Deloitte.

Web Services 2.0
Policy-driven Service Oriented Architectures

Thomas B Winans and John Seely Brown

Web service-oriented architectures do not just happen. Their development is intentional, and can be accomplished by transforming existing architectures over time, or as greenfield efforts.

This Working Paper discusses architectural principles and guidelines for developing service oriented architectures from an outside-in point of view to identify characteristics of a service architecture that will deliver thepotential of global web service-based platforms for tomorrow's enterprise.

Introduction

Web Services entered the technology scene somewhere between 1998 and 2000, not long after a subset of SGML called XML did the same. Perception of Web Services began as “the next generation of RPC” to, now, “the basis of standardizing interoperability between heterogeneous application platforms, and on which tenets of modern web architectures rest.”

Talk with technologists today about machine to machine software interoperability invariably becomes a conversation about Web Services, SOAP, and WSDL, and includes a sprinkling of WS-Security, WS-Reliable Messaging, Semantic Web and Web 2.0 for good measure. Conversations with software vendors are similar: they apply the term service oriented architecture to their product platforms to imply technology freshness, future proofing, and ease of standards-based interoperability with prospective client and 3rd party application systems.

If Web Services are so revolutionary, why do we not see greater success with their use?

However, when we consider how highly Web Services are touted as the basis of next generation integration and application architectures, we wonder why we don’t see more successful and widespread implementations of service-oriented architectures that enable us to do more than the application architectures that are hosted in enterprise contexts today, with which we’ve become quite familiar.

We would expect to see easier enterprise application interoperability (not just enterprise application integration), increased reuse of business functionality, greater numbers of distributed applications, a more usable web service registry than UDDI has unfortunately proven to be, and hosted software exposing software as a service that becomes governable, reliable, and robust such that it can be embedded in enterprise applications. While we know that significant investments in Web Services and service-oriented architectures are being made, but we see neither the kind of revolutionary breakthroughs vis-à-vis Web Services that indicate we’re able to more easily develop enterprise application functionality using them, nor the ability to deploy them more easily than applications provisioned on traditional application architectures.

One reason we may not see success with Web Services is simply that there is unwillingness to grapple with the challenges of platform modernization ... a problem with which all sizable IT shops must deal at some point, whether they wish to or not.

We believe one reason that we don’t see the success we’d expect is that service orientation more commonly than not is viewed as an add-on to existing architectures rather than a fundamental strategy on which an architecture is based … a kind of wrapper put around existing functionality solely to simplify systems integration. While Web Services technology could be used in this way, doing so stops short of the potential of a well-formed service-oriented platform.

A second reason is that even were we to take Web Service-oriented architectures seriously, we have neither a means to find web services (which arguably could be some form of LDAP database that could be used both to meet design time registry and runtime directory needs), nor a means to govern and manage their use.

The cost of not capitalizing on the potential of Web Services is so high that it warrants a closer look at how Web Services should be architecturally viewed. Without having a proper point of view regarding architecture, we will find ourselves using Web Services only as a commoditized enterprise application integration platform. The travesty of such would be trading the nominal gains and optimizations of implementing integrations between enterprise systems for those of implementing loosely coupled services, possibly arranged in service grid ecologies, that provision globally-scoped business process networks.

A second reason we may not see success with Web Services is that their effective use requires capability to govern use both at design and run times. IT shops are not prepared for this, and required and enabling infrastructure, e.g., a web service registry, has not been implemented that addresses both design and runtime directory and registry needs.

In Search of Web Services 2.0

It seems that anyone who is anyone in Enterprise IT is pursuing a plan to make his or her enterprise platform service oriented. Before actually putting forward a definition of what that might mean, it is helpful to describe the experience of one company that successfully undertook that goal.

Rearden Commerce developed what it calls a personal assistant network that over 135,000 merchants have joined because Rearden implements a foundation of services that simplify use of the Internet as a marketing and sales channel. Many times that number of individual and corporate users have subscribed to Rearden to consume its merchant services because Rearden also implements services that enable corporate clients to set, monitor and manage travel and expense policies, designate preferred vendors, and generally enforce corporate purchasing policies. Now Rearden's functionality is provisioned by a service-oriented platform that supports service composition to a user interface mashup level.

