RC22494 (W0206-107) June 20, 2002
Computer Science

IBM Research Report

Web Services: Promises and Compromises

Ali Arsanjani*, Brent Hailpern, Joanne Martin, Peri L. Tarr**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

**IBM
Rt 100/MD 2302
Somers, NY 10589

*IBM Global Services
105 North D St.
Fairfield, IA 52556

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

Web Services: Promises and Compromises

ALI ARSANJANI **BRENT HAILPERN** **JOANNE MARTIN** **PERI TARR**

Abstract

This article will introduce the foundational concepts of Web services, levels of service integration, discuss the real promise of intra-enterprise integration through an underlying component-based development and integration approach on which Web Services can be built, and then review the technical problems that any implementer or development manager should realize will be part of Web services for the foreseeable future. Although much of the craze surrounding Web services revolves around inter-enterprise distributed computing, much of the initial power will come from integration within a single enterprise, either with existing legacy applications or with new business processes that span organizational silos. Enterprises will need organizational structures that support this new paradigm and its distribution of control and ownership of information. Finally, technical challenges will be paramount, as Web services are required to satisfy serious, demanding SLAs. When that is the environment, developers will need to confront the task of designing, building, configuring, deploying, and managing software that must meet, and continue to meet, constraints beyond functional correctness.

Keywords: Web Services, Component-based Architecture, UDDI, SOAP, WSDL, Service-oriented Architecture.

Introduction

Web services are the latest software craze: the promise of full-fledged application software that can deliver its function across the World Wide Web without being installed on your local computer. Their magic comes from using XML and a distributed systems model based on Web standards (e.g., SOAP, WSDL, UDDI) to provide an infrastructure for applications just as the Web provides the infrastructure for hyperlinked multimedia documents. The use of XML and other Web standards will enable systems running in different environments and at different locations to exchange information, interoperate, and be combined more readily than ever before.

The first generation of Web-service infrastructure and applications is already in development, allowing Web services to issue requests to others and to register/describe/find a service to use. Though much of the craze surrounding Web services revolves around this nirvana of inter-organizational distributed computing (supply chains being integrated across enterprises and across continents with applications built out of small parts that are supplied on demand by many distinct vendors), much of the initial power of Web services will come from integration of legacy applications within a single enterprise.

Web services infrastructure and the tools that will help build Web services "applications" hold out the promise that investments in large quantities of existing software can be leveraged into new applications both for the productivity of employees and for the benefit of customers. The promise is there, but there are numerous pitfalls along the way: security, performance, and software engineering issues of development, testing, deployment and maintenance.

Web Services involves a lot of new technology and as such business and technological teams are rapidly engaged in surmounting initial hurdles. But the time to start with these technologies that are gaining extremely rapid adoption is now and the place to start with them is within the enterprise. This will insulate the enterprise from initial technology hurdles that are imminently being resolved and as

well as the ones that remain to be resolved before this architectural paradigm and its associated technologies become ubiquitous.

Foundations of Web Services

Web services, and service-oriented architectures, have the potential to enable a new paradigm for enterprise application development and deployment. A web service is a self-contained and self-described modular application that can be published, located, and invoked across the Web. Based on standards and designed to exploit existing infrastructure, Web services provide significant opportunities for technical and business innovation. They encourage an approach to application development that is evolutionary, building on investments previously made within an IT organization, and developing new capabilities incrementally.

The service-oriented architecture is distributed, permitting elements of an application to be deployed on multiple systems and executed across connected networks. Because the transport mechanism is built on HTTP, which is ubiquitous in today's computing environments and supports compliance with basic firewall policies, it is possible for program elements to interact within and across enterprises. The elements of an application are designed to support specific tasks within a broader workflow or business process. Each of these service elements is responsible for defining its inputs and outputs using the standards for Web services, so that other application elements are able to determine how this element operates, how to make use of its functionality, and what result to expect from its execution.

The service-oriented architecture is built on a foundation of standards, which define the roles and activities of the architectural elements, and thus support the interoperability of incompatible systems across the Web.

- Simple Object Access Protocol (SOAP) [10] is the XML-based communication protocol used to access Web services
  - SOAP is a lightweight XML-based protocol for sending messages to and invoking methods on remote objects. Messages and method invocations are defined as XML documents and are sent over a transport protocol, typically HTTP. SOAP is language and platform independent.
- Web Services Description Language (WSDL) [11] is the Web services description API
  - The WSDL description of a service is an XML document with a specific format that provides the technical aspects of that service. The description includes the operations that can be invoked, their associated data types, the supported transport protocols, and the endpoint for the service. The description is typically used by tools to generate client code that accesses the service.
- Universal Description, Discovery, and Integration (UDDI) [12] is the publishing and discovery API for Web services
  - UDDI is a SOAP-based API for publishing and discovering Web services. The publishing API allows service providers to register themselves and their services. UDDI registry nodes replicate Web services information among themselves to provide the same information from any node. The discovery API allows service subscribers to search for available services. The UDDI registry provides the WSDL document, which allows the consumer to use the Web service.

Within this architecture, there are typically three participants: a service provider, a service broker, and a service requestor. The service provider creates a service and *publishes* it description in a UDDI directory. The service broker maintains the UDDI repository and acts a 'yellow pages' capability for Web services. The service requestor *finds* a service in the UDDI repository and then *binds* to that service. **(Need a diagram here)**

The standards provide the foundation for developments in this new paradigm. Built on that foundation will be the application structures and the solutions that drive businesses by integrating key business processes. In some cases, new solutions will be developed in this model in a greenfield approach. More frequently, solutions will be built by extending current systems, extracting value from the existing business rules [1] and databases, and combining with other applications across the Web.

Domains of Enterprise Component-based Development and Integration with Web Services

The authors have been engaged in implementing multiple component-based development and integration (CBDi) projects across diverse industry sectors in telecommunications, mortgage, financial services, government and banking. Over the past two years, with the rapid growth of interest in Web Services the integration side of CBDi has come to the forefront. One of the key success factors in these projects has been to manage issues and apply approaches and methods and many cases best-practices across five domains of CBDi [5] which have become the five domains of Web Services, namely:

Organizational – the project management, skill development, and change control, business-driven decisions that impact the growth and evolution of software within larger organizations. Also, this pertains to making sure that the selection of which services to expose is done correctly and ties the business and technology side of things together.

Methodology – The need to extend and refine current methods to utilize a smaller set of more pertinent artifacts and activities that will support the creation of a service-oriented architecture on top of a component-based paradigm which is currently not completely supported within object-oriented analysis and design methods that focus primarily on the design and creation of small-grained objects rather than larger grained Enterprise Components.

Architecture – The set of best practices and patterns to help build solid yet flexible designs that leverage the highly re-configurable architectural style that is synonymous with building Web services.

Technology Implementation – The choice of technology with which to implement the design; often a slew of technologies exists and need to be integrated.

