Web

BY ALI ARSANJANI, IBM
BRENT HAILPERN, IBM
JOANNE MARTIN, IBM
AND PERI TARR, IBM

Much of web services' initial promise will be realized via integration within the enterprise, either with legacy applications or new business processes that span organizational silos. Enterprises need organizational structures that support this new paradigm.

Web services are the latest software craze: the promise of full-fledged application software that needn't be installed on your local computer, but that allow systems running in different environments to interoperate via XML and other web standards. Much of the hoopla surrounding web services revolves around the nirvana of inter-organizational distributed computing, where supply chains can be integrated across continents with applications built from small parts supplied on demand by various vendors. To get to this place, we need to chisel down current methods and build a component-based architecture of large-grained, message aware, enterprise scale, and highly re-configurable enterprise components exposed as web services.

The time to start adoption is now, but start within the firewall, inside the enterprise, and work yourself outwards. This wisdom will insulate you from the as yet unresolved security, intellectual property exposure, and performance issues associated with exposing web services outside the enterprise. It also gives you ample time to establish standards and best-practices. Over the past two years, we have contributed to several component-based development and integration (CBDi) projects in the telecommunications, mortgage, financial services, government, and banking sectors (5). A key success factor in these projects, one of which we will discuss in this article, involved applying CBDi best-practices across the following five web service domains:

- Organizational, including project management implications and education programs;
- Methodology, including extending methods to provide full life cycle support for component-based development;
- Architectural, including best-practices and issues in creating scalable architectures;
- Technology implementation, which involves mapping a given design onto a technology standard such as Enter-

The background of the page is a complex, abstract composition in various shades of red and maroon. It features a dense arrangement of overlapping geometric shapes, including polygons, rectangles, and lines, creating a sense of depth and complexity. In the lower right quadrant, there is a prominent wireframe sphere, resembling a geodesic dome or a molecular structure, which stands out against the darker background. The overall aesthetic is technical and futuristic, suggesting themes of technology, architecture, or data.

Services

Promises and Compromises

Web Services

Promises and Compromises

prise JavaBeans or .NET.

- Infrastructure, including development tools, gateway servers, APIs, middleware, and browsers.

THE LEGACY TRANSFORMATION IMPERATIVE

The idea of clearly defined ownership of data, process, and application code will shift as we move to a distributed model, where we can reuse components to automate business processes. Also the fixed-link notion of a supply chain will give way to a more agile notion of links based on particular conditions to provide fast, cheap, and rich interactions. This shift is inevitable: the business potential of the dynamic interconnection of processes that can continually re-form and re-connect is enormous.

While this evolution will minimize multiple manual operations and transform batch processes into efficient self-service queries, among other things, it requires a clear understanding of who has responsibility for the pieces, and even more important, for the whole. The redundant systems and design mismatches introduced via mergers and acquisitions also need to be integrated within the current I/T architecture. The challenge is to isolate commonality and externalize variations so they can be configured and applied to the application. Departments also need to learn to trust other organizational departments, and as security and authentication capabilities enable expansion outside the firewall, the issues around trust will grow more complex.

The need to compete in the eBusiness environment brings demands from customers, suppliers, and business partners to access internal information. But according to Gartner Group, 80% of businesses run on COBOL applications, which tend to require batch processing, and have little flexibility since they were not designed in modular fashion. Several paths exist for organizations to move away from costly systems no longer meeting end-user needs. Among them 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 technique has pros and cons. Screen-scraping is a fast way to make legacy applications available at the user-interface level, but it does not address business process modification, and so has limited flexibility. The integration techniques can be assisted by numerous products available in the market, but again do not capture business process knowledge, may depend on vendors for implementations, and may not scale well. Probably the most robust option is to integrate 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. In addition, the ability of packages to support web interactions is still evolving.

Emphasis is shifting to legacy transformation, as industries such as financial services and insurance explore ways to extract data and make it available to constituents. This strategy involves enabling legacy systems for integration within an enterprise, and eventually transforming core business functions to a web-capable environment. Businesses can preserve their core system investments, by transforming legacy systems through a series of actions, encapsulating business rules into flexible, standalone components that operate as individual web services.

To see the value of restructuring legacy code into components, consider how a large insurance company processes claims. Its claims system is a strategic Cobol program with over one million lines of code. The program is designed to operate in a batch mode with data feeds arriving daily, and flat files merged into a transaction file processed overnight. This process results in a 24-hour minimum customer response time. To shorten this time, and even the load distribution on the system, this company needs to extract the core business rules of its claims engine, and adjusting them to process claims continuously.

