Introduction

The paradigm of service-orientation consists of eight principles [REF-1] in support of the business goals we covered in Part I of this article series:

- Standardized Service Contract
- Service Loose Coupling;
- Service Abstraction;
- Service Reusability;
- Service Autonomy;
- Service Statelessness;
- Service Discoverability; and
- Service Composability.

Each of these principles does its share in driving design and development activities to create the target state of a service inventory architecture. We now describe each individually, and further highlight how each principle relates to specific business goals.

Standardized Service Contract

“Services within the same service inventory are in compliance with the same contract design standards.”

Services express their purpose and capabilities via their service contracts. The Standardized Service Contract design principle requires that inventory-specific standards and processes be specified by the enclosing inventory’s governance board and taken into account when designing a service’s contracts. These considerations impact the nature and quantity of content that will be published as part of a service’s official contract.

Emphasis is placed on the manner in which services express functionality, how media types are defined, and how policies are asserted and attached to contracts. There is a focus on ensuring that service contracts are optimized, appropriately granular, and standardized to ensure that the endpoints established by services are consistent and governable.

The standardized service contract design principle helps to produce services that are intrinsically interoperable by, for example, insisting that services are:

1. described in a consistent and easy to understand form;
2. use consistent request and response patterns;
3. use consistent method names; and
4. use consistent media types is constrained to avoid unnecessary data transformations between services.
The Standardized Service Contract principle supports the following strategic goals:

- **increased intrinsic interoperability** – common service contract design standards encourage consistency across services that drive improved interoperability.
- **increased federation** – having a common way of defining and interpreting contracts encourages reuse of services and increases levels of federation.
- **increased vendor diversification options** – by applying a common design and review process for service contracts we can make explicit decisions about elements of contracts that could inadvertently increase reliance on particular vendor platforms.
- **increased alignment of business and technology domains** – The standard service contract design process applied under this principle involves both technology and domain experts. Experts from the business are able to shape the contracts that are defined and improve their understanding of the resulting information technology capabilities.
- **increased return on investment** – contracts designed under a common methodology produce services that are more reusable, leading to actual reuse and increased returns.
- **increased organizational agility** – common contract design standards and processes allow for more normalized representations of data and functionality, resulting in greater ability to reuse existing services to rapidly build new applications.
- **reduced information technology burden** – common contract design standards result in normalized ways to express functionality and standardization of media types. This reduces the need for special converter logic within a service inventory and increases reuse. Both of these properties reduce the amount of redundant logic and the cost of maintaining this logic.

**Service Loose Coupling**

“Service contracts impose low consumer coupling requirements and are themselves decoupled from the surrounding environment.”

Service loose coupling is about minimizing the assumptions about service by service consumers so that the impact of changes across the architecture is minimized. It is often related to the expression of service contracts, and the decoupling of those contracts from specific features of their environment. It also covers coupling within the service itself. Several common types of coupling to avoid include:

- coupling of a service contract to the implementation logic or other implementation details, such as when a contract and corresponding media types are automatically generated from software in ways that are prone to change when the implementation changes;
- coupling of a service contract to the implementation technology, including the use of language-specific (such as, Java remote method invocation) or platform-specific (such as, Windows distributed component object model) communication mechanisms;
- coupling of a service contract to the needs of specific service consumers or specific parent processes in ways that reduce its reusability as part of other tasks; and
- coupling of a service consumer directly to service implementation details, bypassing the service contract to find more straightforward or more performant means of access to service logic and data.

Loosely-coupled services define a contract that is the only means by which consumers of the service can invoke its capabilities. The contract is defined in terms of technology-neutral message elements such as SOAP, HTTP, and XML rather than technology-specific elements such as Java RMI or .NET remoting. In addition, any
endpoint identifiers should avoid ties to specific physical locations or specific server hosts.

Starting with a basic level of loose coupling and a common approach to defining contracts across a service inventory reduces the need for complex integration exercises and allows services to be reused more readily. Loose coupling accommodates changes to underlying technologies by avoiding any references to these technologies in contract definitions.

The Service Loose Coupling principle supports the following strategic goals:

- **increased intrinsic interoperability** – reduced coupling encourages freer exchange of information and functionality across the applications of an information technology portfolio.
- **increased federation** - by reducing coupling we can encourage reuse of services and increased dependence on reusable services.
- **increased vendor diversity options** - this principle encourages vendor-neutral technology foundations for defining service contracts.
- **increased return on investment** - looser coupling results in the ability to support service consumers with a wider range of needs, increasing reuse and returns
- **increased organizational agility** - freedom to evolve service implementations independently of service consumers results in a greater ability to respond quickly to changes as they arise, compared to a whole of application upgrade.
- **reduced IT burden** - loose coupling keeps costs localized where changes need to be made, and limits the cost of making them when required.