Infrastructure and Tools – The standards, tools and technology that is needed to actualize this service-oriented paradigm. This includes development tools, gateway servers, APIs, middleware, browsers, etc.

Although Web Services and service-oriented architectures have definite applicability to both inter- and intra-enterprise contexts, it is important to address the migration to a web services architecture in a stepwise manner. Starting from within the enterprise, gaining the skills and expertise, overcoming the technological barriers and organizational issues first will insulate the organization from exposure to early adopter woes. The time to start adoption is now, but start inside the enterprise and work yourself outwards as you establish standards and culture for each of the above domains.

Business and Organizational Challenges

Although we will focus in this article on the technical challenges associated with the Web Services phenomenon, there is a corresponding set of business and organizational challenges that will need to be addressed as this technology proliferates.

Today, there is a clear boundary that designates ownership of data, process, and application code. As we move to a distributed model in which components are developed and integrated across silos, we will enable the sharing of key processes, thus streamlining business operations and reducing costs. The opportunity here is to re-use components, to automate business processes that today might require multiple manual operations, to transform manual or batch processes into efficient self-service queries, or to increase the visibility of a senior management team into core business operations parameters. Each of these opportunities can enable significant improvements in efficiency, productivity, and cost. However, they're not free. The corresponding challenge is to understand, in the context of this connected world, who has responsibility for the pieces, and even more important, who has responsibility for the whole. Where does responsibility lie for the overall architecture, for the development of the components? How is performance managed? And, who ensures that each of the service components is available and accurate during the execution of any particular end-to-end task? Organizations will need to be structured to support this new paradigm that distributes control and ownership. The notion of resolving the issues associated with an enterprise application will not be able to be solved independently within any part of the organization, but rather will require a management structure that facilitates shared responsibility.

Moving to the next stage of Web Services adoption, there is a scaling of both the opportunities and the challenges. Managing the distributed infrastructure and application domain within an enterprise has the significant advantage that all functions exist within the corporate firewall. Departments need to learn to trust their data and their processes to other departments within the organization, but they remain 'safe' from external intrusions. However, to exploit the full value of the service-oriented architecture is to move across the firewall, and to interact with processes made available from outside the organization, or to make internal processes externally available. The security and authentication capabilities that exist today are not sufficient for this level of interaction and need to be developed. But, once those technical issues are resolved, there will remain the business issues associated with who to trust, which processes to protect and not make public, what to consider proprietary, where benefits can be derived from sharing, which to trust from others, and how to best leverage the spectrum of possibilities that will be available. Businesses will need to sort through the value propositions associated with buying or selling access to processes. And, the community will need to confront the question of how to charge for Web Services in general. Today, nearly all of these services that have been made available are free. We will need to understand how and when to charge and the terms of use for Web Services.

Finally, the business potential of the dynamic interconnection of business processes that can continually re-form and re-connect is enormous. The result is a loosely-coupled and heterogeneous set of relationships between a company and its environment, whether that includes suppliers, customers, or partners. There may be, in this mix, changing partnerships depending on geography, product mix, or market situations, implying an agility that few organizations are prepared for today. Although the notion of a supply chain seems to indicate a fixed set of links, that will give way to a broader notion of sets of links that will be selected based on conditions at a particular time. Managing this level of complexity will require new models for business designs, and the organizations that support them. On the positive side, the breadth of possibilities for linkage may provide fast, cheap, and rich interactions. On the negative, this model will bring with it very complex boundaries and corresponding management challenges.

Mergers and Acquisitions. The result of mergers and acquisitions are multiple redundant systems with design mismatches that have to be somehow integrated to function in unison within the current I/T architecture. The challenge is to identify and isolate commonality and externalize variations so the variations can be configured and then applied to the application.

Methodological Implications

The implication on methods is that we need to chisel away some of the baggage of current methods and add some relevant artifacts and activities that are conducive to building a component-based architecture of large-grained, message aware, enterprise scale, highly-re-configurable Enterprise Components that

expose their services as Web Services. Internally, we build a Component Enterprise Architecture exposing its services for reuse within the organization. Externally a subset of those services will be exposed based on business drivers and competition; while not succumbing to expose every I/T asset that has been painstakingly developed within the organization and needs to be maintained for competitive advantage and liability reasons, among other issues.

Some additions include Subsystem Analysis, Variation-oriented Analysis and Design and Business Language Specification [4].

Integration Across the Extended Enterprise

Large businesses today are confronted by conflicting imperatives. The need to compete in the eBusiness environment brings with it demands from customers, suppliers, and business partners to access information that has been embedded within the internal workings of the enterprise. According to Gartner Group, 80% of the world's business continues to run on COBOL applications. These applications contain proprietary algorithms and techniques that offer competitive advantage and enormous databases of accumulated knowledge and historical data. The environment associated with those applications has tended to be batch processing, with little flexibility in the systems since they were not designed in modular fashion. In addition, maintenance can be costly as the skills base dwindles and the challenges of poorly documented systems are confronted. Nevertheless, significant business value is locked into the applications in the form of business rules, and those business rules need to be made available to a broader set of users than has historically been necessary.

A number of paths exist for organizations that are confronted with the need to move away from systems that have become costly to maintain and that no longer meet the needs of end users. Among the possibilities are the non-invasive techniques associated with screen-scraping, integration techniques at the transaction or application level, and large-scale replacement by package solutions. Each of these techniques has benefits and challenges. Screen-scraping is an easy and fast way to make legacy applications available to the web at the user-interface level, but it does not address the modification of the business process or data, and so has limited flexibility and can create maintenance issues. The integration techniques can be assisted by numerous products available in the market, but again do not capture business process knowledge, may have dependencies on vendors for implementations, and may not scale well. Probably the most robust option is to move to full packages, integrating a standard business process into the environment. This path can provide access to new and advanced functionality, but it requires abandoning investments in existing systems, and can be disruptive to implement. In addition, the ability for many packages to support web interactions is still evolving.

Emphasis is beginning to shift to legacy transformation, as companies explore different mechanisms to extract data and make it available to new constituents. Industries leading the charge here include financial services, insurance, and government. This strategy will include enabling legacy systems for integration within an enterprise (making the data and transactions enabled across the enterprise instead of being limited to the specific vertical application that the legacy code was developed for) as well as allowing the transformation of core business functions to a web-capable environment.

In an ideal world, businesses could preserve the investments they have made in their core systems over the past 20-30 years, while addressing the challenges with which they are confronted today. One path to that ideal is to focus on the transformation of legacy systems through a series of actions, with the goal being to encapsulate business rules into standalone components, which can then operate as individual web services in a flexible manner. Methods, tools and techniques have matured to a state of readiness for componentization and an evolutionary web services approach that exploits existing investments. We will address a number of the challenges in this approach, but first consider why it might be useful, and some of the steps required to make it work.

