

The SOA Magazine

Feature Article



Service-Oriented and Object-Oriented Part I: A Comparison of Goals and Concepts

by Thomas Erl

Published: February 9, 2008 (SOA Magazine Issue XV: February 2008, Copyright © 2008)

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

Abstract: It's no big secret that one of the primary influences of service-orientation has been the well established object-oriented design paradigm. Though there are distinct differences, there is also much common ground between these two design philosophies. In fact, if it wasn't for the innovative design principles and patterns formalized by object-orientation, the service-oriented architectural model and the Web services framework would not exist as they do today.

This two-part article series studies object-orientation and service-orientation by providing a comparison of goals, concepts, and principles. Both articles are comprised of excerpts from the book "SOA: Principles of Service Design" [REF-1].

Note: The terms "object-oriented analysis and design" (OOAD) and "object-orientation" are used interchangeably in this article. Also, the upcoming sections contain references to service-orientation design principles that are not explained. If you are new to service-orientation, visit SOAPinciples.com [REF-2] for introductory descriptions of each of the eight design principles.

Introduction: A Tale of Two Design Paradigms

Object-oriented analysis and design was responsible for popularizing the vision of building streamlined applications comprised of reusable, flexible software. Further supported by the sophisticated processes and conventions of the unified modeling language (UML) and a set of classic design patterns that changed the face of distributed application design, object-orientation evolved into a well-rounded and mature design framework.

OOAD originally grew out of a need to bring order to unstructured development processes that had resulted in various problems, including the creation of the notorious spaghetti code. It drew from best practices that emerged from procedural programming approaches and combined these with a design philosophy that aimed to shape software into units that more closely mirror the real world.

Object-orientation aspires to maximize the fulfillment of business requirements throughout the lifespan of an application, including its post-deployment upgrades and extensions. It provides numerous rules and guidelines that govern the careful separation of application logic and data into objects that can be individually maintained to help minimize the impact of change on the application as a whole.

Many of the UML conventions and documentation techniques further provide a comprehensive means of expressing customer requirements and predictable runtime application behavior. Collectively, families of UML diagrams and specifications combined with established principles and practices help designers ensure that applications are built to be both robust and flexible. Also on the agenda for object-oriented applications is the fostering of reusable code. Key techniques, such as inheritance and polymorphism (discussed in Part II of this article series) are positioned to allow different software programs to benefit from logic already created for others.

As established in the upcoming *A Comparison of Goals* section, service-orientation shares many of the same goals as OOAD. It seeks to establish a flexible design framework that allows for the agile accommodation of ever-changing business requirements. Much like OOAD, service-orientation is very concerned with minimizing the impact of change upon software programs already deployed and in use. Principles such as Service Loose Coupling and Service Composability, for example, address long-term governance requirements so as to allow implemented services to continue to evolve in tandem with the business.

A common distinction between the two design paradigms is one of scope (Figure 1). While object-orientation never explicitly limits the extent to which its principles can be applied, in real world environments, they have commonly been realized within single applications or collections of related applications. When reuse was attained, it was often at the utility level resulting in libraries of "common components" shared by custom-developed applications.

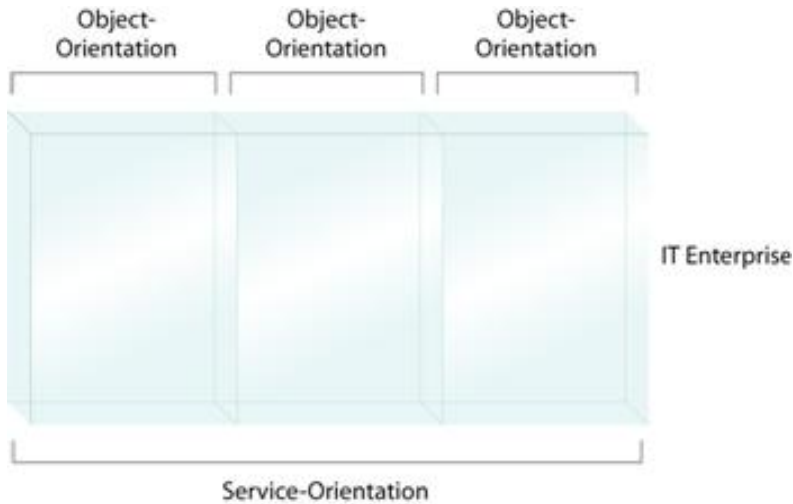


Figure 1: Historically, object-orientation has been applied to segments of the enterprise. Service-orientation aims to harmonize a larger amount of the enterprise or, ideally, the enterprise as a whole.

