Requirements-driven Approach to Service-oriented Architecture Implementation

ZELJKO PANIAN
The Graduate School for Economics and Business
University of Zagreb
J.F. Kennedy Sq. 6
CROATIA

Abstract: - This paper is an attempt to present an approach for transforming Service-Oriented Architecture (SOA) principles from concepts to design and then to code. We present a systematic, requirements-driven approach for designing and building a comprehensive framework for developing enterprise applications using the SOA principles. Some of the key design considerations for SOA are identified, as well as the logical design elements required to address design considerations.

Key-Words: - Service, Service-Oriented Architecture, Web Services, requirements-driven approach, orchestration, management

1 Introduction

There is lot of literature on what Service-Oriented Architecture (SOA) is, and so we will just cover this topic very briefly. SOA concepts are primarily designed to achieve the vision of an agile enterprise with a flexible Information Technology (IT) infrastructure that enables a business to respond to changes in the best possible way. As the business dynamics change and new opportunities emerge in the market, the IT infrastructure of an enterprise should be designed to be able to respond quickly and provide the applications needed to address the new business needs before the business opportunity disappears [1]. This is possible within reasonable costs only through reuse of existing investments. This is where SOA concepts come in; they are based on the principle of developing reusable business services and building applications by composing those services instead of building monolithic applications in silos [2].

One of the best ways of enabling application developers to understand concepts and put them to use is by providing a framework that provides the infrastructure needed while designing and developing applications based on those concepts. Unfortunately, there is not enough literature that can help application architects and developers in the design and implementation phases to build on the SOA concepts, apart from those from product vendors, which mostly explain in terms of their products/technologies.

So, naturally, there are fewer options for frameworks that provide all of the basic building blocks needed to build applications using SOA. In this paper, we attempt to fill this gap by providing a systematic requirements-driven approach to designing a framework for SOA.

2 SOA and Web Services

The advent of Web services has precipitated a fundamental change in how IT infrastructures can be developed, deployed and managed [3]. The success of many Web services projects has shown that technology does exist that can enable an enterprise to implement a true SOA. It allows the enterprise to take another step back and examine its application architecture—as well as the basic business problems it is trying to solve. From a business perspective, it is no longer just a technology problem, it is a matter of developing an application architecture and framework within which business problems and implement solutions can be defined in a coherent, repeatable way.

First, though, it's important to understand that Web services does not equal SOA. Web services is a collection of technologies, including XML, Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description, Discover and Integration (UDDI), which allow the enterprise to build programming solutions for specific messaging and application integration problems [4]. Over time, these technologies can be expected to mature, and eventually be replaced with better, more-efficient, more-robust technology. But for the moment, the existing technologies are sufficient, and have already proven that an SOA can be implemented today [5].

An SOA is exactly what its name implies—an architecture. It's more than any particular set of technologies, such as Web services. It transcends these technologies – and, in a perfect world, is totally independent of them [6]. Within a business environment, a pure architectural definition of an SOA might be an

application architecture within which all functions are defined as independent services with well-defined invokable interfaces, which can be called in defined sequences to form business processes [7].

Important components of this definition are:

- All functions are defined as services. This includes purely business functions (such as create a mortgage application or create an order), business transactions composed of lower-level functions (such as get credit report or verify employment) and system service functions (such as validate identification or obtain user profile).
- All services are independent. They operate as "black boxes;" external components neither know nor care how they perform their function, merely that they return the expected result.
- In the most general sense, the interfaces are invokable; that is, at an architectural level, it is irrelevant whether they are local (within the system) or remote (external to the immediate system). It doesn't matter what interconnect scheme or protocol is used to effect the invocation, or what infrastructure components are required to make the connection. The service may be within the same application, or in a different address space within an asymmetric multiprocessor, on a completely different system within the corporate intranet, or within an application in a partner's system used in a B2B configuration.

In an SOA, the interface is the key, and it is the focus of the calling application. It defines the required parameters and the nature of the result. This means that it defines the nature of the service, not the technology used to implement it. The system must effect and manage the invocation of the service, not the calling application.