As an example of where the restructuring of legacy code into components could provide value, consider a large insurance company and the contrast between how they currently process claims and how they might prefer to have the processing managed. Today, the claims system might be a strategic Cobol program with over 1M lines of code. That system might collect flat files from a number of partner agencies. The system was designed to operated in a batch mode, with data feeds arriving once a day, and the individual flat files being merged into a single transaction file, which could then be processed in an overnight window. This process results in a minimum of 24 hours response time to customers. Access to the web has caused a population of customers to be unwilling to wait 24 hours for information that could be provided during a single online session. Trying to address the demand for shorter response times within the existing system would put severe strains on the system. This company needs to be able to extract the core business rules on which its claims engine is based, but change the operational process around those rules so that individual claims can be processed continuously, providing rapid response to customers, while evening the load distribution on the system since claim requests arrive throughout the day. The requirement to changing the process means that the techniques associated with screen scraping and non-invasive integration methods will not suffice. This company needs to identify the specific business rules that govern the process that they want to change, extract those rules from the legacy system, create new components to satisfy those rules, and then those components can expose the services they provide in various forms, the most promising of which is as web services; to be exploited dynamically and independently.

There are many opportunities in the legacy transformation domain. There is an existing and rapidly growing set of mining tools available today to help in legacy mining, integration and transformation. These tools can be used not only as a programming productivity tool, but as a management decision tool. Mining tools enable companies to understand their systems, by creating calling trees and maps of system components. Primarily, the tools available address the legacy understanding portion of the problem, including analysis to extract business rules. Approaches are being developed to create components from the extracted rules. It is important to note that in order to succeed with this approach, the tools themselves are to be applied within the context a recognized set of best-practices and legacy transformation methodologies.

The promise of Web services provides an opportunity to leverage the legacy transformation discussion. The use of components to encapsulate, extend, and carry forward legacy applications into Web-based applications by exposing their services as Web services is an emerging trend. It is easy to talk about legacy code and Web services at the philosophical level, but the practicality and feasibility of it remains to be demonstrated. Access to legacy enterprise assets such as databases and systems of record is a necessary but not a sufficient condition to enable enterprise components and services to succeed in the extended enterprise. Business process integration through the careful selection of enterprise component boundaries is critical to provide a layer of process extension across the extended enterprise, rather than merely data or information integration. Nevertheless, working through the challenges to enable the promise is extremely appealing, and would provide significant business value.

Integrating the New and Transforming the Old

The challenges of transforming “older”, legacy systems is further complicated by the need to integrate this transformation with the new development efforts that typically consist of n-tier architectures, application servers. This integration is at least on two levels: data integration, process integration. Access to legacy data is important; access to business rules locked within legacy systems is crucial to enable business process integration. To achieve this, the need for a component-based development and integration (CBDi) approach is necessary to enable development of newly integrated systems that leverage legacy assets and to guarantee that the necessary approaches, methods, architecture, technology implementations and infrastructure are there to support the integration and transformation with the legacy within the larger enterprise context. We will briefly describe such an approach in the following sections.

Integration of Services Within the Enterprise

For many large corporations that partake in component-based projects, the promise of components and services; leveraging legacy and exposing them to the agility of dynamic e-business lies not merely across business partners, but right within the enterprise. The enterprise itself consists of information silos that serve individual lines of business. Integration of services within an enterprise across multiple business lines (e.g., retail, wholesale) and product lines (e.g., financial services, credit card, travelers cheques, etc.) is best done by identifying the units of business functionality required and provided by each portion of the business in order to complete a business process.

Conceptual componentization at the business level is critical to achieve this enterprise agility and information and process integration capabilities that are crucial to dynamic e-business. Componentization consists of characterizing chunks of business-driven functionality that corresponds to business goals. This process is a first step towards identifying services within the organization that can be used by many, if not at, business lines within an organization rather than being locked within an information or application 'silo.' Unlocking these embedded services is often accomplished through Component-based Development and Integration (CBDi) where old systems are integrated to function with the newer ones often running on n-tier architectural platforms. This integration may be taken further than mere "wrappering" of functionality and may involve the refactoring of back-end business logic on hitherto unexposed services residing in legacy systems.

Often a common conceptual best-practice such as the Enterprise Component pattern can be used to provide the analysis and development teams a reference point for consistent design and development of large-grained enterprise-scale business components.

These components provide a set of services that other components from other product lines may customize, re-configure or directly reuse; with the preference being for the ability to rapidly re-configure components to be used within new contexts rather than any intrusive changes applied for customization. These services can be exposed at the department, corporate enterprise level as a migration path towards the extended enterprise.

Thus, in order to avoid the initial hurdles that new technologies often present, it is advisable to commence the adoption of Web Services from within the enterprise. This will bypass the current technology related issues of security, intellectual property exposure, and performance that follow from exposing web services outside the enterprise. Use the current state of the technologies to help the business without having to deal with the details of security and performance within the enterprise, especially if there exist communication infrastructure (such as message-oriented middleware) adequately supporting internal non-functional requirements.

Migrating to a Service-Oriented Architecture

The process of strategic migration to a service-oriented architecture can be seen as having five levels of evolution as depicted in [17]. In the first level, data transfer between legacy "silo" systems occurs in batch mode. There is little processing of information relevant to the other silos; merely raw data being transported often for ELT (Extract Load Transform) scenarios. The next level or phase (in this case) is information flow coordination where an enterprise application integration hub-and-spokes architecture may be put into place or handled with other rudimentary technologies that effectively do the same kinds of things.

