The Case for OKI in the Enterprise

Realizing a Service Oriented Architecture

S Thorne 5/8/06

Most Enterprises today are faced with an array of IT challenges. They need to maintain an existing technology base while absorbing a wide-array of new technologies. The costs of integration are high. The complexity and number of systems is growing. Technology continues to change at an increasing pace. Duplicate efforts result in pockets of overlapping functionality. Many of these issues are related to integration; making applications work with IT infrastructures and making system components work with each other. This is a hard problem that we've struggled with for years, and so we're hoping that Service Oriented Architecture (SOA) will solve it all.

This document explores the benefits of SOA and introduces the value of O.K.I. Open Service Interface Definitions (OSIDs), including;

- Service Oriented Architecture & Web Services
- O.K.I. OSIDs
- Web Services & OSIDs
- Benefits of O.K.I.
    - Common Factoring
    - Enabling Service Evolution
    - Integration & Choice
    - Enabling Service Patterns

Service Oriented Architectures & Web Services

Architecture is a framework that defines an organization of system components. SOA is an architecture organizing systems around services. Often SOA is equated with Web Services, but this is not accurate. Web Services is a set of technologies, while SOA is an approach. The largest obstacle to implementing SOA isn't the technology, but rather how we think about software in a new way.

As an example of thinking differently, let's use the example of integrating more than one Calendar system. Imagine you have two calendar systems in your enterprise that are used by different sets of people based on history. Everyone wants to continue using what they're using, but they want it "integrated." Integration could take a couple of basic forms: a data centric approach or a service centric approach.

The more common data centric approach would be to export all the scheduled items from system A and import them into system B, and visa versa. Whereas, in the service centric approach, the front-end applications would be separated form the back-end calendar stores. Then the user interface of system A could also display items managed in B, and visa versa.

These are radically different approaches to the integration problem. In the data centric approach, data is actually copied. This raises questions of how to manage and synchronize it in two places. How often is the data exchanged and resynchronized? How are conflicts between A and B handled? Are you allowed to change items in System A that originally were created in B? If more than two systems exist, this approach soon becomes unwieldy.

While the service approach requires having to redesign software, this model enables each system to freely manage the integrity of their information. Information doesn't have to be kept and managed in multiple places. Ultimately this should be the simpler approach. But, what are the barriers to doing it this way?

Web Services and more specifically SOAP and WSDL provide part of the solution, a mechanism to provide and consume remote services. But just having the ability to make service calls doesn't provide the benefit promised by SOA. You could implement the data centric approach of synching using SOAP and WSDL, but you really wouldn't have implemented a service-based approach.

Web Services are often used today to wire systems together. In practice though, specific services remain hard to use in another context, and applications can't easily switch service instances. Eventually we would get to a state of truly reusable and evolving services that can be swapped around as needed. But in order to fully achieve this, we need to have common semantic definitions of particular services. For example, without a common understanding of an authorization service, we'll never be able to choose from several competing ones.

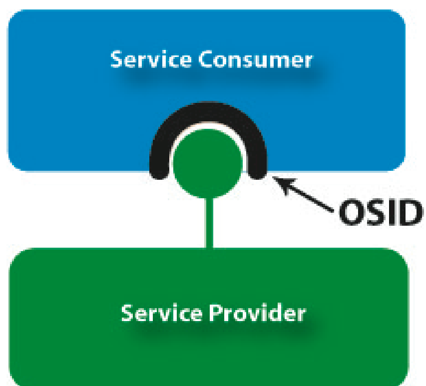
O.K.I. Open Service Interface Definitions (OSIDs)

There is an old saying that most computer problems can be solved by adding another layer of abstraction. OSIDs provide an additional abstraction point in an SOA. Adding Service Interfaces to other Web Service technologies allows for a more complete integration solution. The extra abstraction point, provided by OSIDs, is the independent contract that separates service consumer and service provider. This enables many integration related benefits.