Additionally, several of the object-oriented design principles and patterns were developed during a time when the majority of IT enterprises were building componentized or distributed applications using RPC technology. The reuse potential of any given object was limited to the boundary of the RPC platform. In larger environments comprised of various technology platforms, an RPC implementation therefore represented a specific architectural zone. To enable connectivity with other zones required bridging or integration technologies. The increased demand for cross-application and cross-platform connectivity led to the emergence of EAI (which, incidentally, is another major influence on service-orientation).

Although they have many roots in object-orientation, SOA and service-orientation owe their current mainstream status to the emergence and successful adoption of the Web services framework. Even though the feature-set provided by the first generation Web services platform was primitive at best, it established the potential to break through proprietary application and platform boundaries so as to inspire visions of true, cross-enterprise inter-connectivity and federation.

The architectural model that underlies SOA and the principles behind service-orientation were all developed in support of this vision. As a result, they have a great deal of synergy with the maturing second-generation Web services platform. Figure 2 illustrates how OO, EAI, and Web services, along with BPM comprise the major influences of service-orientation.

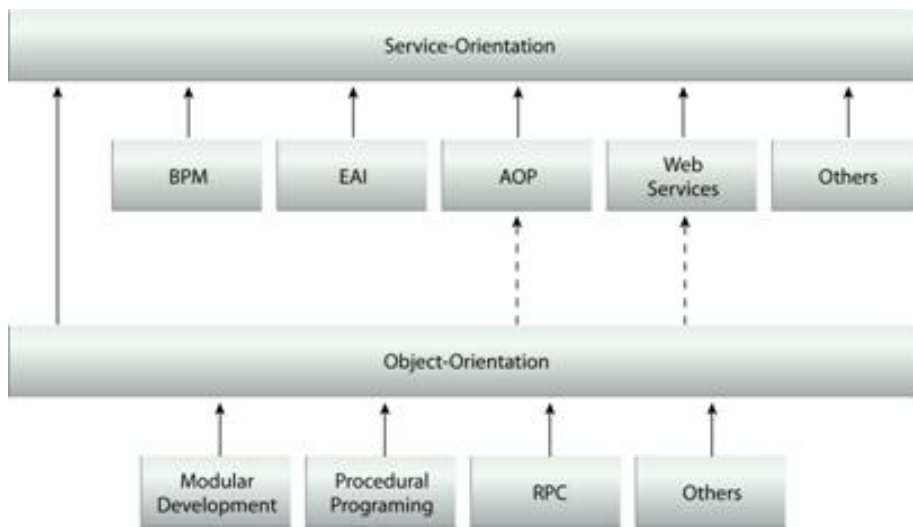


Figure 2: While object-orientation evolved out of approaches that included procedural programming, service-orientation builds upon the object-oriented design paradigm and, together with additional influences, establishes a distinct paradigm of its own.

Within service-orientation, solution logic designed as services is intentionally positioned as enterprise resources and sometimes even enterprise-wide resources. This enterprise-centric perspective is one of the main reasons that only a subset of object-orientation principles were carried over into service-orientation (as explained in Part II of this article series).

A Comparison of Goals

Before we begin comparing concepts and principles, it is important that we establish the fundamental objectives behind each of these design approaches. The strategic goals associated with service-oriented computing are as follows:

- Increased Intrinsic Interoperability
- Increased Federation
- Increased Vendor Diversification Options
- Increased Business and Technology Domain Alignment
- Increased ROI
- Increased Organizational Agility
- Reduced IT Burden

WhatIsSOA.com [REF-3] describes each of these goals, so we won't repeat them here. Several of these are in direct alignment with the original goals of object-orientation. Some, however, differ in that they are specific to service-orientation's enterprise-centric scope.