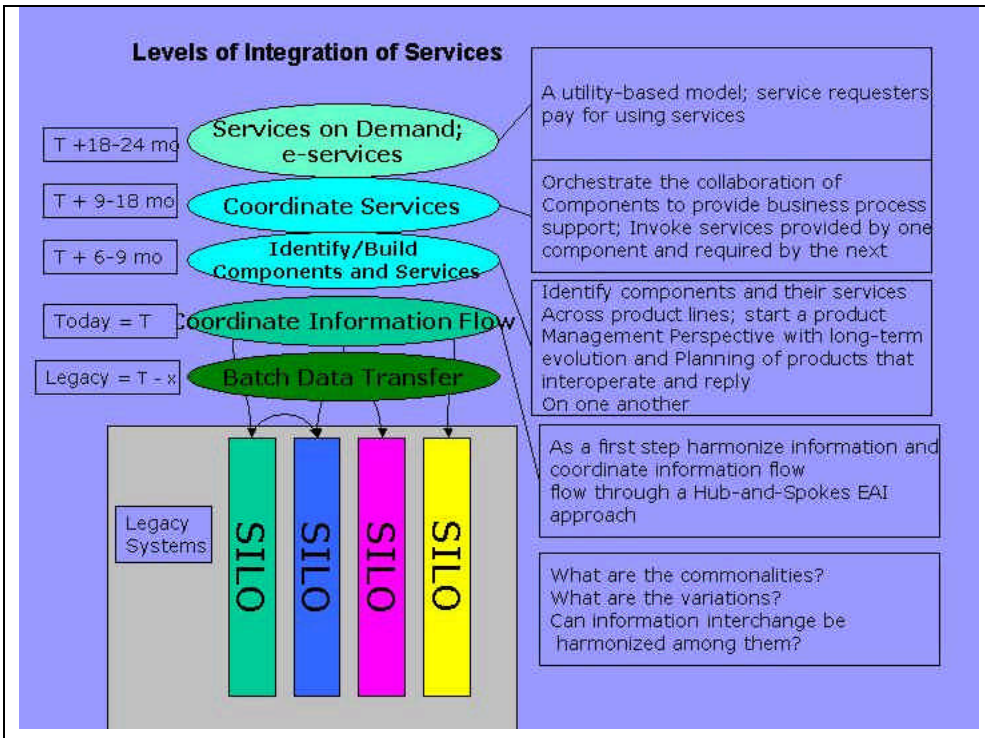

Figure 1 : Levels of Integration of Services, The Strategic Path to E-services

Figure 1Caption: "Experienced driver on a closed course!"
The authors have experienced these kinds of durations on real projects. It should be realized that these timeframes are not typical and come only with experienced teams and skills. The durations for projects without these criteria of best-practices and skills may vary.

Current Architecture: "Where we are"

In order to migrate from the more static state imposed by legacy systems to a more dynamic architecture and process required in today's competitive market, the current level of the enterprise architecture must be identified, characterized and inventoried to facilitate the identification of functional areas that partition the domain into business lines and product lines. The data interaction needs of these systems can be coordinated through a hub-and-spokes architecture that takes the enterprise from data transfer to information coordination. The coordination of information is then considered in light of business processes and the boundaries of enterprise scale components that collaborate to create a business process that maps back to business goals.

There are a number of tools that are being introduced into the marketplace and yet tools alone will not solve organizational or methodological problems. Tools can help you make the system work; but not creating a robust, architecturally sound design. Web Services and its tools are not a substitute for sound architecture and strong life-cycle management and methodology within the software development life-cycle. Tools or infrastructure will not migrate enterprise architecture from data transfer across legacy silos to service-based models of inter and intra-enterprise computing. Thus, the impact from five dimensions have been determined to be known and prepared for: organizational aspects (such as project management implications, education programs), methodology aspects (such as extension to current methods to provide full-life-cycle support for component-based development) , architectural aspects (such as best-practices and patterns such as Enterprise Component and the issue sin creating scalable architectures) as well as Technology Implementation aspects which cover how to map a given design onto a technology standard such as Enterprise JavaBeans or .NET.

One of the typical issues that is often encountered is not only that of component granularity but of service granularity: "should we expose a method on a class as a web service?" Or what is the right level of service granularity?

Component-based Architecture: "The Next Step"

The partitioning of the business domain into a set of sometimes overlapping subsystems that correspond to business process boundaries leads to a natural separation of concerns for the domain in question. For example, a financial services company can differentiate between its various product lines and business lines by considering the partitioning of Enterprise scale business components (called *Enterprise Components*): Customer Management, Account Management Product Management, Security Management, Billing, Rating could all provide the boundaries to create large-grained Enterprise Components as depicted in [17] that encapsulate a set of loosely coupled and mediated medium- to small-grained components, objects or proxies and adaptors to legacy systems.

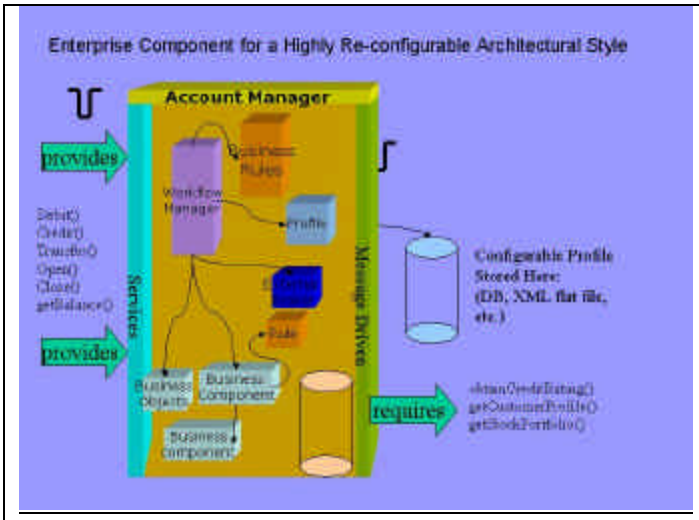

Figure 2: Enterprise Component Pattern Enables the Creation of a Highly Re-configurable Architectural Style

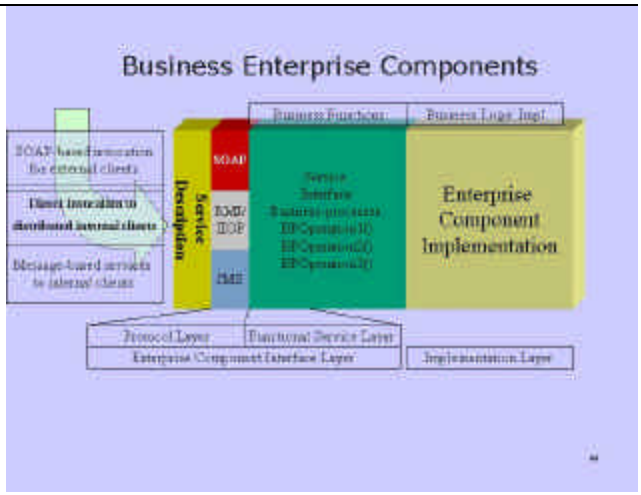

Figure 3: Enterprise Components provide multiple protocol access to business rules and functionality

Service-oriented Architecture: "The Future"

Enterprise Component services can be expressed and exposed for invocation within the enterprise using WSDL. Such service descriptions are often defined in an internal UDDI registry. How much of the set of services and which ones to expose to the outside world of business partners, suppliers and customers (consumers) is then driven by business imperatives, roles, business rules and competitive analyses. In this scheme, there is a gradual process in which the WSDL residing in the internal UDDI registry is migrated out to the public or partner UDDI registry for greater public exposure where that service provision is guaranteed to provide competitive advantage or otherwise address business needs.

Once the service has been defined and exposed for finding and binding and ultimate invocation, an essential question arises: "what protocol should be used to perform the invocation?" This is an important design decision that impacts functional and non-functional requirements of the application environment and the business model and its goals as realized through the I/T structure. Many of these requirements can be defined in a configurable fashion through rapid re-configuration of a the EC's Configurable Profile in order to ensure that the component has the characteristics of self-description,

rapid collaboration alteration and dynamic configuration. However, this is not the only means of achieving a robust yet pragmatic architecture that satisfies service level agreements (SLAs).