But Rearden's architecture was not always a service-oriented one …

Prior to two and a half years ago, Rearden's architecture was essentially like many of the web applications we see today: three tiered, open source web and application server technologies, and a relational database that combined to expose a framework to which merchant clients could interface to Rearden business "services" or functions.

Rearden's management team had the foresight to recognize the company's need to create a platform (not just an application), and the corresponding need to make architecture changes to support more rapid development and simpler deployment of new services. By this time, Rearden already had clients, so it understood that change had to be made transparently to its user base whenever possible, or in a way that the user base viewed as a positive upgrade of capability to which they could migrate as this became expedient to their business.

Development of a service-oriented architecture requires acknowledgment and understanding of the differences between SOAs and the typical enterprise architecture.

Rearden strengthened its leadership team with technologists who had participated in web service infrastructure companies and could guide in Rearden's architecture modernization. This new leadership team undertook a transformation of the company's 3-tiered architecture to a service-oriented one over a two year period. It started by developing business service interfaces. It then began to eliminate duplicated copies of code, componentizing functionality and building it out for reuse. It focused on cleaning up the architectural mistakes of the past by partitioning the platform into functional domains, and factoring out business rules so that both Rearden and client rules could be managed in one place, in a policy engine. After using service interfaces to wrap the current implementations of their functionality, the technology team was positioned to re-implement functioality to make it natively service-oriented - an effort it has now completed. At the end of the two and a half year period, Rearden had transformed its traditional web application architecture to being a service oriented one.

During this process, Rearden had to make a number of key architecture decisions:

- It had to decide whether or not the benefits of simply wrapping its existing application were sufficient to meet future needs. Rearden decided to transform the entire platform to service orientation because of the benefits of reuse, composition, uniformity of structure, modularity, and ease of deployment.
- It determined that policy must be formally managed in a way that at least made it seem that policy was virtually centralized. Conventional architectural wisdom is to layer business rules management between the user interface layer and a business objects or data layer within an application. But Rearden's tens of thousands of merchants and corporate users caused it to realize that even a well-formed business rule layer would not make business rule management simple: policy management had to become a formal architecture component.

- It recognized its platform needed to be time sensitive. Time becomes important when managing policies which are effective from one point in time until another point in time. Without support for time, Rearden could not easily evolve its platform as client policies evolved.

And there are other challenges:

- Business transactions are long lived and do not conform to short lived transaction semantics. An alternative unit of work semantic, called compensation, must replace or at least complement the transaction management techniques used to manage short lived transactions.
- People need to participate in long lived transactions, so the architecture must enable such. Many times people are able to correlate faults in context much faster than software systems that usually can only detect faults from a known set and possibly fix them – in prescribed ways. People also are necessary to maintain policy constraints as business evolution occurs. Business conducted over longer periods of time are impacted by market changes that a software system would never detect as a form of exception.
- The granularity of functional interfaces must become more like business functions that people perform and much less like fine grained application programming interfaces. Fine grained interfaces make interoperability between applications and between business partners difficult because the fine granularity is, at least in part, dictated by technology that provisions their business capabilities. It is far easier to inter operate at business functional levels because businesses largely perform common business tasks (where interoperability often must take place) in similar ways.
- And traditional means of communication, e.g., paper documents that either are faxed or scanned and voice technologies like the phone, still must be used both as input and output communication channels. Use of these technologies affects structure of business processes and underscores the need for humans as first class actors in architectures that scale beyond the enterprise.

Architecture Fundamentals

The challenges noted above actually represent fundamental architecture principles that are critical to successfully implementing service oriented platforms – whether web service oriented or not. So it is important to explore some of them further to establish a point of view on developing next generation service oriented architectures.

We wish to highlight native service orientation, the importance of implementing policy management as a formal architecture component, the impact of a long lived transaction model, the importance of people as formal actors in the architecture, and a method to coordinate transactions that - at least in our view - more effectively incorporates policy.

Native Service Orientation

An architecture that manages the life cycle of a homogeneous set of architecture entities is simpler to manage than one that manages a heterogeneous set. A conundrum that faces all architects developing service oriented platforms is whether or not to make their architecture manage services alone, or if there is room in the architecture for some other kind of entity (e.g., services and objects co-exist, or services and event based middle ware applications co-exist).