This function allows two critical characteristics to be realized: first, that the services are truly independent, and second, that they can be managed. Management includes many functions:

- Security, to authorize requests, encrypt and decrypt data as required, and validate information.
- Deployment, to allow the service to be moved around the network to maximize performance or eliminate redundancy to provide optimum availability.
- Logging, to provide auditing, metering and evaluating capabilities.
- Dynamic rerouting, to provide fail-over or load-balancing capabilities.
- Maintenance, to manage new versions of the service.

3 SOA Implementation Framework Design

One of the first activities in designing a SOA implementation framework is to have an approach that helps in arriving at the desired objective systematically [8]. There should be a clear vision and goal, a set of core guiding principles, and a systematic process.

The goal is to provide a framework with the infrastructural components needed to develop enterprise applications based on SOA concepts. Some of the core guiding principles that will be used include [9]:

- Should be driven by requirements.
- Should be simple to use.
- Should be standards-based and pattern-driven.
- Should be practical.
- Should not become outdated quickly.
- Should buy/reuse anything existing instead of building it again.

Our suggestion is to first identify the significant requirements for developing services, and then to identify the key design elements needed to address those requirements, based on applicable design patterns. Then, define a framework that provides the basic design elements identified.

From a technical perspective, the core principle of SOA is that, to use some functionality, a service consumer should be able to look up a service that provides that functionality and use it [10].

The design implications are:

- Service design should be interface-driven. The focus of such an interface should be the requirements of the functionality to be provided exposed as a reusable service.
- There should be a well-defined service lookup mechanism that the service consumers can use to get a handle to the implementation of the service interface.
- The user code should not be tied to the implementation specifics of the service. Ideally, user code should not change if the technology used for the service implementation changes or if the underlying implementation logic is subject to change.
- The user code should not have to deal with the life cycle aspects (ideally all aspects) of a service like creating, initializing, configuring, deploying, locating, and managing a service. There should be well-defined mechanisms that take care of creating, initializing, configuring, deploying, and managing a service that finally provides a mechanism for the end user to look up the service and use it [11]. There should also be mechanisms that will allow for defining other service aspects, like access control to the services or audit of

service access where the user can plug in their logic [12].

The framework should standardize the definition, initialization, deployment and configuration, and management of services to address these requirements.

One of the key aspects that the framework should be addressing is providing a unified interface for all of the multiple technology options available for implementation services such as Enterprise Java Beans (EJB), .NET, mainframe-technology-based, etc.

A framework that enables an enterprise to implement its applications using SOA should therefore enable the service providers to define the service and the users to look up and use the service in a standard and consistent way and then take care of all of the "aspects" of the services.

4 The Framework Components Required to Implement SOA

The architecturally significant features of the framework are:

- A clearly defined mechanism to define a service interface with the available operations and input and output parameters.
- A registry of services that the service providers can use to register their service implementations and that the service consumers can use to look up a service implementation.
- An enterprise service bus (ESB) into which the service implementations can plug in and out, and which supports multiple calling semantics (such as synchronous, asynchronous etc.), and features like transformation, routing, etc.
- A well-defined service orchestration mechanism to take care of flow-based and long running interactions.
- A well-defined mechanism that takes care of service aspects such as configuration, management, access control, audit, etc.
- Well-defined service invocation mechanisms with adaptors that will allow the service to be invoked and implemented through multiple different technologies.

4.1 Service Definition

One of the first things that the framework should provide is a standard mechanism for defining the service interface with the various requests supported and the request and response parameter data formats.

The enterprise architecture teams usually establish the mechanism to use for defining the service interface in an enterprise. Since most enterprises use applications and systems that are implemented using multiple technologies and platforms, most enterprises select an XML-based mechanism to define the service interface. Web Services Definition Language (WSDL) is an industry standard for defining the service interface, but it is not necessarily the only way; there are several enterprises that already have existing XML formats for the defining service interfaces [13].