**Service Abstraction**

“Service contracts only contain essential information and information about services is limited to what published in service contracts.”

Service Abstraction goes beyond the basic loose coupling principle to look at the specific details of each contract in exposing service capabilities, and at the underlying capabilities themselves. Four basic types of information are identified as beneficial to hide or abstract in order to allow details to change without impacting service contracts and consumers:

- **technology information** – Information about the technologies in use within a service can leak through to service consumers, and even if they are not explicitly stated in a contract can influence the expectations of developers about the underlying behavior and performance of a service
- **functional information** – It can be tempting to expose every available function or method within a service as part of its published contract, however exposing unnecessary or unused functionality can make future refactoring of the service difficult and require major surveys of service consumers to ensure features are not being used before they are removed from public contracts
- **programmatic logic information** – Information about internal algorithms and data structures in use can often be abstracted to simply state the pre-conditions and post-conditions of a given service capability, or by stating a guaranteed computational complexity of an algorithm. This allows greater freedom to change implementations over time.
- **quality of service information** – over specifying qualities of service such as availability, response times, and throughput can lead to over promising on performance requirements or can lock the service implementation down in the ways it can solve problems as they arise. By focusing on real and high-level quality of service concerns we allow the service owners greater freedom to build the most effective solution.
Service loose coupling hides information about the implementation and environment of a service. Service abstraction also hides details about the service logic itself in order to allow the logic and its implementation to change over time independently of its consumers without changing the service contract. Applying the service abstraction principle requires expert analysis of business application requirements in order to find the most suitable abstraction level for each service.

Only information necessary for service consumers to invoke capabilities and perform their functions is exposed in contracts, leaving services free to change implementation details that the client does not need to know about. This principle acts as a counter-weight to the later service discoverability principle which advocates supplying meta-data about the service to developers of applications so that they can easily understand the service capabilities and effectively reuse them.

The Service Abstraction principle supports the following strategic goals:

• **Increased intrinsic interoperability** – abstracting further from implementation details and encouraging greater reliance on the published service contract reduce costs in ensuring unbroken long-term interoperability.

• **Increased federation** – improved long-term compatibility increases consistency and uniformity across the enterprise.

• **Increased vendor diversity options** – this principle encourages services to shield their consumers from vendor-specific implementation details that might otherwise emerge through their contracts.

• **Increased return on investment** – accessing services only through the published service contracts ensures that service interfaces are well understood and can be supported through internal refactoring and other changes required to support a diverse range of service consumers.

• **Increased organizational agility** – limiting communication to the service contract ensures that services can be modified reliably when required by the business.

• **Reduced information technology burden** – a solid definition of the interface between a service and its consumers allows changes to be made more reliably with less spill over into other information technology assets.

**Service Reusability**

“Services contain and express agnostic logic that can be positioned as reusable enterprise assets.”

Reuse is strongly advocated as a core part of typical service analysis and design processes. This principle emphasizes the positioning of services at a level of abstraction that supports reuse across a number of different business applications, as well as an organizational culture and supporting development processes that bias project teams towards reuse of existing service assets.

Functionality is only built into services if it is a genuine candidate for reuse. Application-specific logic is split from this core inventory of services and is explicitly planned to be disposed of at its end of life. The services that make up the reusable portion of the application are carefully designed to consider:

• how this logic will fit with existing reusable services in the service inventory;

• current business requirements and use cases;

• the timeline of when actual functionality needs to be delivered;

• trends and predictable changes in the organization that may impact the service; and

• predictable changes in the technology mixed within the enterprise that may impact the service.
Service reusability is the notion that a service should be built for, or at least planned for, future demands that may be placed upon it. Today’s application may only require a capability to update a value, but tomorrow’s application will very likely need to query that value. While it is important to avoid wasteful gold plating of service capabilities, it is also important to make sound judgments about the capabilities likely to be required in the future and either implement them or accommodate their future addition.

The Service Reusability principle supports the following strategic goals:

- increased intrinsic interoperability – services that exhibit genuinely reusable logic and information drive higher levels of interoperability across a service inventory as value is delivered.
- increased alignment of business and technology domains – the process of identifying reusable logic to encapsulate in a service involves both business and technology experts.
- increased return on investment – actual reuse produces increased actual returns.
- increased organizational agility – reusable services form a base of pre-existing logic and data that new applications can be built up on quickly and efficiently.
- reduced information technology burden – a focus on reusability reduces redundant implementation of functionality and reduces the costs of maintaining duplicate implementations.

Service Autonomy

“Services exercise a high level of control over their underlying runtime execution environment.”