Agreement about the way that functionality is provisioned by an architecture is critical to keeping an architecture simple, flexible, extendable, and manageable. Without such an agreement, an architecture becomes complex, standards for development and deployment become collections of special cases, and the architecture devolves to a hodgepodge of modern as well as legacy functions and technologies that might serve the current business model well enough, but cannot accommodate change as a norm. The profundity of this observation is the simplicity that results from constructing an architecture to expose functionality in one way.

We've seen in enterprise application integration (EAI) products how the EAI architecture has been extended to enable functionality to be exposed as services. We note that J2EE platforms have been similarly extended. In hybrid architectures like this, functionality can be exposed as services, objects, or events. The addition of services to object based or event-based architectures adds complexity to architecture use forcing an architect or developer to choose between use of services, objects, or events. The result is that services are used here, objects are used there, and the mix of paradigms becomes confusing. It should be clear that benefits relating to composition, reuse, extensibility, and deployment are somewhat muted by the complexity of hybrid architectures.

With all that said about hybrid architectures, one can point to existence proofs where hybrid architectures have been deployed and function well. But it is also important to note that such hybrid architectures incorporate current-day enterprise notions of short lived transactions and prescriptive work flows that expose fine-grained application programming interfaces, and these drag along with them exception management and human mediation strategies that do not scale to meet the requirements of long lived transactions, and more straightforward interoperability that aligns with business functions. Further still, applications wrapped by a web service wrapper do not meet requirements for time sensitivity, easier policy management, and so forth.

Implementing a native service oriented architecture, an architecture where services form the core functional primitives of the architecture, naturally affords the benefits of a homogenous architecture. Note that the term native could be taken to mean services and only services are used to provision functionality in an architecture - clearly this is the ideal case. But this may not be possible in reality. Are the benefits of service oriented architecture still possible? Certainly! We have proof points in operating systems today in the form of abstraction layers for device and systems management that cause devices and systems to look the same to the operating system, despite the fact that they might be radically different in reality Unfortunately, architects of enterprise platforms and their sponsors too often under invest in the implementation of abstraction layers that could simplify the architecture in so many ways.

Policy Management

Policy can be thought of as a collection of rules to manage an architecture and functionality built on it.

IETF Terminology RFC3198

- "Policy" can be defined from two perspectives:
  - A definite goal, course or method of action to guide and determine present and future decisions
  - Policies as a set of rules to administer, manage, and control access to network resources (RFC3060)
- These two views are not contradictory since individual rules may be defined in support of business goals

Andrea Westerinen, VP Technology, Cisco Systems, IEEE 2003 Policy Conference

Modern application architectures typically have a business object or business rule layer where all rules in a set of integrated applications are placed so that they may be more easily managed. However, as we consider distributing applications across enterprise boundaries by exploiting web services, we quickly see that sources of business rules exist all over the enterprise, making clear the need to manage policy differently so that the policies of multiple enterprises can be managed as easily as the rules of an integrated set of applications. To do so requires factoring of policies out of application business layers and into a policy engine so that it can at least appear to be centrally managed.

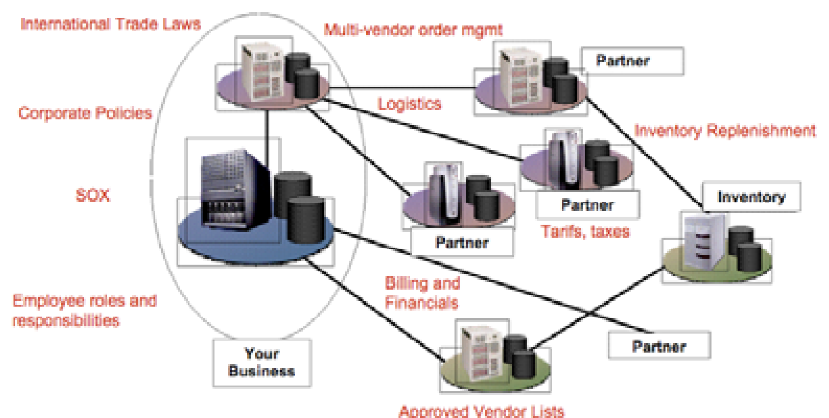
When policy is factored to be more easily managed, then there is opportunity to extend how and where policy is applied. Perhaps it is clear to readers how policy could be applied to the way that business is conducted, or that infrastructure management policies can be enforced by enterprise systems management platforms.

