Service Design Essentials

Srikanth Inaganti and Srini Chintala

Enterprise level SOA transformation involves collaboration and integration across many projects that are critical to a business, with the iterative development of services and composite applications, some of which have to be developed afresh and some that will have to be re-architected or re-designed to use new platforms. Many of the enterprise-level services are typically identified during the portfolio assessment and realized in parallel with developing new applications or re-architecting existing applications. The challenge for architects and designers working on service design and development has been to think about how well design can absorb the provisioning of future requirements. This is not possible without crossing the boundaries of the projects and looking at the whole, related IT environment. Perhaps a thorough involvement of business owners in the service identification [4] and definition phase would clarify some of it. But the rest of the solutions come from the completeness, extensibility, and configurability that are to be built into the service design itself. SOA-based service design needs to be executed with long term thinking and hence requires experience. In this context, the common questions that some of the project managers and quality managers ask is this: How do we determine that the design is futuristic, compliant with SOA principles, and what are the additional things that SOA must bring into the design of applications and services?

During SOA transformation, some of the implementations at enterprise level are found to be non-compliant with the definition of the service itself. As per our observation, the sources of such anomalies have been a high technology focus and the misuse of the utilities being offered by most of the products in the SOA market when converting functions/components and deploying them as web services! These result from a lack of far sightedness, delivery pressures/commitments on the service provider side, and a lack of SOA evangelization, appropriate governance processes, and system integrator skills on the client side that make the SOA project implementations too tactical, leaving strategy in its silo!! Hence, this paper is aimed at highlighting the important technology-agnostic design aspects of the service along with their importance, in order to help with a quick review of SOA project implementation. If the following proposed design aspects are not considered during the services design phase, one can probably say there is scope for further improvement in the way SOA is implemented.

A simplified definition of service is that it is “a function or piece of code available over the network that consists of interface, implementation, and associated data.” If the service is not used in more than one scenario, what is the point of making it available or discoverable over the network? It might as well stay within the application itself. The missing element in most service designs has been reuse analysis – especially in the case of project-specific SOA implementations. Architects and designers should always keep questioning themselves: Should a function or piece of code that cannot be used in more than one application or scenario be treated as a service or component? Along similar lines, the following sections depict a few important guidelines to be considered during service design to make it extensible so that it can accommodate changes with minimal impact.

Interface Definition Considerations

Architects and designers often debate for long hours about sizing, granularity, or boundaries of service because of the subjectivity involved. A simple way to diffuse the situation is to look for completeness of functionality or operations around the business process or data entities in question. A well-designed, high-level component (façade) interface mostly addresses the appropriate service interface requirements. Most of the business services are usually composite services that are formed making use of existing low-level services. For business services, 100% functionality coverage would make it a right boundary for a service – for example, a payment gateway service that takes all potential forms of handling payment transactions like credit cards, debit cards, travelers’ checks, promotional vouchers, gift cards, etc. In the case of data services, such as a customer management system being designed for a bank, perhaps CRUD operations on CUSTOMER data entity can provide a potential list of methods in a service.

Note that a service should be right-sized so that network bandwidth is conserved.

Best Practice: *Avoid too finely grained methods in the service interface definition – an essential matter when that service has high reuse potential and usage becomes very high over a period of time.*

Another important aspect is encapsulation. The service's interface (contract) should be designed to enable evolving the service without breaking contracts with existing consumers. In an overall service lifecycle, it is common to go back and revise the service method signatures, as new consumers get added to portfolio, by introducing new attributes that bring in additional business logic.

Best Practice: *In these cases, encapsulate the business and operational data required for service invocation into logical units to promote extensibility.*

For example, always use USER object as one of the parameters rather than the individual attributes of USER as parameters. If the addition of one input parameter in the method signature would disturb the existing service consumers, this becomes a non-acceptable change, with respect to serialization, for the rest of the service consumers using the old method signature. If USER object is used, adding one attribute to it will become an acceptable change.

Message-centric design is preferred over method-centric design when defining the interfaces for a service since it creates the necessary loose coupling between service consumers and providers [2]. Message-centric design promotes the use of message structures – in other words, SOAP and XML schemas for message definition – while taking advantage of the extensible nature of XML schema (xsd:any) and the SOAP processing model. Many vendors are offering tools to generate (i) stubs from WSDL (ii) WSDL from stubs.