The next set of sections explore the following common OOAD goals and discusses how they compare and relate to service-orientation principles and goals:

- Increased Business Requirements Fulfillment
- Increased Robustness
- Increased Extensibility
- Increased Flexibility
- Increased Reusability and Productivity

To better understand how service-orientation relates to and supports these particular object-orientation goals (Figure 3), we need to take a closer look at each.

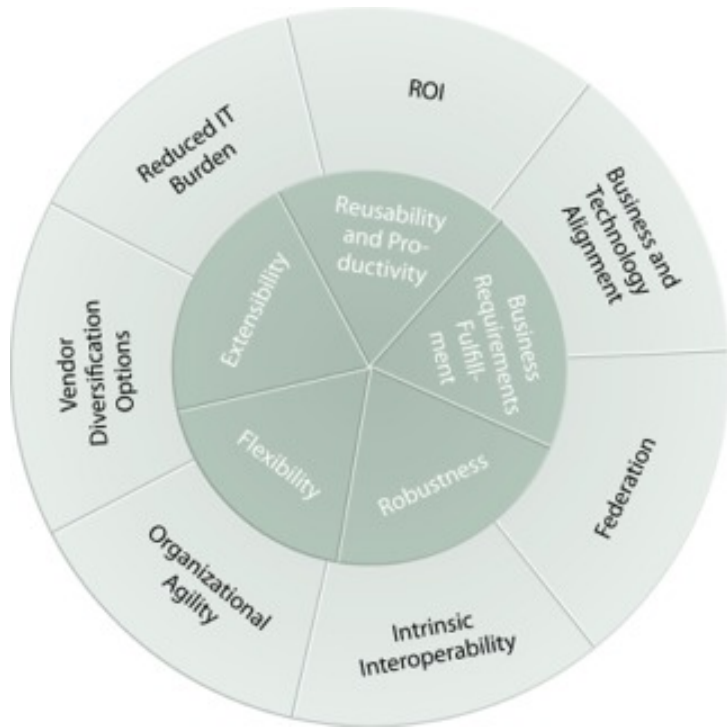


Figure 3: Service-orientation inherits all of the primary OOAD goals (inside circle) but further increases their scope and adds others (outside circle).

Increased Business Requirements Fulfillment

Through specialized analysis and design techniques (that include business-centric deliverables such as use cases) OOAD advocates the design and development of applications more capable of meeting specific business needs.

Increased business requirements fulfillment is also a priority for service-orientation and a primary design consideration for all of its principles. Many of the design characteristics fostered by these principles are geared toward enabling a design-time process that allows for the creation of sophisticated composition configurations in response to a range of envisioned business requirements.

Strategic service-orientation goals, such as "increased vendor diversity options," are intended to establish an environment that empowers an organization to continuously leverage technology innovation in support of maximizing the fulfillment potential of business requirements. Furthermore, OOAD's emphasis on partitioning solution logic into units that more closely resemble real world objects is in alignment with the "increased business and technology domain alignment" goal which aims to incorporate real world representation at domain and enterprise levels.

Increased Robustness

Object-oriented solutions can be delivered to withstand a range of exception conditions because of extra design considerations applied to the various parts (objects) that comprise the solution and due to the use of formal design-time deliverables, such as activity, sequence, and state diagrams, that map out potential runtime usage scenarios.

Increased robustness is a goal of service-orientation from both short-term implementation and long-term governance perspectives. Service compositions are expected to work as required in their immediate deployment, but are also designed to remain robust as their individual members are repurposed in support of fulfilling different business requirements as part of different compositions.

Service Autonomy and Service Statelessness represent two key principles dedicated to ensuring services are reliable and scalable during their runtime existence while concurrently supporting the automation of multiple solutions.

Increased Extensibility

Once implemented and in use, an object-oriented solution's functional scope can be increased without requiring significant redevelopment by leveraging the componentized nature of its application design.

Several service-orientation principles aim to establish the freedom for service compositions to be extended or

recomposed in response to an increase in business requirements scope. The fact that functional contexts are carefully modeled and defined for each service allows for individual service contracts to be cleanly extended with new capabilities and without disruption to existing consumer programs.