However, the opportunity also exists to relate policy at the business level to managing systems and network infrastructure. For example, one could reason that it should be feasible for service providers to prioritize use of their infrastructure (e.g., bandwidth, ser ver resources, staff members, etc.) in order to ensure ser vice level agreements to which they've contracted are not violated such that they must pay some penalty. In order to do this, infrastructure policy must be harmonized with business policy.

Business Policy Meets Infrastructure Management Policy

Application vendors and IT shops that enable or implement application hosting understand the need to manage critical hosting and data center resources, and the relationship between the ability to manage them and a characterization of application resource requirements. We've seen an early wave of virtualization products that enable some management of constrained or limited resources in hosted environments, but these kinds of products do not begin to formalize the use of policies in a way that make it possible for application systems to specify the resources they will need, together with estimates of how much of these resources will be needed when, so that resource usage policies can be constructed and refined as a function of real-time monitoring. Cassatt Corporation recognizes the need to manage resources using capabilities that exceed what we see today in virtualization.

Cassatt has implemented what it calls Active Response, a data center management platform that treats policy as a first class component, and manages runtime policies using a Policy Engine. Policy is defined in a declarative way and used together with configuration information that defines resources and how they can



Policy influences…

- How customers and service providers interact
- How service providers and suppliers interact
- How work is accomplished, whether manual or automated, whether internal or external
- Where business is conducted, under what constraints, and how it is reported
- How IT systems provision business functionality, and how they interact with partner systems
- How IT systems are operationally managed

be used. Policies can be used to govern when resources in a Cassatt managed utility computing environment can have their power cycled, when more of one set of services should be available to manage peak load time or to compensate for a server that had to be removed from service – taking system dependencies and system redundancy requirements into account.

It would be easy to consider policy in Active Response to be systems management oriented and similar to HP Open View or IBM Tivoli. But systems management is not a stopping point, and Cassatt takes both systems management and virtualization - for that matter - to an entirely new level by enabling dynamic policy-based resource provisioning.

Policy provides one of a number of ways to relate virtual layers of an architecture. Management platforms can correlate events that have business meaning just as easily as they correlate system level events, and these correlations can bind to SLAs which must be monitored and enforced. Enforcement of a business SLA might lead to resource dedication for a certain time (maybe immediately), and even priority and policy-based preemption to commandeer resources for a specific task until it is completed. In such a policy driven process, policy could: optimize energy utilization in a data center; determine services which need to run to meet current or very near-term projected resource demands; and monitor functions that inform operations staff of SLA violations averted when Active Response put a spare service node into production in response to a greater-than-expected seasonal peak load against critical business services.

Levels of Policy

- Goals –> Rules –> Device Commands
- Considering constraints (What can't be allowed or can't be done)
- Purpose to allocate and guide the operation of computing and networking resources
- Manages and is driven by business and mission-critical data and operations

Andrea Westerinen, VP Technology, Cisco Systems, IEEE 2003 Policy Conference

Policy Meets Business Context

Factoring policy from the many component parts of an architecture provides opportunity to make policy enforcement context sensitive. Consider an application that manages medical records in a hospital, and that secures access to services as a function of a medical professional's role in the hospital and in relationship to a particular patient. Policy determines whether or not a professional can perform a service at all, but also whether or not a service may be performed relative to a particular patient. Can this professional see a particular patient's records or not? If the patient has HIV, how is that information communicated to a professional allowed to see some but not all records in a patient's chart? When searching for information in response to a legal request vs. an attending physician's request, how much of the hospital's information should be considered valid for search?

These questions identify points of context sensitivity that can be captured in a policy definition, making policy far more dynamic in nature than one might otherwise think.

Confluence of Policy

Policy can cut across all architecture layers – whether business or infrastructure related. One could logically partition policy into policy spaces, each relating to one area of an architecture. But how discrete the policy spaces are depends upon the richness and extensibility of the way that policy is defined in a platform. Hopefully the way to define policy makes it feel like there is one policy space, and whether policy is factored into separate spaces or not becomes immaterial.

