combination of them. A BlocGo brick (like a service) is only just a generic piece until it is combined to form something like a toy boat or a house. When services are combined into an application, the collective identity is realized when it is combined with other services into an application such as an Inventory Management application, or Personnel Management application. The main point of this principle states that a service should not hold state for the sake of increased scalability and reuse, however collective identity is also a result of the state and combination of services.

Service Discoverability

Service use is highly dependent on whether it can be found and understood. Within a SOA a service is often catalogued in a service registry. A common implementation of a service registry is the UDDI (Universal Description Discovery and Integration) server. This registry is essentially a catalogue of services much like a catalog of BlocGos. A UDDI server is essentially a catalog of services with information and computer readable meta-data about the service's interface and general functionality. The service's entry in the registry provides a description of the service via standardized Service Contract (WSDL, XML Schema, WS-Policies) to help the "Service Consumer" find the appropriate service needed. In addition to the technical meta-data about the service, a human readable text description of the service should also be provided. Much like a service consumer, a child may look for a particular brick in a BlocGo catalogue containing pictures and a description of the functionality to enable a child to find the exact one needed.

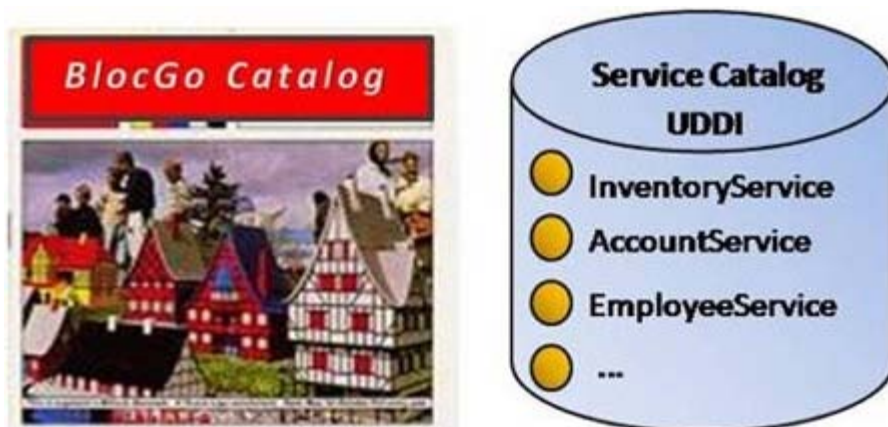


Figure 4: A UDDI Server is a catalog of Services just like a BlocGo Catalog.

Service Composability

The principle of Service Composability can also be viewed as the "combinability" of a service. SOA applications are created by combining various services together. The degree of composability determines the number of applications a particular service can be used for. The combination of different services into new applications can also be applied to BlocGo bricks. In many ways the most significant concept behind BlocGos lies in the ability to combine with others to create things. Because of the well designed shape and standard interface, the "degree of composability" of a BlocGo brick is very high.

Using the Service Design Principles

SOA engineers should use these principles as a set of guidelines when building their inventory of services. These principles were created to be used like a checklist for engineers to ask themselves before publishing them into their service inventory. Conversely, if a candidate service does not follow most or all of these principles, a redesign of the service should be considered. These principles provide a means to determine the degree of "Service Orientation" of a service. Many organizations claim to already have a SOA because they have already deployed Web services that expose some of the functionality of their applications. Using these principles, an engineer can determine if their organization's services are truly "Service Oriented".