From the service provider's perspective, the framework should provide support for designing a service implementation. The framework should define a mechanism and, if possible, tools that help keep the service interface defined in the Enterprise Message Format (EMF) synchronized with the service implementation. Similarly, from the service consumer's perspective, the framework should provide the components to define a service stub to represent the Service Interface that is specific to the service consumer implementation.

So, for a Java-based implementation, there should be a mechanism to create a Java interface, and a stub and proxy implementing the interface and encapsulating the service invocation specifics. The stub and the proxy help insulate the service usage and service provider code from the service definition formats and the invocation specifics [14]. The framework should provide a tool that can generate the implementation code from the service definition and vice versa.

It's possible to design a generic stub that can represent any service, but the first choice should be a strongly typed interface for each service, to help in compile-time checks and in better object-oriented design. If, say, an attribute in the message changes, then the re-generated service interface can help identify problems at compile time instead of causing a costly runtime debug exercise. The dynamic stubs should only be used for scenarios where a service is used as a generic service.

The framework components for service definition are shown in Figure 1.

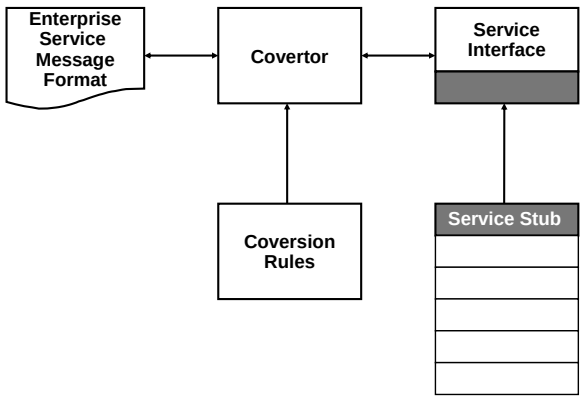

Fig. 1 – Service Definition

Having these mechanisms defined helps maintain consistency in service definition and usage across projects and also helps in service specification management later during the maintenance phase.

4.2 Service Registry

One of the important requirements to be addressed by the framework is to provide a Service Registry with details of the service interfaces and the service providers. It should also provide a standard mechanism for the service providers to register their implementation of a service interface and for the service consumers to look up the implementation of a service interface.

This mechanism is illustrated in Figure 2.

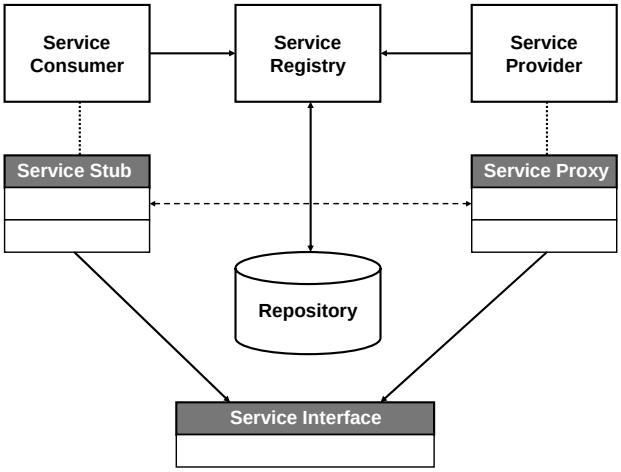

Fig. 2 – Service Registry Mechanism

The framework needs to provide a Service Registry design element that provides the Application Program Interface (API) to register and look up the Service Stubs that implement the service interface [15]. The Service Stub encapsulates the invocation details for the consumers and interacts with the Service Proxy that encapsulates the invocation details for the service providers. The service invocation details are explained in the next section.

4.3 Service Invocation

The next important requirement to be addressed by the framework is to standardize the service invocation mechanism and provide the infrastructural components with clearly defined interfaces that shield the service consumers and the service providers from the underlying implementation details.