OSIDs provide the semantic service descriptions that define an enterprise SOA infrastructure, but don't say anything about how the service is implemented. This provides a common way of factoring services, so that they can be easily switched as needed. Without the ability to switch services easily, the real benefits of SOA won't be achieved. OSIDs are software interfaces that define a contract between a service consumer and a service provider. Looking at services from these distinct perspectives provides flexibility for both Service Consumers and Service Providers.

From the Service Consumers' point of view, the service implementation details are hidden. Service Consumer don't build a dependency on a particular service instance, so they're locked into one service provider. The OSIDs provide common service definitions that insulate the Service Consumer, so they can easily switch Service Providers as needed.

The Service Providers want their services to be used by anyone, but also want to be free to evolve and enhance it. We can expect that services will continually evolve, so these services will need to change while minimizing the impact to their consumers. Service Providers want to be free to use whatever technologies they choose, so that they can compete in areas such as performance and reliability.



OSIDs and Web Services

Web services and specifically WSDL and SOAP represent useful technologies for providing distributed services across an enterprise and beyond. Web services can provide any kind of business function. The key characteristic that all web services share is that they can be accessed by messages sent over the Internet using standard web protocols. This makes them language and platform neutral. For a service provider this makes their service accessible to anyone.

However, from the Service Consumer perspective, even if it's using this technology, each instance of a particular service will have differences. For example, variations in the WSDL will exist. For example, the method calls will likely have differences. This means that an application will require specific code for that service instance, which will likely need to change when a new version of a service is adopted. OSIDs provide the generic calling interface for the application that shouldn't change when a new version of a service is swapped in.

It should be noted that OSIDs are programming language specific and this is often sited as a drawback. However, a service consumer is usually an application written in a particular programming language. A stable service interface in the local programming language environment is exactly what is needed by the Service Consumer in order to remain independent of particular service implementation.

The combination of Web Services and OSIDs address the needs of both Service Providers and Consumers. Web Services provide for a service implementation, while the OSIDs shield the application from variations in particular service implementations. Web Services and OSIDs are complimentary since they are each designed to solve a piece of the SOA problem. Web Services define how the information moves across the wire, and OSIDs define how applications use a service. Using OSIDs with Web Services enables you to evolve the service without always affecting the Service Consumer, and therefore gives maximum flexibility in an SOA.

An SOA is a complex system and will require many different pieces to make it all work. Just as pipes are an important component of a water system, they are not the only thing. You still need an array of fittings to make the pipes useful. A basic fitting just connects two pipes, others allow different size pipes to be joined, pipes to be split or combined. How all of these are combined makes for an entire water system. Similarly, OSIDs and web services are examples of pieces needed to be used in combination to implement a Service Oriented Architecture.

Benefits of OKI

O.K.I. OSIDs provide many benefits of particular importance to the Enterprise. Among them are:

- Common Service Factoring and Reduced Integration Cost
- Enabling Service Evolution
- Integration and Choice
- Enabling Service Patterns

Each of these areas will be covered in more detail.

Common Factoring and Reduced Integration Costs

In an enterprise you have unique business services that you'd like to define and build. Underlying these services or applications are more fundamental services such as authorization and logging. Usually you don't need anything special for these and it would be better to use a generic service. This allows you to concentrate on the part of the problem where you can add value. This common definition is what the OSIDs provide. Without common service definitions your enterprise will

likely create many separate authorization services. Wouldn't it be better to have one well-managed authorization service? Starting with a predefined service interface saves time.

Centralizing a service offers several benefits: economies of scale, improved synchronization of data, and more consistent application of business rules. All the business logic associated with access to and control over the function can be managed in one place. Delivering an enterprise service increases control, ensures consistency and uniformity, and allows all applications to leverage implementation improvements. OSIDs serve as a way to centralize a function or service for use throughout the enterprise.

Agreeing up front on the general service factoring supports system integration and organizational effectiveness. By defining the integration points early, each system is free to implement their piece, while knowing exactly what the ultimate integration will look like. The OSIDs act as generic templates or placeholders for services in this model. A basic test version of the service can be provided while the application is in development and then swapped for a production service later.