Standards and technologies such as the Web Services Invocation Framework (WSIF) [13] or Web Services Inspection Language (WSIL) help support protocol transparency for achieving acceptable levels of service without compromising the flexibility provided by this highly re-configurable architectural style that dynamic e-business provides.

WSIF provides a way to describe the underlying services directly, and also how to invoke them independently of the underlying protocol or transport. The ports and bindings of WSDL are extensible. WSIF allows the creation of new grammars for describing how to access enterprise systems as extensions of WSDL. This allows the enterprise developer, for example, to use a tool to create a description of the CICS transaction, EJB or other service directly. The user of the service builds their application or process based on the abstract description. At deployment time or runtime, the service is bound to a particular port and binding.

WSIF is an extensible framework, which can be enhanced by creating new "providers". A provider supports a specific extensibility type of WSDL, for example SOAP or Enterprise JavaBeans. The provider is the glue that links the code the developer wrote to use the service to the actual implementation available.

The Web Services Inspection Language complements the discovery features of UDDI . UDDI provides for a global or wide-scale directory of services. Analogous to large-scale web-content directories such as the Open Directory Project, UDDI is key to finding and identifying services. However, it is also important to be able understand which local services are available on a particular site. The Inspection standard offers this ability to query a given site or server and retrieve a list of available services. The list of services includes pointers to either WSDL documents or UDDI service entries. WSIL is a useful lightweight way of accessing a service description without requiring access to a UDDI server. Most person facing Web sites provide a "site map" to help guide users. WSIL is a similar function for programs that wish to explore a site.

A Financial Services Industry Project Outline

Next we will outline a typical financial services project within which a heterogeneous environment of legacy and Greenfield applications are functioning or being built. In one such project, the organization had five back-end legacy systems running on different hardware and software platforms. Each application was built piecemeal as a silo to address a particular business need. Many of them had been added to the I/T infrastructure through mergers and acquisitions, resulting in an inordinate amount of redundancy which not only led to maintenance difficulties but presented challenges when new functionality was added. In some cases this functionality had to be replicated across all systems.

Most of the organization's business knowledge was locked within the business rules embedded within these five legacy applications. But the business rules did not have access points and could not be invoked from other applications that needed the same functionality.

Furthermore, the enterprise was encountering serious competition from smaller, more "web-aware" rivals. They needed to make the back-end legacy applications accessible through the web. Furthermore, the interface to the web user had to be uniform; they were to experience one system not five different ones. This was difficult because the business processes and business rules for each system were different and had not been architected to function in unison or even coordinate information flow other than through nightly batch processes. This created lengthy turnaround times to the customer; frequently business partners who were relying on the timeliness of services.

Using the CBDi approach and addressing the domains of CBDi, the back-end legacy systems were inventoried, knowledge mined, message-enabled and componentized as shown in . This was done in preparation for interaction with a set of J2EE style program running on an application server that could now access the chunks of business functionality and business rules that were hitherto locked within the

legacy systems. Furthermore, this facilitated a common unified user-experience that made it appear as if there was one single back end systems handling everything.

Enterprise components as shown in Figure 2 and Figure 3 were built to encapsulate the functionality of middle-tier business logic that would encapsulate the utilization of back-end legacy systems.

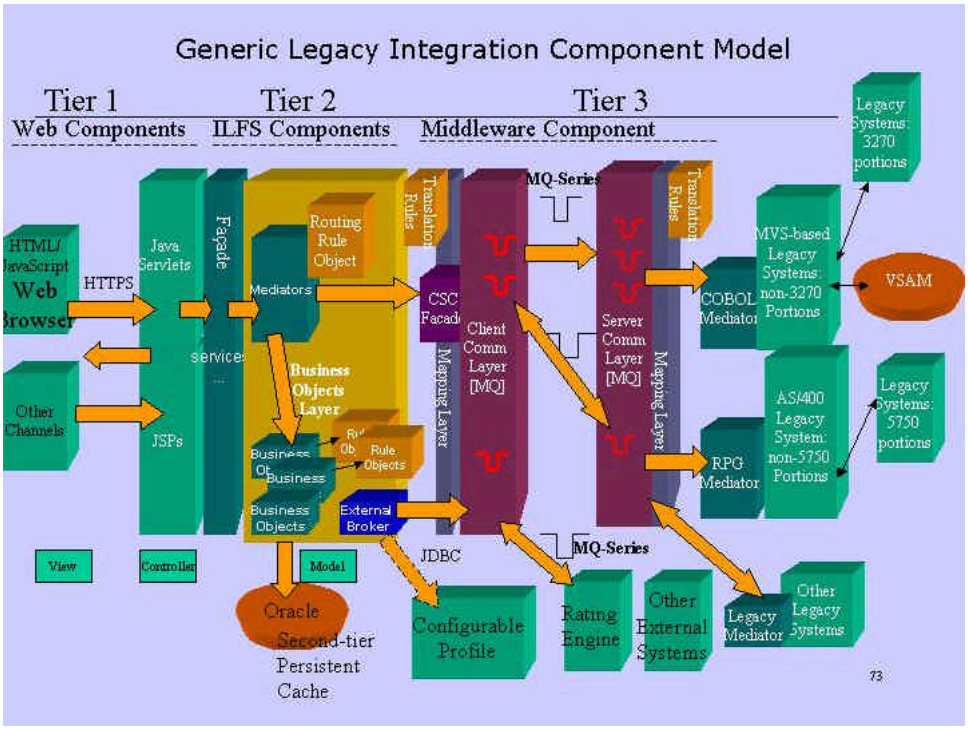

Figure 4 : Target Architecture

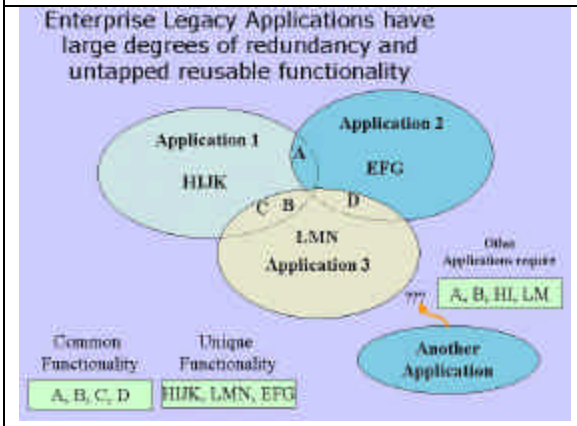

Figure 5: Legacy Applications with inaccessible and redundant functionality

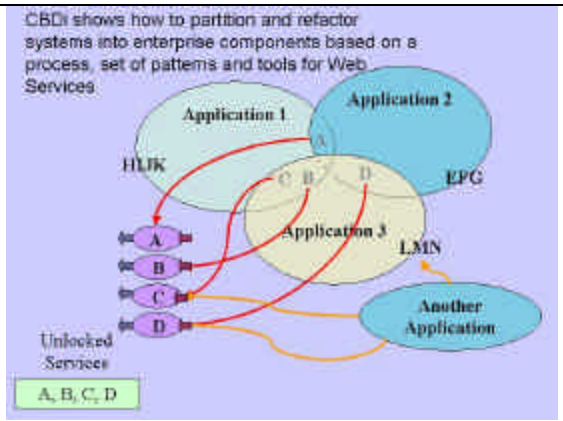

Figure 6: Legacy Transformation : Componentizin Valuable Legacy Assets

Recurrent Themes

Within the multiple systems of larger organizations reside legacy applications with overlapping functionality. The redundant functionality can be grouped into the services of an Enterprise Component that will serve as an asset providing a consistent, reusable and re-configurable interface into its services required by internal and external clients.