Services require a significant degree of control over their environment in order to carry out their capabilities consistently and reliably. The autonomy principle encourages service designers to consider carefully whether dependencies on other services should be introduced and to avoid dependencies on shared infrastructure or on other unnecessary parts. A number of levels of autonomy are considered under this principle:

- shared logic – the service shares some core service logic with other services, as well as data and infrastructure.
- shared infrastructure – the service logic is separated from other services, but it still relies on common infrastructure such as common databases or hardware.
- dependent – the autonomy of the service is dependent on the autonomy of other services it composes.
- autonomous – the service logic and infrastructure is specific to the given service, and can be scaled, maintained and refactored independently of other services.

Databases that are shared between services, or complex deployments of multiple services hosted on a shared server can harm both performance and scalability. Autonomy may initially only be planned for, rather than being an immediate priority for implementation. The allocation of a virtual machine with limited and shared resources may be all that is required to meet initial demands. As the service is reused more and more these demands grow, and the virtual machine can be re-provisioned with more resources and ultimately moved to its own dedicated hardware or load-balanced cluster.

The Service Autonomy principle supports the following strategic goals:

- increased intrinsic interoperability – greater levels of autonomy reduce cost and risk in depending on a given service for the lifetime of an application.
- increased vendor diversity options – increasing the autonomy of a service from the implementation of other services allows its technology foundations to be replaced more readily when needed.
- **increased alignment of business and technology domains** – the needs of the business and expectations of change are actively considered in determining the initial and ongoing hosting arrangements of services.

- **increased return on investment** – greater autonomy increases the reliability of a service and its ability to accommodate future changes. This encourages greater reuse and greater return.

- **increased organizational agility** – changes to hosting arrangements and other changes required by the business can be more readily accommodated by autonomous services.

- **reduced information technology burden** – greater reliability and predictability reduces support costs associated with services.

This concept seeks to find ways that business and information technology specialists can work most effectively together and develop a common language for framing business problems and solutions.

**Service Statelessness**

“Services minimize resource consumption by deferring the management of state information when necessary.”

The management of excessive session state information can compromise the availability of a service and undermine its scalability potential. Session state is any information that a service needs to hold on to on behalf of active service consumers. It differs from other kinds of information held by the service because it grows as the number of concurrent consumers increases, and falls when the number decreases. When there are no active service consumers, the session state information can be empty.

Services automatically need to manage some amount of session state while they are handling a request message. They need to store the message, typically in memory. They need to process it, and they need to respond to it. These activities all consume some memory resources.

Session state information can extend out beyond the processing of a single request. For example:

- A sequence of several requests may refer to the same session state (such as, “query for information” or “give me the next set of results”).

- A service may need to call out to other services or perform other asynchronous processing as part of handling an incoming request; a service that automates the business process of recording a business transaction may need to make requests to the invoice service.

If a large number of service consumers all make concurrent requests on a service, the service could easily run out of memory capacity in handling those requests. It may not be able to free up that memory until its current set of consumers have finished their sessions or until it has completed its calls out to other services. A service that is running close to capacity will fail to process new incoming requests until it has the ability to free up resources to cope with them. This can compromise the availability of a service and the management of excessive session state information can undermine its scalability potential.

Reducing session state information or reducing the time that a service needs to hold on to state information allows services to deal with greater numbers of concurrent consumers and improves their availability. For example:

- Sequences of requests are designed as far as possible to capture the information required as part of each message to avoid needing to hold on to session state (such as, “query for information set 1” or “query for information set 2”).

- As much information as possible associated with the processing of a given request is deferred to databases or included in messages while invoking the capabilities of other services or while processing of a request is paused for any reason.
Services are designed to remain stateful only when required, deferring session state to databases or back to service consumers as appropriate.

The service statelessness principle supports the following strategic goals:

• *increased intrinsic interoperability* – the improved scalability and availability that result from stateless design make service level agreements easier to establish and meet, reducing costs and risks in incorporating services into application design.

• *increased return on investment* – improved scalability and availability result in greater reuse and greater returns.

• *increased organizational agility* – increased scale can be accommodated more readily by stateless services than their stateful counterparts.

• *reduced information technology burden* – scalable services require less corresponding hardware and other support infrastructure investment than their stateful cousins.

**Service Discoverability**

“Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.”

For services to be positioned as reusable IT assets with repeatable return on investment they need to be easily identified and described so that it can be understood when opportunities for reuse arise. Developers that are looking for a service to reuse need to be able to clearly connect their requirements to the capabilities of pre-existing services in order to understand whether or not they can be used immediately and also to request changes if they are required. The service design needs to take into consideration the quality of information available to project teams who may reuse the service, and explicit registries or other mechanisms for discovering existing services may also be used.

This principle uses the same kinds of descriptive metadata as we covered under the service abstraction principle. The key is two forms of information that we want to provide as part of the service metadata are functional and quality of service information, as technology and programmatic metadata are likely to have been abstracted out of the service description.