Best Practice: *If we are building new services, start writing WSDL and then generate native program model stubs. If an existing code is being modeled as a service, then generate WSDL from the native program code and modify as appropriate to make the service interoperable.*

Based on the notion of standards-based integration, most of the SOA product vendors are generating a lot of hype about ESB and web services implementation as the way to good SOA design. The truth is that good SOA designs can be achieved even with traditional messaging infrastructures and without web services [3]!

Best Practice: *Limit the usage of web services unless the IT landscape is diverse and the impact of flux in web services standards is evaluated.*

Implementation Considerations

Every service request should recognize the service consumer and have locale as one of its input parameters.

Best Practice: *Every service consumer should send CONSUMER_ID as specified by service provider.*

This is essential in tracking the service usage per consumer for reuse ROI purposes, and it helps determine the charge-back model for that consumer. It also helps in identifying/classifying the consumer for applying the various custom business rules required to perform the service request [1]. Passing the locale and CONSUMER_ID information to the service would help pick up the right business rules that are applicable to different regions or perhaps would lead to executing an altogether different set of back-end components. This can be implemented either through a business rules engine or properties or XML files or by using a template method pattern to externalize the business logic into an additional piece of code, to use it during runtime. For example, in the banking and credit card business, the locale would determine the right set of credit limits modeled as rules to be applied in different countries or regions.

Another aspect to be considered is the provisioning of consumer specific requirements within a service request. It is a common phenomenon in the SOA world to readjust the deployed services to meet new or changing consumer requirements. A traditional design approach has been to differentiate the flow of execution based on some key parameters like CONSUMER_ID, locale, etc. But this approach requires a service to be redeveloped and deployed, affecting other consumers during the time of redeployment.

Best Practice: *Separate the entire service design or implementation in to two parts such as the service core and the application of specific pieces on the service side, and invoke the application specific service logic using a callback object.*

This facilitates adding extra packages to the service library without bringing service down. Applications or service consumers would get this callback object as the return parameter of a service request which would shield the consumer/application from nuances of native technologies with protocols for invoking additional business logic on service side. This technique would improve the extensibility of services components and would also fuel further reuse opportunities for similar consumers.

Although designed services have interfaces implementation with a particular programming model most of the time, there are missing pieces such as (i) separate data schema (ii) availability over the network as a separate entity.

Data Schema Considerations

Many implementations have the service data mixed with application data. This is partly due to lack of SOA experience, but also far-sightedness with respect to reuse. The implication is that there are strong referential dependencies between service and applications data, and impediment to reuse leading to a failure to accommodate a new service consumer's data cleanly.

Best Practice: *Make sure service has its own data schema separate from that of the application and ensure that service is responsible for the data integrity.*

And also within the service data, sometimes, having a separate schema for every consumer would make the component or service easily saleable for reuse. The decision to have separate schemas for different consumers depends on the volume of the data being created and handled per consumer, its impact on data retrieval for reporting and the business sensitivity involved in keeping the data mixed up with other consumers. It is common for application and business owners to express concern that their data is mixed up with others – a strong impediment for reuse promotion! Having CONSUMER_ID in a service request, as specified above, would enable the service to target the appropriate data schema. Please note that offering data load services in to the service (target) schema would help a lot to promote the service reuse by making consumers shift from existing implementations in their applications.

The service data schema should also have a provision to record the service invocations and activities for various reporting requirements. Though most of the service management tool kits available in the market are providing this facility with pre-defined schemas, it is preferable to design the service schema with inherent configurable tracking capabilities for critical business transactions.

Packaging Considerations

Services should be deployed with versions independent of the system in which they are deployed and consumed. Often, services and the applications from which they evolved are being packaged into one runtime bundle. In the future, if that component/service is reused in different scenarios, separating it may not be so simple when native protocols are used within the application. In addition, once a service or component is separated from the application, usage of invocation frameworks like WSIF would shield the service from changes in technologies and in binding protocols.