Common or similar factoring makes it easier for programmers moving between systems. OSIDs benefit enterprise IT by defining responsibilities among programming groups. Groups' responsibilities can be bounded around specific services. Programmers can be divided into those implementing an OSID and those writing applications that consume specific OSIDs. Clear boundaries improve management's ability to measure progress and apportion resources. For example, one can ask, "Has the Group Service been implemented and tested?" A precise division also simplifies testing and focuses problem determination. This idea of defining responsibilities extends to software suppliers. For example, one can ask, "Does your application work with this Group Service?" Eventually, language can be included in RFPs that dictates precisely integration and interoperability characteristics.

Enabling Service Evolution

Services can be thought of as the common functions that are required in many systems and applications. Usually, each application has built in this functionality. One of the problems getting started with Service Oriented Architecture is that these functions are already being performed in a particular way. Changing all the underlying implementations would be too disruptive, so a strategy to evolve into an SOA is needed.

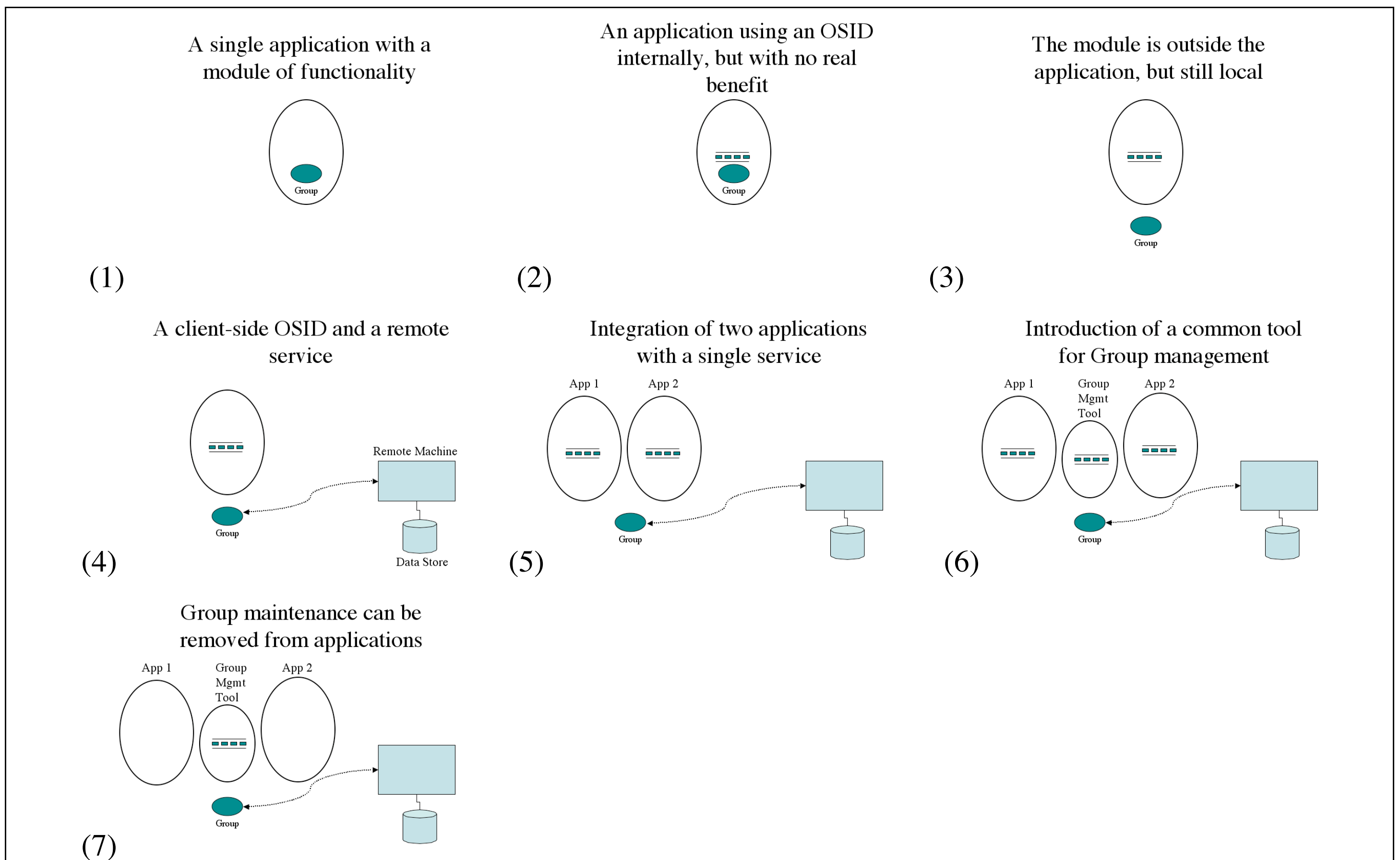
As an example, the concept of managing groups of people for mailing lists and access is often part of a system's functionality. This common functionality is often implemented in different ways, which results in information being maintained in different places and unable to be easily reused. In a SOA approach, a group service would be responsible for the integrity and management of group information, usable by all applications. .