The goals of "increased federation" and "increased intrinsic interoperability" aim to harmonize an enterprise in which solutions comprised of service compositions can be modified and extended through the incorporation of new service capabilities with minimal impact (due to the native compatibility established by standardization in support of intrinsic interoperability).

Increased Flexibility

After an object-oriented solution is deployed it can be further evolved and enhanced with minimal disruption to its users through the targeted application of key design techniques, such as encapsulation, abstraction, and inheritance.

Enabling an organization to freely govern and evolve a service is a prime concern of the Service Loose Coupling and Service Abstraction principles, both of which protect an enterprise from the proliferation of unhealthy dependencies. This establishes an environment in which individual service capabilities can be refactored and enhanced as required.

The flexibility to augment services individually carries over to an increased flexibility to evolve a service inventory and the underlying service-oriented architecture itself. Flexibility, in fact, is at the heart of the "increased organizational agility" goal.

Increased Reusability and Productivity

Object-oriented solution logic can be designed for reuse thereby lowering the subsequent effort to build applications that require the same type of logic. The Service Reusability principle clearly corresponds to this goal, but it is worth mentioning that all of the other service-orientation design principles are also positioned to fully support the widespread realization of reusable service logic.

As a result, reusability, as part of SOA, becomes more of an expected, secondary design characteristic than an actual objective. The goal of "increased ROI" is closely associated with the successful application of this principle.

A Comparison of Fundamental Concepts

Conceptually, object-orientation and service-orientation have similarities, but they are not the same. The upcoming sections establish the following terms and definitions used by each design approach in relation to both common and differential concepts:

- Classes and Objects
- Methods and Attributes
- Messages
- Interfaces

Note that the examples in the upcoming sections use UML conventions. Various approaches have emerged for applying UML to the design of XML schemas and Web services. The focus of this article is not on how to adapt UML conventions to express XML schema or WSDL definition structures. Unless otherwise indicated, this comparison is specifically about fundamental service-orientation and object-orientation, which naturally raises issues as to how UML relates to services (regardless of their implementation).

Classes and Objects

Object-orientation provides a means of organizing solution logic into *classes* (Figure 4) that essentially act as containers for definitions of related behaviors and properties.

A runtime instance of a class is an *object* (much like a runtime instance of a service is a service instance). Therefore, a class can be seen as a design template from which various objects are spawned, each with their own unique runtime state and data.

A class is comparable to but not equivalent to a technical service contract. A class can define a combination of public

access and private implementation details, whereas a service contract only expresses public information. In this regard, a service contract more closely resembles an interface implemented by a class (as explained in the *Interfaces* section).



Figure 4: The class symbol (left) and the chorded circle symbol (right) both establish a container and a functional context associated with Invoice-related functionality.

Methods and Attributes

Object-oriented classes define *methods* and *attributes* so as to associate behavior and data with objects. Behaviors represent functionality the class is capable of carrying out. Each behavior is expressed and described by an individual method definition. Methods are sometimes also referred to as operations; however, the term operation has now become more synonymous with the use of Web services.

Class properties represent a form of predefined state data associated with the class, and are expressed through the definition of attributes. Attributes can also be referred to as variables.

Methods and attributes can be declared as private or public to the class. It has become a best practice to only allow the public access or modification of attributes via public methods (further qualified as "accessor methods").

Services express behaviors as capabilities in abstract. A capability is the equivalent of a method if a service is implemented as a component and an operation if the service is deployed as a Web service. A Web service contract cannot define private operations. Due to the emphasis on statelessness, service contracts are discouraged from defining attributes, as shown in Figure 5.



Figure 5: The class symbol (left) expresses an attribute and a method, whereas the service symbol (right) only defines a capability.

Messages

Communication between the invoker of an object and the object whose method is invoked is carried out through the exchange of *messages*. This is an abstract term used as part of the OOAD vocabulary, and therefore does not imply how a message is physically comprised in the real world.