Best Practice: *Make sure service and application runtime assets are packaged into separate runtime packages. Initially, services might run on top of application infrastructure but gradually as the reuse increases, they can be deployed onto dedicated, shared services hosting infrastructure.*

Miscellaneous

- *A service should not have any compile- time dependencies. A service should be able to build and run on its own.*
- *Mark the volatile parts of the applications. If those parts are falling under services – then try to model them as business rules or parameters or metadata that a service should read as appropriate. The tentative variables should be externalized either as parameters or as part of metadata. These parameters or metadata need to be separated for each service instead of there being one repository for many services. However, keep in mind that too much parameterization would make the users dependent on user guides, and b e more prone to errors in documentation with the complexity of correlating the impact of each parameter on the system. Hence, balance between code and metadata size is very important.*
- *Do not use web service interface unless (i) the service needs to be exposed to the outside world, (ii) consumer technology characteristics are unknown, and/or (iii) the IT landscape within the enterprise is heterogeneous, but it is certain to be reused. If the enterprise IT landscape is homogeneous, perhaps web service interfaces are not required except for exposing it to the outside world/partners.*
- *Check whether the design has a pre-processing layer (a.k.a service interaction layer) containing routing, validation, transformation, and enrichment as parts of a set of centralized services. Based on service request, CONSUMER_ID etc, this central service should be able to transform both input and output parameters as appropriate configurationally. Do not build the validation and transformation logic into the service end-points (a.k.a service processing layer).*
- *Make sure that the service interaction layer is capable of dealing with service outages. This means two things that depend on the criticality of the service to business – (i) how the service interaction layer really queues up the messages or service invocation requests, and (ii) how service unavailability is communicated to the consumers.*
- *In case of business logic failures or error conditions, decisions like whether to throw service-specific exceptions or to let the underlying system throw system-specific exceptions need to be addressed during the service design. There must be a plan for recovering from exceptions in situations that require recovery.*
- *Services should not lock the resources for too long. Any long-running business transactions modeled with services should be using correlation capabilities to associate to the right service in a right sequence.*
- *A service should be stateless and idempotent in nature. No information is allowed to be cached or remembered in session on server side between service invocations. Never leave the service or its associated resources in an inconsistent state.*

- *Always keep in mind that the portability and interoperability of the service is reduced with the usage of complex parameter types that are by default not supported. Hence it is suggested that the required client be provided stubs for various technologies within the enterprise to reduce high time integration and interoperability errors.*

- *In the case of realizing business services comprising of multiple services based on different technologies, ensure that orchestration logic contains a comprehensive set of compensating transactions. This is essential in the context of flux in web services standards and available product maturity.*

Summary

In a big program like enterprise SOA, not knowing all requirements or expectations from all parts of the enterprise for a particular service is quite common. Trying to gather a whole set of requirements would involve a big-bang approach which would stall or delay the entire transformation process. Hence, in order to support the service evolution in an iterative fashion, extensibility and flexibility should be built into the service design right from first iteration. Spending enough time on reuse with respect to interface definition and implementation, configurability in packaging aspects would bring necessary agility. Trying to expose any or all business functions as web services without reuse analysis and justification would lead to unnecessary performance issues. This article brought out some of the important technology agnostic design aspects, out of a host of service design principles, related to interface, implementation, and data schema that would make systems readjusted more easily in future.

References

1. Reuse Framework for SOA by Srikanth Inaganti, June 2007, BP Trends
2. Contract First Development by Zapthink LLC.
3. JMS based SOA monitors CERN particle accelerators
4. Service Identification: BPM and SOA Handshake by Srikanth Inaganti & Dr Gopalakrishna Behara, March 2007, BP Trends

Acknowledgements

We would like to thank Dr. Udaya Bhaskar Vemulapati, General Manager, Wipro Technologies, for providing the time and support necessary to make this article one part of our SOA Center of Excellence efforts.

———

Authors

Srikanth Inaganti is a Lead Consultant in the Enterprise Consulting and Architecture Practice division of Wipro Technologies. Srikanth Inaganti, PMP, E-Architect, ECAP Group, Wipro Technologies srikanth.inaganti@wipro.com, Office # 91 40 30796818; Mobile # 91 9849058064.

Srini Chintala is an Enterprise Architect in the Enterprise Consulting and Architecture Practice division of Wipro Technologies, India. He is a Sun Certified Enterprise Architect and can be contacted at srini.chintala@wipro.com, Office # 91 80 41363110; Mobile # 91 9980177889.