If I'm responsible for an application that already has this group functionality, what is the benefit for moving to a new service? I don't save development time, since I've already done it and changing it would take more effort. I'd also be dependant on someone else, and would be unsure of the performance and reliability.

One small step towards adopting an SOA approach for this application would be to cleanly separate the group functionality from the rest of the application. This has the benefit of embracing a common approach, while still using the same basic implementation, presenting little risk to the application.

Since the Interface used is standard, this application would then be ready to swap in an enterprise service for their local one, when the conditions were right. It doesn't force a change in the underlying implementation.

Once the system was using the new Enterprise Group Service, it could still provide an interface for group maintenance, but the benefit would be that this was now integrated. If someone made a change to the “blue” group in any system it would affect them all. Eventually a separate user interface for group management could be constructed and deployed. This would relieve each individual application from having to build this functionality.



These diagrams show a single application that evolves to include a module of functionality for managing groups. In step (1) there is only recognition of the functionality. Interaction with this module becomes structured with the introduction of a service interface in step (2). The service implementation moves outside the boundaries of the application, but remains local in step (3). The benefit for this step is that now a different implementation of the group service can be substituted without affecting the rest of the application. The original service implementation is changing, while the service interface does not. In step (4), the implementation evolves to use a remote machine and data store. In step (5), two applications now use the same group service. The significance of this step is that now the two applications can use a single system that enforces uniform business rules such as authorization. A group management tool emerges in step (6) dedicated to the task. In step (7), applications no longer need to build in Group functionality.

Saving code and development time is often stated as a goal of an SOA approach. While it might initially require a bit more work than traditional methods, eventually if the services are designed correctly, their use in more than one application will result in savings in development time

Integration & Choice

Interfaces are a well-known concept used in integration. If a programmer wants to integrate with another product, they code to that system's Application Programming Interface (API). You could do the same thing in a world of services. Just pick the services offered by a vendor and program directly to their definition of the service. However, doing this restricts your future choice. Once you've made the investment in a product and integrated it into your environment, switching to a better solution when it's available isn't a practical alternative.

The current cost of integration is so high, that it prevents new solutions from being easily adopted. That means as an Enterprise, you are not always using the preferred solution, and may be paying extra as well because the cost to switch is so high. Lowering the integration costs reduces this barrier and makes it more likely that you can take advantage of alternatives. OSIDs are a neutral open interface that provides well-understood integration points. Having this open integration point between products means you don't build in a dependency to a particular vendor and you have more leverage in future negotiations.