Mining tools can help in legacy mining, integration, and transformation. These increasingly popular tools are not just programming productivity tools, but also management decision tools. If they are applied within a recognized set of best-practices and legacy transforma-

tion methodologies, they enable companies to understand their systems, by creating calling trees and maps of system components, and to perform analysis to extract business rules. 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.

MIGRATING TOWARD WEB SERVICES

The challenges of transforming legacy systems is further complicated by the need to integrate new development efforts consisting of n-tier architectures and application servers. Integrating services within an enterprise across multiple business lines is best done by componentization, which involves characterizing chunks of business-driven functionality corresponding to business goals. This process helps identify services that can be used by multiple business lines instead of 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 running

on n-tier architectural platforms[3,5]. This integration involves more than mere “wrapping” of functionality, and may include refactoring back-end business logic on unexposed legacy services.

Often a common conceptual best-practice such as the enterprise component pattern can give development teams a reference point for developing large-grained enterprise-scale business components. They can be customized, re-configured or directly reused, at the department or corporate enterprise level, and as a migration path towards the extended enterprise.

The evolution of strategic migration to a service-oriented architecture has five levels as depicted in Figure 1. First, data transfer between legacy “silo” systems occurs in batch mode, with little processing of information relevant to the other silos, except raw data being transported for ELT (Extract Load Transform) scenarios. The next level is information flow coordination, where the enterprise architecture is identified, characterized and inventoried to identify the functional areas of business and product lines, and an enterprise application integration hub-and-spokes architecture is set up. This architecture takes the enterprise from data transfer to information coordination. Information coordination is then considered in light of business processes and the boundaries of en-

terprise scale components that collaborate to create a business process that maps back to business goals.

Such partitioning leads to a natural separation of concerns. A financial services company, for example, can differentiate between its various product and business lines by considering the partitioning of enterprise components (see Figure 2) such as customer, account, product, and security management, as well as billing and rating. These components provide boundaries to create large-grained enterprise components as depicted in (see Figure 3) that encapsulate a set of loosely coupled and mediated medium- to small-grained components, objects, or proxies and

Web Services

Promises and Compromises

adaptors to legacy systems.

A future aim is to expose enterprise component services for invocation within the enterprise using WSDL. Such service descriptions are often defined in an internal UDDI registry. The issue of how much to expose to the outside world of business partners, suppliers, and customers is driven by business imperatives, roles, rules and competitive analyses. In this scheme, the WSDL residing in the internal UDDI registry gradually migrates to the public or partner UDDI registry for greater public exposure.

Once the service is defined and exposed, the question of what protocol should be used to perform the invocation arises. Many of these requirements can be defined in a configurable fashion through rapid re-configuration of the EC's Configurable Profile to ensure the component has the characteristics of self-description, rapid collaboration alteration, and dynamic configuration. But this is not the only means of achieving a robust yet pragmatic architecture that satisfies service level agreements (SLAs).

Technologies such as

Enterprise Component For A Highly Re-configurable Architectural Style

The enterprise component pattern enables the creation of a highly re-configurable architectural style.

Business Enterprise Components

Enterprise components provide multiple protocol access to business rules and functionality.

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 flexibility. WSIF provides a way to describe the underlying services directly, and to invoke them independent 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 WSDL extensions. This allows the enterprise developer to directly create a description of the CICS transaction, EJB, or other service, and the service user to build an application based on the abstract description. At deployment or runtime, the service is bound to a particular port and binding. WSIF's extensible framework can be enhanced by creating new "providers," which support a specific extensibility type of WSDL, such as SOAP or Enterprise JavaBeans. The provider is the glue that links the developer's code to the actual implementation.

WSIL complements the discovery features of UDDI, which provides a large-scale service directory. Like other 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 understand which local services are available. The Inspection

standard enables site or server queries to retrieve a list of services. This list includes pointers to either WSDL documents or UDDI service entries. WSIL is a useful lightweight way of accessing service descriptions without accessing a UDDI server.

A FINANCIAL SERVICES INDUSTRY PROJECT

We worked with an organization with five back-end legacy systems running on different hardware and software platforms (see Figure 4). Each application was built piecemeal as a silo, and many had been added through mergers and acquisitions. This redundant set-up was difficult to maintain, and adding functionality often meant it had to be replicated across all systems. Most of the organization's business knowledge was locked within the business rules embedded in these five legacy applications. But the business rules lacked access points, and could not be invoked from other applications requiring the same functionality (see Figure 5).

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, but the business processes and rules for each

Web Services

Promises and Compromises

system had not been architected to function in unison, or coordinate information flow, other than through nightly batch processes.

Using the CBDi approach, the back-end legacy systems were inventoried, data-mined, message-enabled, and componentized as shown in Figure 6. This was done to prepare for interaction with J2EE-style programs running on an application server that could now access the chunks of business functionality and rules previously locked within the legacy systems. This facilitated a unified user-experience, as if one back end system handled everything. Enterprise components shown in Figures 2 and 3 were built to encapsulate the functionality of the middle-tier business logic of back-end legacy systems.