Because object-orientation is typically applied to components that rely on non-industry standard (often RPC-based) communication protocols, messages are most commonly expressed as binary units of communication that are exchanged synchronously. The contents of a message depend on the data type of the input or output values defined as part of the method and the supporting technical platform. RPC platforms support a variety of data types including types that can represent objects themselves.

Messages used by services implemented as Web services typically manifest themselves as text-based units of communication that can be exchanged synchronously or asynchronously. In this context, they are messages in a more traditional sense (as used by e-mail systems or messaging-oriented middleware).

The input and output values of Web service operations are represented by messages that are usually structured by XML schema complex types. They can have document-centric complex type hierarchies comprised of numerous values, each with a different data type. This is why the base chorded circle symbol expresses service contracts without specifying data types.

Object methods are frequently designed to exchange fine-grained parameter data. This is because the connection they establish with other objects (whether local or remote) is generally persistent. Once in place, data exchange is efficient and reliable.

Web services commonly rely on the stateless HTTP protocol to exchange messages. Because they do not have the benefit of a persistent, stateful connection, operations often need to be designed to exchange document-centric messages; messages comprised of larger amounts of data, such as entire business documents. Almost every service-orientation design principle can impact the size of service messages by influencing capability, data, and validation granularity.

Figure 6 illustrates how the design of a class can be affected differently when shaped by object-oriented and service-oriented principles and further contrasts this with a typical service contract. Note the differences in method and operation granularity across these three samples; service-oriented design encourages the addition of coarse grained capabilities that are more message-centric and support the exchange of XML documents. This affects both the granularity of capabilities as well as the choice of data type.



Figure 6: An object-oriented class (left), a service-oriented class (middle), and a service contract (right). Note that the attributes omitted by the middle class are public.

Interfaces

Collections of related methods can be defined (but not implemented) within *interfaces* (Figure 7). A class can then be designed to implement an interface, thereby establishing a formal endpoint into the logic encapsulated by the class. In this role, the interface can abstract additional details about the class from the outside world.

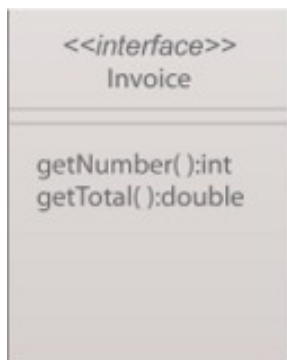


Figure 7: An Invoice interface expressing two methods.

Service-orientation is focused on both the definition of the service contract and its underlying solution logic. A service contract on its own is comparable to an implemented class interface, in that it provides the official entry point for

publicly available service functionality while also abstracting underlying service details.

Unlike a class which exposes its attributes and methods (with or without the use of an interface) as an embedded extension of itself, a service contract exists as a physically decoupled architectural component when implemented as a Web service.

A Web service contract can be viewed as a potentially sophisticated form of technical interface in that it is capable of expressing a range of logic, including data exchange requirements, validation rules, and even semantic policies in addition to implementation details, such as ports and bindings.

The WSDL definition used to define a Web service contract contains a `portType` element construct that formally establishes the Web service operations. In this regard, a Web service `portType` is a lot like an object-oriented interface (in fact, the `portType` element is renamed to `interface` in WSDL version 2.0). Note that a WSDL definition can contain multiple `portType` constructs, much the same way as a class can implement multiple interfaces.

A service that exists as a Web service may or may not encapsulate object-oriented logic. If it does, then service-orientation design principles can affect the manner in which classes are designed, as explored in Part II of this article series.

Conclusion

It is important to keep in mind that object-orientation and service-orientation are complementary design paradigms. The more we understand their differences, the better we can assess when to use them separately or how to apply them successfully together. In Part II of this article series we will continue this comparative study by contrasting the design principles that essentially comprise these two paradigms.

References

[REF-1] "SOA: Principles of Service Design" by Thomas Erl, <http://www.soabooks.com>

[REF-2] SOAPrinciples.com ("An Introduction to the Service-Oriented Paradigm"), <http://www.soaprinciples.com>

[REF-3] WhatIsSOA.com ("An Introduction to Service-Oriented Computing"), <http://www.whatissoa.com>

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



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

Copyright © 2006-2008
SOA Systems Inc.