A cultural expectation of reuse and a level of inventory-wide governance is also required under this principle. Development teams must look for existing implementation rather than leaping to develop their own implementation of solution logic. Governance activities must coordinate any required service changes to avoid unnecessary duplication of logic.

The information published in a service registry contributes to governance activities that control the set of services within a service inventory. The service metadata has enough detail to understand how the service fits within the defined service inventory blueprints (if any), and to avoid the implementation of duplicate service logic or functions. The Service Discoverability principle supports the following strategic goals:

• *increased intrinsic interoperability* – the first step in interoperating with a service is often to simply know it is there and to understand its service contract.

• *increased alignment of business and technology domains* – services are discoverable by both technology and business experts. Where business experts need assistance in understanding the capabilities of their information technology infrastructure, technology experts are able to access detailed information and report effectively.

• *increased return on investment* – greater accessibility of service metadata results in greater reuse and greater returns.
• *increased organizational agility* – greater opportunity for reuse of existing services results in developers being able to construct new applications more quickly.

• *reduced information technology burden* – greater accessibility to service contract information reduces the cost of disseminating information as well as the costs of rework arising from misunderstandings when accurate and timely information is not available.

**Service Composability**

“*Services are effective composition participants, regardless of the size and complexity of the composition.*”

The ability to effectively compose services is a critical requirement for achieving the strategic goals of service-orientation by enabling the actual reuse of services. Enabling complex compositions requires design considerations be built into services over the lifetime of the service inventory that are difficult to retro-fit when the time comes to build the composition. This principle acts as an overarching guide to support the interpretation of other principles of service-orientation, providing a context within which principles can be traded off against each other. When deciding how much information to include in a service contract versus how much to abstract for example, the service should be considered from the point of view of a composition that will include it over the lifetime of that composition. At every point in the analysis and design process, both project teams and governance bodies should consider the composition implications of their design decisions.

The Service Composability principle supports the following strategic goals:

• *increased intrinsic interoperability* – a high degree of composability requires a corresponding high level of interoperability. Focusing on composability drives attention to improving interoperability.

• *increased alignment of business and technology domains* – the ability to assemble services into many different compositions in support of new applications is a key differentiator between service-orientation and more conventional monolithic approaches to application development.

• *increased return on investment* – increased composability means that services can be configured together in more ways and at greater levels of complexity, increasing returns without increased cost.

• *increased organizational agility* – having an inventory of services that are pre-built to consider the needs of complex compositions means that these compositions are straightforward to implement when required.

• *reduced information technology burden* – the reuse opportunities opened up by building readily-composed services reduces redundant logic implementation and the costs of maintaining the duplicate logic.

**Conclusion**

This is the second part of a three-part article series dedicated to service-orientation. The third and final article will describe SOA architecture types.

**References**

About the Author: Raj Balasubramanian

Raj Balasubramanian is an Enterprise IT Architect for IBM Software Group, delivering customer engagements around projects related to SOA, BPM and Web 2.0. Raj is the co-author of the upcoming “SOA with Java” book for the Prentice Hall Service-Oriented Computing Series for which he contributed chapters relating to portal technology and REST service design and development.

Raj has further developed a series of REST-inspired design patterns that have been contributed as candidates for the SOA Design Pattern Catalog. In addition he speaks at various industry conferences on a regular basis on the topics of SOA, Semantic Web and Web 2.0.

He just completed his Masters in Software Engineering from University of Texas at Austin and is starting his PhD.

http://www.servicetechmag.com/contributors/rajbalasubramanian

About the Author: Benjamin Carlyle

Benjamin has been involved with the REST community through his blog and other forums since 2004. He is credited with inspiring the popular Restlet framework for Java, he coined the term “REST Triangle”, and has deep understanding of both the theory and practice of REST-style architecture.

As an architect working in the Rail industry he is experienced in bringing together REST architecture, systems architecture, systems integration, and a variety of other topics at an enterprise scale. Benjamin has recently been working with the likes of Thomas Erl on documenting the convergence between SOA and REST.

http://www.servicetechmag.com/contributors/benjamincarlyle

About the Author: Cesare Pautasso

Cesare Pautasso is assistant professor in the new Faculty of Informatics at the University of Lugano, Switzerland. Previously he was a researcher at the IBM Zurich Research Lab and a senior researcher at ETH Zurich. His research focuses on building experimental systems to explore the intersection of model-driven software composition techniques, business process modeling languages, and autonomic/Grid computing.

Recently he has developed an interest in Web 2.0 Mashups and Architectural Decision Modeling. He is the lead architect of JOpera, a powerful rapid service composition tool for Eclipse. His teaching and training activities both in academia and in industry cover advanced topics related to Web Development, Middleware, Service Oriented Architectures and emerging Web services technologies.

For more information, visit www.pautasso.info

http://www.servicetechmag.com/contributors/cesarepautasso