TECHNICAL CHALLENGES

The current generation of web service infrastructures and tools has the typical problems of early software. Both XML tagging and text representation cause a data size explosion compared with binary representations. XML data

must be parsed when it is read into an application, to create the internal data representations. Further complicating performance is the need to read in and parse the tag definition set. Encryption and de-encryption also increase overhead. These performance issues will be addressed as the technology matures, but developers today can expect factors of 10 to 100 slowdown compared to conventional distributed computing operations.

Challenges will mushroom as web services grow beyond the new distributed computing infrastructure of SOAP, WSDL, and UDDI, to become electronic utilities (eUtilities) delivered to end users over the Internet. The SLAs of eUtilities will include functionality, availability, scalability, performance, resources, and reliability. Terms of use may vary by customer, and may change over time, necessitating dynamic monitoring of customer needs and resources. SLAs describe not just a service's functional interface, but also legal consequences of non-compliance, and levels of support and access. To satisfy SLAs, developers must create software that meets constraints

beyond simple functional correctness. Web service applications must include both conventional APIs and integrated interfaces for metering, performance, manageability, auditability, replication, and reliability.

To further increase complexity, Web services must be dynamically configurable. Just as the telephone company depends on an electric utility, so future Web service applications will depend on other eUtilities. Since such services may change in some way that affects the eUtility, it must detect and respond to changes, or at least degrade gracefully. This on-demand characteristic makes the use of context-aware component with formally defined manners all the more essential [2].

OBJECT ORIENTED WEB SERVICES

Object-oriented (OO) technology such as Multi-dimensional Separation of Concerns (MDSOC) (6, 7) will help define web services. MDSOC and the allied discipline of Aspect-Oriented Software Development (AOSD) (8), notably, Hyper/J™ and AspectJ™ give us methodologies for using self-contained entities with clean definitions. From the point of view of web services, however, objects are not created equal. Avoid thinking of objects in the classic textbook sense. To keep overhead low, and to work within the constraints of web service design methodology, web services need larger-grained objects that represent business functions such as customer accounts or purchasing services, or large IT components such as security or authentication services.

MDSOC helps create “crosscutting” concerns that affect many modules throughout an OO system. Modularizing system aspects that do not conveniently fit the dominant hierarchy minimizes duplication and implementation errors and isolates changes during maintenance. Various aspects are then woven into the base code

Web services (eUtilities), SLAs, and interaction among functional, management, and infrastructure interfaces.

to form a new program. A well-designed MDSOC system eliminates the distinction between “base code” and “aspect code.” MDSOC’s ability to permit the separation of interacting concerns, such as a monitoring service from one supplier and a multileveled-performance service from another, will be essential to web service development.

Since web service customers negotiate for specifics they will obtain from the service, a web service must define one or more functional APIs, as well as present non-functional interfaces that permit control over performance, reliability, metering, and level of service. These interfaces must interact with one another in fundamental ways, as suggested by Figure 7. For example, obtaining high performance may necessitate the use of different communication protocols, search algorithms, and concurrency control models. The definition and support for such deeply interacting interfaces presents a significant challenge to service engineering and deployment.

An object-oriented Web service (OOWS) is a software component built on a highly reliable infrastructure perhaps including other OOWS, which provides APIs. Unlike other components, OOWSs must be dynamically controllable by SLAs, so that they can change during build-time, integration-time, or runtime in response to SLA changes. These components must also provide non-functional

Web Services

Promises and Compromises

management interfaces either as direct programming interfaces (similar to APIs) or through configurable profiles, using a re-configurable architectural style via manners and enterprise component configuration profiles (13). These non-functional interfaces or profiles permit control over attributes such as performance, scalability, level of service, and potentially, OOWS capabilities.

As Figure 8 illustrates, capabilities embodied in the non-functional management interfaces of web services may crosscut functional API capabilities, making them good targets for MDSOC technologies. Also, some functional and management capabilities may require their own class and interface hierarchy to implement the necessary domain model. These hierarchies must be integrated as the capabilities are combined into a particular OOWS. The same issue arises when OOWSs are combined to build a new service. This is likely to be a critical problem for OOWS developers, who lack control over the domain models used in underlying eUtilities.

While MDSOC technologies are promising, they do not currently address several critical issues in the interaction of functional and management interfaces. One is the problem of semantic mismatches, which occur when two or more capabilities have mutually incompatible assumptions or requirements. For example, a client may negotiate for a service level 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 identify semantic mismatches before they bite a client. One must know 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. This must be determined before the capabilities are promised to the client.