Enabling choice may be the most important benefit of a Service Oriented Architecture, because it is the aspect that will be most noticeable for the average user. No user likes to be restricted in which tools they use. Not only do different users like different sorts of User Interfaces, they also want choice of the device they use. One user may prefer a web interface, another desktop application. Others want to use a phone, PDA or iPod. Enabling this wide variety of choice without prohibitive cost is what an SOA promises.

As an example look at email; people read email in a variety of ways. Depending on circumstances, a user may access their mail with any number of techniques.. The reason they have this choice is that the back-end mail system is exposed to the mail "clients" in an open way. This makes it feasible to have many front-ends and provides choice to the user. We'd like to be able to change back-end systems in the future without impacting the application a person is using. Forcing users to switch applications has a cost that an Enterprise would like to avoid. Often people only think an SOA will allow them to use new services, but what most users will see is the ability to use more applications. O.K.I. directly helps this by allowing applications to build to one service definition and integrate with multiple services.

Having open specifications creates the potential for a market, since many can compete in providing products that meet a common definition. Prior to having open specifications in an area, customers have to pay a premium for software and its integration. Once a practical open specification is defined, the cost of both the software and the integration is dramatically reduced. Accelerating the time it takes to establish these open specifications in services should have profound effects on the software market. O.K.I. is one attempt at such a set of service definitions.

One of the major concerns of an Enterprise is maintaining their ability to choose appropriate technologies as needed. Often though, existing technologies limit the choices that are applicable. Wouldn't it be a major advantage if you could pick from any vended or open source solution? Open source products are often thought of as offering and providing an advantage in integrating. Having the ability to see and alter the source code means they can be integrated with, but it doesn't mean that it is any easier. Products that integrate using open specifications are better for the Enterprise regardless of whether they're open or vended. OSIDs are a set of open service definitions and therefore help by reducing Enterprise IT costs in two ways; by directly reducing integration cost and enabling a more effective market place for software.

Enabling Service Patterns

Just as in current software engineering, the idea of design patterns is useful when thinking of services. Services are just emerging, but several patterns can be anticipated. OSIDs make it easier to think about service patterns because each of the services definitions is only a contract. This makes it possible to wrap one contract over another without a performance impact. It is also easy for an implementation to adhere to more than one contract.

The most basic pattern is that high-level business services will be built upon more basic services. For example, an Authorization service might underlie a business transaction. It is important that this business service remain unchanged, even if it internally changes to a different authorization service or mechanism. As services evolve, changes to underlying services should be expected. Having an independent service definition will buffer systems from these changes.

Another basic service pattern is multiplexing. Here the use of OSIDs is to hide the complexity of more than one system. For example, although records are housed on three different student systems, an educational application can use them through a single enrollment service. This integration effectively hides the detail of multiple systems underneath. This enables the Enterprise to consolidate these systems without impacting the application.

Conclusion

Implementing an SOA isn't a matter of choosing a technology, training and implementation, but requires significant for-thought and top-down direction to achieve results. Over the past decades many IT benefits could be achieved through grassroots application of new technology, but to get to the efficiencies promised by SOA, a different approach needs to be taken. O.K.I. provides a new way of thinking about the problem and has the advantage that it can be done incrementally. You only need to implement as a particular integration benefit is recognized. Implementing only when the benefits are well understood guarantees the benefits are related to investment and thereby mitigates the risks of moving to an SOA. To get started down the path to an SOA, every time someone wants something integrated, ask, "How would we do this in an open service model?"

O.K.I. is more of an approach to enabling a Service Oriented Architecture than it is a technology. O.K.I. offers many benefits for the enterprise as it looks to build a Service Oriented Architecture. It provides a set of common service definitions that helps while laying out an Enterprise Service Strategy. It provides a migration path from existing technology to future service based technologies. It reduces long-term integration costs by providing a neutral integration point. It enables choice, both in underlying services and in user applications. The OSIDs define the integration points among for services, not the technologies or other service implementation details. This is important because it means existing technology can be used. Starting to think of services and how they could benefit your enterprise should be the first step down the path to a Service Oriented Architecture.