Usually, enterprise architecture teams define the communication policies for applications in an enterprise. The communication policies define the strategies on when to use native protocols, when to use point-to-point communication, and when to use message-oriented communication. A common debate while determining communication policies is whether to use message-oriented communication to invoke a service or to directly invoke the service using its implementation-specific synchronous protocol such as RMI. One of the fundamental business requirements is to differentiate the service levels offered to the end customers based on their business value to enterprise, so that it helps achieve the desired client experience business objectives.

This is technically possible only if the communication mechanism used in invoking services is controlled and the service invocations can be prioritized. Using message-oriented communication instead of directly invoking the service using its implementation-specific synchronous protocols thus provides the mechanism needed to address this requirement. Strategically, the preferred communication mechanism should be one based on a messaging infrastructure instead of point-to-point invocations.

The infrastructural logical components that are needed for service invocation include:

- Service Stub
- Service Proxy
- Adaptors
- Message Broker
- Message Bus
- Gateways

The Service Stub implements the delegate pattern and provides the service interface to the service consumers, hiding the invocation details.

The Service Proxy implements the proxy pattern and provides the abstraction of the invocation details for the service providers.

The Adaptors provide the technology-specific integration mechanisms for the service stubs and proxies. The adaptor can provide the listener mechanisms that the stubs and proxies can use to receive the messages and the API to send a message.

The Message Broker and the Message Bus provide the transformations, routing, and other such services. The broker and the bus take care of transforming the message representations from the service consumer and service provider internal formats to the Enterprise Message Format and vice versa. They also provide the routing of the messages, store and forward, message retries, prioritizing of messages, etc.

The Gateways provide the mechanisms for external integration. The gateways provide the single points of contact for the external partners and transform the invocation protocols and message formats from the external partners to the internal enterprise message formats using a message broker and an adaptor. They also enforce the security checks, audit requirements, etc.

Having clearly defined interface driven components for each of these helps replace implementations with minimal impacts.

4.4 Service Orchestration

Further important requirement to be addressed is the orchestration of services for the implementation of a business process.

The framework should provide a mechanism tools to define, execute, and manage the service orchestration. The framework should define an Orchestration Adaptor that helps abstract the interactions with the orchestration implementations (Business Process Management tools) through an adaptor interface with an API to initiate processes, get the list of process instances, get the list of activities and their states, and to manipulate the state of activities, list of exceptions and unusual conditions, etc. that provide an abstraction over the implementation specifics [16].

The adaptors can then be implemented for the selected orchestration implementation. Since they all provide the same API, they can be replaced easily as needed without greatly impacting the services interacting with the orchestration component.

4.5 Service Management

The next important requirements to be addressed include providing a standard mechanisms for management of the services, for configuration of services, for taking care of the cross-cutting concerns like the access control, audit, etc. that apply to all or most service requests driven by centralized policies.

The diagram in Figure 3 shows some of the components that the framework needs to define/provide to address these all of these requirements.

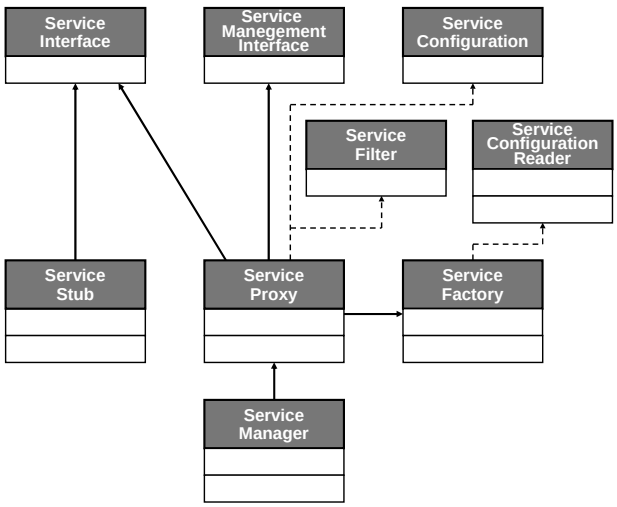

Figure 3 – Service Management

To address manageability requirements, the framework should define a standardized mechanism to design the Service Management Interface for the services and to make the service proxy provide an implementation of the management interface of the service and then provide a Service Manager component with the mechanism to make the service management interfaces accessible through standard management tools.