Interference is a typical semantic mismatch problem in concurrent software, but potential exists for interference to occur in OOWSs between the management and functionality interfaces, when software interactions

produce undesirable results. Interference can occur when messaging and transactions are present together. To allow a transaction manager to preserve atomicity and serializability, messages should usually not be sent until a transaction commits. A messaging capability could easily send messages independently, thus inhibiting the transaction manager from satisfying its requirements.

As this discussion suggests, it is not always possible to determine what a piece of composed soft-

ware will look like. Unlike non-compositional paradigms, where one does not generally get behaviors without programming them, developers using compositional paradigms experience unpredictable software behavior. This is partly because compositors add logic, but it is also because compositors break existing encapsulations to integrate concerns. The developers of those encapsulations made certain assumptions about the module's behavior, 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 be unacceptable in an OOWS context. Here are five additional tips for working with OOWSs:

Verify conformance to SLAs. Ensuring SLA confor-

Web services **challenge** software engineering **models** that have dominated application development for **traditional clients** and server platforms.

mance requires 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, since an OOWS may not provide certain functionality or performance for users requesting lower levels of service.

Find ways to build 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, with one service depending on another, each with their own SLAs. Ideally, the service designer should spend more effort integrating services than constructing new ones, but the services may not be under his or her direct control, since someone else may own them. In such cases, designers or builders can’t use traditional software engineering methodologies to developing composite services, since these methodologies depend on the centralized control and static deployment assumptions.

Limit the impact of change. An OOWS may change without notice, potentially affecting services that depend upon it. Thus, services must come with integration and runtime support systems that identify service change and respond to it via actions like real-time version control, configuration management, “hot swapping” bits and pieces of services, and upgrading and degrading gracefully.

Again, changes affect both the functional and management aspects of a service, and such changes have legal implications. 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 choose a set of modules to put together. At runtime, it must potentially replace much smaller-grained pieces, to ensure the lowest impact of change on the modified service. Advanced configuration management approaches, like⁹, may be of use here.

Treat SLAs as software. SLAs themselves must be considered a key component in service engineering, as they both define and configure the functional and non-functional aspects of a service. They are, therefore, a service specification, and they must be satisfied both statically and dynamically. 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 link directly with the OOWS capacity planning, design, and architecture, the service monitoring/execution/control, and the compliance checking and reporting to the provider and end-user. Given the competitive nature of the first

Web Services

Promises and Compromises

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

Enterprise architectures that capitalize on web service capabilities are evolving rapidly to assimilate assets into a dynamic structure of services on demand. New technologies and methods are maturing to achieve acceptable service level characteristics. One of the best ways to implement 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 within the organization rather than exposing them externally. As you gain project experience and uncover best practices, get ready to migrate to a full service-oriented architecture that externalizes useful business services. Q

REFERENCES

1. Arsanjani, A. Enterprise Component Services, Communications of the ACM Oct 2002.
2. Arsanjani, A., Alpigini, J. Using grammar-oriented object design to seamlessly map business models to software architectures. In Proc. 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. In Proc. 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. In Proc. Technology of Object-oriented Languages and Systems 39, 2001.
6. Tarr, P., Ossher, H., Harrison, W., and Sutton, S.M. N degrees of separation: multi-dimensional separation of concerns. In Proc. ICSE 21, May 1999.
7. Ossher, H. and Tarr, P. 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. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M. Palm, J., and Griswold, W. An overview of AspectJ. In Proc. ECOOP 2001, June 2001.
9. Chu-Carroll, M.C. and Sprenkle, S. Coven: brewing better collaboration through software configuration management. Proc. 8th International Symposium on Foundations of Software Engineering, 2000.
10. www.w3.org/TR/SOAP.
11. www.w3.org/TR/wsdl.
12. www.uddi.org/about.html.
13. Web Services Invocation Framework, www-06.ibm.com/developerworks/library/ws-wsif.html.

ALI ARSANJANI is a Senior Architect with IBM Global Services and leads the Component Competency, specializing in the implementation of component-based and service-oriented architectures.

BRENT HAILPERN has worked on and managed many projects relating to issues of concurrency and programming languages. Since 1999, he has been the Associate Director of Computer Science for IBM Research.

JOANNE MARTIN is currently Solution Development Executive in IBM Global Services, specializing in the transformation of legacy application systems through Web services.

PERI TARR is a Research Staff Member at the IBM Thomas J. Watson Research Center and a pioneer in the field of aspect-oriented software development (AOSD), in which she co-invented multi-dimensional separation of concerns and the hyperspaces approach to AOSD, and has been exploring issues in multi-dimensional separation of concerns throughout the software lifecycle.

BIO