The as-is architecture for a typical Customer Management system consists of a set of redundant systems per business line: card services, financial advising services, mortgage, insurance, etc., that all have a customer information component to them. Each hold slightly different data elements and interact with different business rules and in the context of different processes.

Business partners would like to integrate directly with each of these systems in a uniform manner to, for example, provide services for the financial services company's customer loyalty programs (frequent user points) third party vendors would like to dynamically invoke and utilize a certain subset of services that are exposed as web services.

A highly re-configurable and consistent Enterprise Component consolidated the access to data services and business rules that varied from product line to product line. Multiple ECs were defined following a domain analysis that partitioned the domain into its business process level of granularity subsystems.

Technical Challenges

The current generation of web service infrastructures and tools have the obvious set of problems one would expect to see from early software: performance, interoperability, and usability. We will take as an example the problems associated with performance. The basis of web services is XML, which represents all data as tagged text. Both the tagging and the text representation cause an "explosion" in data sizes when compared with carefully crafted binary representations. More data means slower information transfer rates. Add to that the need to parse the XML data, whenever it is read into an application or system, to create the internal representations of the data, and you have more overhead whenever data goes between independent applications (or services or systems). Further complicating the performance picture is the need to read in and parse the definition of the tag definition set (the DTD), if it has not already been read in, parsed, and cached by the application. Security (encryption and de-encryption) causes even greater overhead. All these performance issues can and will be addressed as Web services technology matures, but today developers can expect factors of 10 to 100 slow down compared to conventional distributed computing operations (such as remote procedure calls using RMI or IIOP).

Beyond these initial problems, the future direction of Web services holds more serious technical challenges. Eventually, Web services will grow beyond the new distributed computing infrastructure of SOAP, WSDL, UDDI, etc., to become electronic utilities (eUtilities) that are delivered to end users over the Internet. EUtilities represent a critical new application domain in electronic commerce. Like traditional utilities, such as telephone and electricity, Web services will be metered and customers will pay for their use of the eUtility.

The terms of use (called service level agreements, or SLAs) of the eUtilities will include functionality, availability, scalability, performance, resources, and reliability. These terms of use may vary from customer to customer, and they may change over time. This, in turn, necessitates dynamic monitoring of eUtilities, dynamic control over SLAs, and the ability to respond quickly to changing customer needs and available resources. Providing these important eUtility capabilities impose some challenging requirements on the design, development, deployment, and evolution of web services. SLAs represent a legal description of a service—not simply in terms of its functional interface, but also in terms of performance, payment, legal consequences of non-compliance, and levels of support and access. Because SLAs are legal agreements, it must be possible to monitor Web services to verify that the services being provided conform to those that were negotiated, and to redress any conformance failure immediately.

As soon as Web services must satisfy serious, demanding SLAs, developers will have to confront the task of designing, building, configuring, deploying, and managing software that must meet, and continue to meet, constraints beyond the simple functional correctness with which we have been struggling for 50 years. Web service applications must include both the conventional APIs, and also interfaces for metering, performance, manageability, auditability, replication, and reliability. These interfaces, though distinct, reflect capabilities that must interact with one another in deep and profound

ways. Supporting the description, realization, integration, and separate evolution of these interfaces will be a major challenge facing eUtility engineers.

To increase the complexity even further, we note that Web services must be dynamically configurable. Just as the telephone company depends on the electric utility in providing its phone services, so future Web service applications will depend on eUtilities provided by others. So the environment in which this software runs will be subject to change without notice – both in terms of the underlying eUtilities and in terms of the number and kind of users (where some of those users may be higher-level Web services). Other services on which a given eUtility depends may disappear or may change in some way that affects the eUtility, and it must be able to detect and respond to such changes whenever possible. When it is not possible, they must be able to degrade gracefully, since organizations may rely on them for critical parts of their business.

Note that the following discussion is based upon terms from object-oriented (OO) technology. OO provides us with languages and methodologies for describing, creating, and using self-contained entities with clean definitions. From the point of view of Web services, however, not all objects are created equal. Avoid thinking of objects in the classical text-book "small" sense (like the Cartesian-coordinate point object with methods for pen-up, move, and pen-down). Web services need larger-grained objects – objects that represent a business function (like a customer account or a purchasing service) or a large IT component (like a security or authentication service). There are two main reasons for this perspective: overhead per service (described above) and the constraints of the Web service design methodology.

Our description of the technical challenges associated with the software engineering of Web services is based on, but goes beyond, notions Multidimensional Separation of Concerns (MDSOC) [6,7] and the allied discipline of Aspect-Oriented Software Development (AOSD - notably, Hyper/J™ and AspectJ™ [8])). MDSOC is a convenient language for describing the issues involved. MDSOC supports the identification and creation of "crosscutting" concerns that affect many modules throughout an OO system. By modularizing aspects of a system that do not conveniently fit the dominant hierarchy (e.g., OO inheritance), one can avoid duplication and error in implementation and isolate changes during maintenance. Various aspects are then woven into the base code to form a new program. In fact, a well designed MDSOC system eliminates the distinction between "base code" and "aspect code" – the system is built out of a fabric woven from the different concerns. MDSOC will be useful in the definition of particular Web services, for example interfaces for one particular subset of customers of the eUtility. Different Web services that must ultimately work together will be developed independently (and will be able to run as standalone applications), by different organizations, so MDSOC's ability to permit the separation of interacting concerns, such as a monitoring service from one supplier along with a multileveled-performance service from another supplier, will be essential to the Web service development process.

We believe that the engineering of Web services will require developers to identify a set of concerns—both functional and non-functional, such as performance, scalability, security, manageability, and monitoring; implement the software to realize and address these concerns, and to integrate some set of these concerns together—possibly dynamically—to produce Web services that can be configured differently to meet the needs of different customers.

In this sense the underlying principles of "design for change" implementing a highly-re-configurable architectural style needs to be adopted. We expect that performance management, for example, will impact many portions of an eUtility (what level of resources are available for a user or class of users, who has legal requirements that must be met, and so on). Hence, collecting those issues together in a concern can serve both as a design, implementation, and evolution tactic. Dependence on remote services can also be modularized as another dimension, so that the underlying application can be prepared to switch between a variety of competing (or failing) suppliers – this concern, then, becomes a configuration/deployment tactic. Providing run-time interfaces during execution for monitoring/control will meet the legal needs without cluttering up the functional APIs. Finally,

maintenance will enjoy the usual simplification that having a multidimensional separation of concerns provides.