One of the common requirements in service design is to ensure that the service is configurable so that a service instance can be localized to a particular context and deployed [17]. The framework should therefore provide a standard mechanism for service configuration. The framework should define a standard Service Configuration Format, a Service Configuration Reader component, a Service Configuration component to represent and hold the service configuration information, and a Service Factory component that takes care of the creation of the service, loading the service configuration, and initializing the service with the desired configuration.

The framework should provide a mechanism to allow the separation of the service core functional logic from the logic for enforcing the cross-cutting concerns like access controls, audits, etc. The framework should define a Service Filter component that can be plugged into the service invocation mechanism at service proxy to intercept the service requests and apply the Quality of Service (QoS) aspect logic.

4.6 Implementation

The last step is to implement the framework with the various design elements identified earlier. Using the "buy instead of build" principle, it makes sense to first evaluate what can be leveraged off the shelf, and then build the missing components on top of the selected implementations.

5 Conclusion

The paper presented an approach to transform Service-Oriented Architecture (SOA) concepts to appropriate implementation through a comprehensive framework. Some of the key design considerations for SOA are identified, as well as the logical design elements required to address design considerations and the framework that provides the basic components needed.

The primary intention was to present a systematic, requirements-driven approach to developing an enterprise application framework for SOA and take it from concepts to the design-elements level. The next steps in research would require further investigation of the deployment opportunities, constraints and limitations

to the proposed approach. It would also be necessary to evaluate the real-world potentials, strengths, and weaknesses of the framework modelled.

References:

[1] Leffingwell, Dean; Muirhead, Dave, Tactical Management of Agile Development: Achieving Competitive Advantage, http://www.rallydev.com, 2004

[2] Fiorano Staff, Business Component Architecture – Driving Business Agility, Fiorano (http://www.fiorano.com), 2005

[3] Clabby, Joe, *Web Services Explained*, Prentice Hall PTR, 2003

[4] Newcomer, Eric, *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, Addison-Wesley, 2002

[5] Panian, Željko, Creating Agile Business through Service-Oriented Architecture, *Proceedings of the IPSI 2006 Conference*, Marbella (Spain), February 2006, pp 140-146

[6] Finkelstein, Clive, The Enterprise: Service-Oriented Architecture (SOA), *DM Review*, No. 1/2005

[7] Kaye, Doug, *Loosely Coupled: The Missing Pieces of Web Services*, RDS Press, 2003

[8] Bloomberg, Jason, The SOA Implementation Framework: The Future of Service-Oriented Architecture Software, ZapThink (http://www.zapthink.com), April 2004

[9] Morgenthal, J. P., Enterprise Architecture: The Holistic View: In a SOA Universe, How Important is Platform Selection?, *DM Review*, No. 7/2005

[10] Thomas Manes, Anne, *Web Services: A Manager's Guide*, Addison-Wesley, 2003

[11] Kelly, David A., What You Need to Know About Transitioning to SOA, http://www.ebizQ.net, 2005

[12] Roguewave Staff, Leveraging C++ Business Logic In a Service-Oriented Application, Roguewave (http://www.roguewave.com), 2004

[13] Marks, Eric. A., Werrell, Mark J., *Executive's Guide to Web Services*, John Wiley & Sons, Inc., 2003

[14] Kelly, David; Ashton, Hather, J2EE Service Management: Are You Ready?, http://www.upsideresearch.com, 2005.

[15] Infavrio Staff, The Web Service Registry, key to a successful SOA, Infavrio (http://www.infavrio.com), September 2004

[16] CapeClear Staff, Principles of BPEL, Orchestration, and the ESB, CapeClear (http://www.capeclear.com), 2004

[17] Kay, Russell, QuickStudy: SOA, *Computer World*, No. 03/2004, pp 12-16