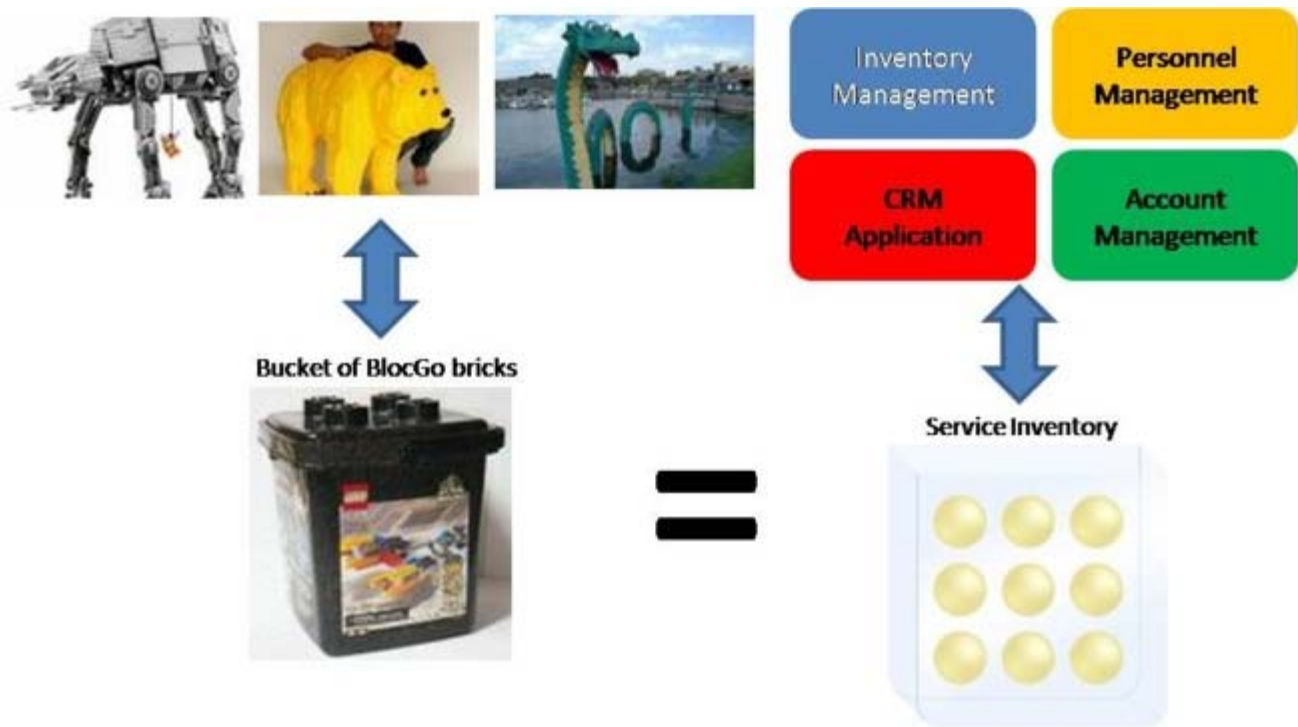


Figure 5: SOA and BlocGo bricks: What Can You Build?

Conclusion

Knowing what makes a service "Service Oriented" is crucial to understanding the core concepts that underpin a service oriented architecture. It is essential for a SOA engineer to consider these "Principles of Service Design" when designing new services and to determine the degree of service orientation of the existing services within their organization's service inventory. Service oriented architecture is a very broad discipline which also incorporates many technologies and concepts that predate it (Object Oriented Programming, Corba, Software Design Patterns, J2EE, XML, etc.). However, understanding the principles of service design should be the first among the concepts considered when one endeavors to learn SOA. Throughout history the lessons of flexible component design re-emerge regardless of discipline. One can imagine that many of the same principles that apply to SOA were also used to create (and provided inspiration for) the engineering principles used by the ancient architects of the Egyptian Pyramids and more recently by the true inventor of BlocGos. In this article I presented the principles of service design and compared them to the design principles of BlocGos. The analogy and comparison between BlocGo bricks and SOA is not perfect, considering that BlocGos are constrained by the rules that govern objects in the physical realm while SOA exists in a realm that is somewhere between the physical and conceptual. Thus, many of the comparisons require a creative imagination to follow some of the parallels and concepts presented. However, I hope that this has helped you to understand the basic concepts of a "service" and enables you to have a better understanding of Service Oriented Architecture. In closing, I would like to say good luck in your SOA endeavors and "leg - godt" which means "play well" in Danish.

References

- [REF-1] <http://www.soabooks.com/>
- [REF-2] <http://www.soabooks.com/>
- [REF-3] <http://www.soaprinciples.com/>
- [REF-4] Service Inventory <http://www.whatissoa.com/p13.asp>
- [REF-5] http://www.soapatterns.org/enterprise_inventory.php
- [REF-6] <http://www.whatissoa.com/p16.asp>
- [REF-7] <http://www.soaprinciples.com/default.asp>
- [REF-8] OMG: <http://www.opengroup.org/>, W3C <http://www.w3.org/>, OASIS <http://www.oasis-open.org/home/index.php>

[REF-9] http://www.whatissoa.com/increased_intrinsic_interoperability.asp

[REF-10] http://www.soapprinciples.com/service_reusability.asp

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



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

Copyright © 2006-2010
SOA Systems Inc.