Issues in building and deploying eUtility applications

To motivate the issues we are addressing, we begin with a description of our view of an object-oriented Web service (OOWS)[1]. As shown in Figure 7, we view an OOWS as a software component that is built on top of a highly reliable infrastructure and possibly other OOWS, providing such services as those listed above. Like other kinds of components, OOWSs provide functional interfaces (APIs), and each API supports some set of functional interfaces. Unlike other components, however, OOWSs must be dynamically controllable by SLAs, so that they can change their behavior and characteristics in response to changes in SLAs.

Thus, these components must also provide non-functional management interfaces either as direct programming interfaces (similar to the APIs above) or through configurable profiles (using a re-configurable architectural style through the use of manners and Enterprise Component Configuration Profiles [13]). These non-functional interfaces or profiles will permit control over such attributes as performance, scalability, reliability, monitoring, metering, level of service, and potentially, the set of capabilities the OOWS provides including the composition of particular components using which it is constructed. Because SLAs may change at any time, it must be possible to control a service's functional and non-functional properties at any time during the software lifecycle—build-time, integration-time, or runtime.

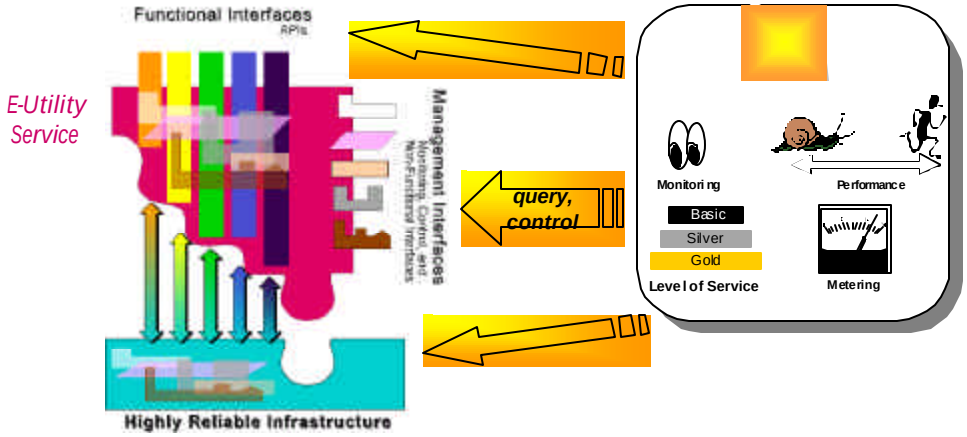

Figure 7: Web Services (eUtilities), SLAs, and Interaction Among Functional, Management, and Infrastructure Interfaces

Interaction among functional, management, and infrastructure APIs: Web services challenge the software engineering models that have dominated application development for traditional clients and server platforms. In particular, note that traditional applications are specified and implemented in terms of a single set of functional requirements. These requirements are generally uniform for all end users; thus, these applications define one or more functional interfaces (APIs). A Web service (and the components that comprise it) also addresses traditional functional requirements, but each customer also negotiates for the particular functionalities, levels of service, performance and reliability that they will obtain from the service, and they will pay for their use of the service. Thus, a Web service must define one or more functional APIs (possibly different ones for different customers), but it must also present non-functional interfaces that permit control over performance, reliability, metering, and level of service. These interfaces must interact with one another in some deep and fundamental ways, as suggested by Figure 7. For example, obtaining high performance may

[1] Remember that we are thinking of large-grained objects in this discussion.

necessitate the use of different communication protocols, search algorithms, and concurrency control models, or it may preclude the use of certain functions that cannot be made to achieve that performance. The definition and support for such deeply interacting interfaces presents a significant challenge to service engineering and deployment.

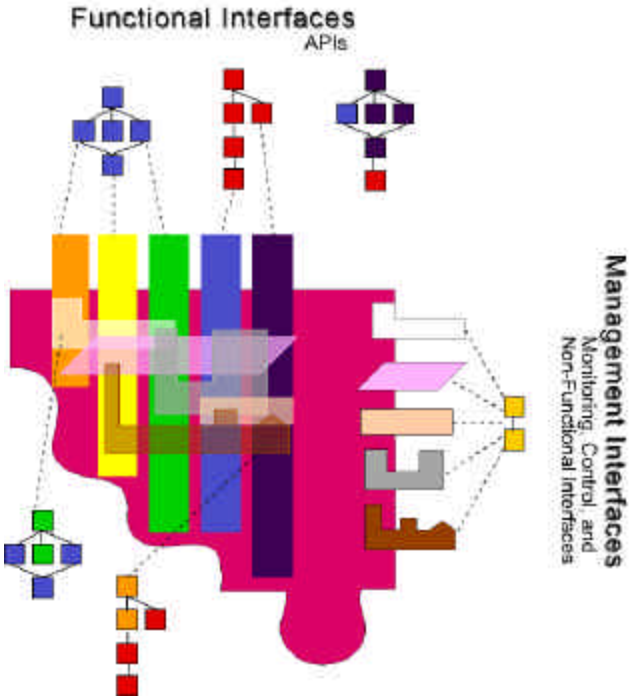

Figure 8: Different capabilities are implemented using different domain models and different class hierarchies. These hierarchies are reconciled and integrated when they are composed.

Figure 8 suggests two key characteristics of Web services. One is that capabilities embodied in the non-functional management interfaces may crosscut capabilities present in the functional APIs, making them good targets of MDSOC technologies. The other characteristic is that some functional and management capabilities may require their own class and interface hierarchy to implement the necessary domain model, and that these different hierarchies must be integrated as the capabilities are combined into a particular OOWS. In fact, the same issue arises when different OOWSs are composed together to help build a new service. This is likely to be a critical problem for OOWS developers, who do not have control over the domain models used in underlying eUtilities.

While both MDSOC technologies are promising, they do not, at present, fully address all the critical issues in the interaction of functional and management interfaces:

- Semantic mismatch: A semantic mismatch will occur when two or more capabilities have mutually incompatible assumptions or requirements. For example, a client may negotiate for a level of service that cannot be attained with a given concurrency control mechanism or with the particular strategy employed to implement some feature. For OOWSs, it is necessary to be able to identify semantic mismatches *before* they bite a client. Thus, for example, it is important to be able to tell that a particular set of requirements in a client's SLA cannot be satisfied simultaneously, or that doing so will necessitate new components, algorithms, or infrastructure, and this must be determined before the capabilities are promised to the client. It is imperative to be able to detect semantic mismatches early enough to fix them, or to prevent agreement on an unsatisfiable SLA.

- Interference: Interference is a common kind of semantic mismatch problem in concurrent software, but there is real potential for interference to occur in OOWSs between and within the

management and functionality interfaces. It happens when an action taken by one part of the software interacts with another's actions in an undesirable manner, causing an effect that does not occur when the actions occur independently. An example of interference can be found when messaging and transactions are present together. To allow a transaction manager to preserve atomicity and serializability, messages should not generally be sent until a transaction commits. A messaging capability could easily send messages independently, thus interfering with the ability of the transaction manager to satisfy its requirements.

- Unpredictability: As the discussions of semantic mismatch and interference suggest, it is not always possible to determine, *a priori*, what a piece of composed software will look like. Unlike non-compositional paradigms, where one does not get behaviors that one did not directly program (excluding problems like concurrency errors), developers using compositional paradigms will encounter circumstances where their composed software behaves unexpectedly and unpredictably. This is, in part, because compositors add logic, as noted earlier, but it is also because compositors break existing encapsulations to integrate concerns. The developers of those encapsulations made certain assumptions about the behavior of the module, and those assumptions are easy to violate when code from new concerns is interposed within an existing module. The unpredictable effects of composition tend not to be found until they manifest as erroneous behavior at runtime. This will not be acceptable in an OOWS context.

Verifiable and controllable conformance to SLAs: Each customer's contract for a service is described and governed by an SLA (see Figure 7). It must, therefore, be possible to determine whether the appropriate SLA(s) is/are being satisfied, to determine how and why it is not being satisfied, and to control the service to bring it into compliance with the SLA(s). This suggests again the need for programmatic interfaces to control metering, performance, and other crosscutting, non-functional capabilities. It also potentially requires dynamic addition, removal, and replacement of capabilities. The interaction between these management capabilities and the functional capabilities is again apparent here, since an OOWS may not provide certain functionality or performance for users who request lower levels of service.

Building services when the "components" are neither local nor locally controllable: As with all software, it is reasonable to assume that "composite" OOWSs may be built—i.e., where one service depends on other (lower level) services, each with their own SLAs. Hence, the designer/builder of a service may—indeed, should—spend more effort in integrating services than in constructing new ones from scratch. But this designer/builder is confronted with a body of services (comparable to traditional components) that may not be under his/her direct control, since some else may own the service and may be delivering it over the network. Thus, such a designer/builder will not be able to use traditional software engineering methodologies in developing composite services, since these methodologies depend on the centralized control and static deployment assumptions that clearly do not hold.

Inevitable, unpredictable, dynamic changes and need to limit the impact of such changes: Any given OOWS may change or become unavailable without notice, thus potentially causing a cascaded impact on all the services that depend upon it. Services are subject to significant reliability and availability constraints, so the change or loss of an underlying service cannot, in general, be allowed to crash or impede other services. Thus, services must come with integration and runtime support systems that can identify service change or loss and respond to it. This likely includes, though is not limited to, the ability to handle real-time version control, configuration management and "hot swapping" of bits and pieces of services, and the ability to upgrade and degrade gracefully. Note again that changes and unavailability affects both the functional and non-functional (management) aspects of a dependent service (and the legal implications of such a change). The version control and configuration management problem is considerably more complex than its traditional build-time analogue. At build-time, a version control or configuration management system need only be

concerned with choosing a set of modules to put together, but at runtime, it must be possible to replace much smaller-grained pieces, to ensure the lowest impact of change on the modified service (the larger the pieces that are replaced, the larger the potential impact on other parts of the service, and on any services that depend on it). Advanced configuration management approaches, like [9], may be of use here.

SLAs are software, too: SLAs themselves must be considered a key component in the software engineering of services. In particular, they both define and configure the functional and non-functional aspects of a service. They are, therefore, a specification for the service, and it must be possible to ensure that they are satisfied both statically and dynamically. If SLAs are to be monitored and their requirements satisfied, they must be treated as software artifacts in their own right—perhaps as declarative specifications of services, or as some kind of operational semantics. In either case, they must have their own build/integrate/test/deploy cycle, which must tie in directly with the capacity planning, design, and architecture of the OOWS, the monitoring/execution/control of the service, and the compliance checking and reporting to the provider and the end-user of the service. Given the competitive nature of the first generation of OOWS-like vendors (ASPs), one can expect rapid evolution of SLAs (at least in terms of cost of service options provided). Hence these SLA "programs" will have to accommodate change during execution.

Conclusion

The evolution of enterprise architectures to capitalize on the capabilities of Web Services to provide strategic advantage for leveraging current assets and assimilating them into a dynamic structure of services on demand is well under way. New technologies and methods are maturing to achieve acceptable service level characteristics. One of the more successful ways of designing and implementing web services is to start with a component-based architecture of large-grained Enterprise Components that expose business process level services as web services. Start using these within the organization rather than exposing them externally. With more project experience in this domain, best practices are uncovered and utilized with increasingly greater success. Then the decisions to migrate to a full service-oriented architecture that externalizes useful business services can be made and a subset of the internal enterprise services can be migrated outwards.

References

[1] Arsanjani, A., Rule Object: A Pattern Language for Flexible Modeling and Construction of Business Rules, Washington University Technical Report number: wucs-00-29, Proceedings of the Pattern Languages of Program Design, 2000.

[2] Arsanjani, A., Alpigini., J., "Using Grammar-oriented Object Design to Seamlessly Map Business Models to Software Architectures", *Proceedings of the IASTED 2001 conference on Modeling and Simulation*, Pittsburgh, PA, 2001.

[3] Arsanjani, A., "CBDI : A Pattern Language for Component-based Development and Integration", European Conference on Pattern Languages of Programming, 2001.

[4] Arsanjani, A.; "Grammar-oriented Object Design: Creating Adaptive Collaborations and Dynamic Configurations with Self-describing Components and Services", Proceedings of Technology of Object-oriented Languages and Systems 39, 2001.

[5] Arsanjani, A.; "A Domain-language Approach to Designing Dynamic Enterprise Component-based Architectures to Support Business Services", Proceedings of Technology of Object-oriented Languages and Systems 39, 2001.

[6] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. "*N* Degrees of Separation: Multi-Dimensional Separation of Concerns." In Proc. ICSE 21, May 1999.

[7] H. Ossher and P. Tarr. “Multi-Dimensional Separation of Concerns and the Hyperspace Approach.” In *Proc. Symp. Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2001.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. “An Overview of AspectJ.” In Proc. ECOOP 2001, June 2001.

[9] M.C. Chu-Carroll and S. Sprenkle. “Coven: brewing better collaboration through software configuration management.” Proc. 8th International Symposium on Foundations of Software Engineering, 2000.

[10] http://www.w3.org/TR/SOAP

[11] http://www.w3.org/TR/wsdl

[12] http://www.uddi.org/about.html

[13] Web Services Invocation Framework, http://www- 06.ibm.com/developerworks/library/ws-wsif.html