In a service oriented architecture, policy spaces come together in a runtime policy pipeline that exposes policy

through pre, post, and invariant extension points that provide opportunity to harmonize policy spaces when necessary, and judge which policy should govern when policy spaces cannot be harmonized (a human could even be consulted to make such a determination at the last minute). Policy configurations exposed as extension points can be used to publish events monitored by a Policy Management system and, in turn, indicate to system monitoring and business components that subscribe to those events that resource use may be preempted, critical events have occurred, and so forth.

As an example of how business and infrastructure management can be brought together, consider Rearden on a Cassatt technology stack. Their combination can illustrate a way to manage both infrastructure and business policies using a common policy engine, enabling harmonization of policies across infrastructure to business services layers of a modern architecture to meet client and merchant business service level agreement objectives and, at the same time, manage operational costs using a utility cost model in which system, network, and application resource consumption can become more predictable.

And perhaps just as important as managing policies across manageable architecture resources is the capability Cassatt provides to identify what can/cannot be managed, and what a reasonable service level agreement could be in a Cassatt-supported service environment.

For example: Cassatt's management layer could be used to monitor round trip performance of services (e.g., 3rd party external services) not directly under Cassatt's control, ensure their access control policies are enforced, and that request/response times fall within expected time windows or appropriate alert notifications are raised/published so that corrective actions could be taken if possible.

We have gotten into the habit of constructing software systems that attempt to automate everything, and this is inappropriate. Long running transactions cannot factor humans out of the mix. And humans detect shifts in the business climate - which can be treated as a kind of business exception - far sooner (usually) than a software system.

Another example: Seasonal peak loads are difficult to predict. While research and modeling could be done to characterize normal vs. seasonal system loads, there is nothing like use of real loads to adaptively tweak policies where these can be tied to resource consumption. In so doing, Cassatt's Active Response could be used to both alert of SLAs that might be in jeopardy, and to guide users when creating new SLAs.

It could be argued that systems management platforms have offered similar management capability for some time now, but we believe that the recent acceptance of:

- service oriented architecture implementations,
- metadata-related innovations of modern service-oriented architecture components,
- recognition of the importance of factoring policy out of applications and services to enable global enterprises to more easily collaborate, and
- recent innovations like those of Cassatt that provide benefits - beyond the basic ones of virtualization – for management of utility-based computing environments, all combine to form a disruptive momentum toward policy-driven computing, provisioned by a utility-based computing technology stack.

The Value of Time

Enterprise application systems today seldom tell time: the only time they know is now. Temporal details of what goes on in an enterprise system often are managed by setting up a data warehouse in which details of system transactions are summarized for reporting purposes. But summarizing history limits the way that past decisions made in a system can be reproduced both for audit purposes. And the capability to reproduce decisions as they would have been made at a particular time is impossible with time sensitivity, and the ability to version artifacts like business rules or entire policies. It is crucial in business interactions to be able to prove that decisions made by the system in the past on behalf of interaction participants conformed to policies in force when the business interactions took place.

This translates into requirements on future systems to define and manage a kind of audit-able version of information they manipulate. This information must be managed like legal documents through their life cycle, versioned on update, associated in a database sense

with policies that govern their production, and these policies must be versioned and persisted, possibly in a way that enables multiple versions of a single policy to simultaneously exist within a single runtime.

Long Lived Transactions

A transaction is a collection of activities, sometimes called a unit of work that must be executed as a unit or not at all. In modern software architectures, technologies like databases and queues are able to manage transactions local to them. For a transaction to be coordinated across multiple resources (e.g., 2 databases), a transaction protocol like XA must be supported by all resources involved in the transaction so that it can be used by an external transaction coordinator to synchronize activities and either committing or rolling back the effects of their execution.

The duration of local or XA transactions typically is as short a time as possible since, while a transaction is in progress, critical system resources are locked and cannot be used to accomplish other work. While most transactions provisioned by software systems today are not XA because the transaction scope is managed within, say, a single database, the transaction semantics can be thought of as a simpler case of what is defined using XA, and duration considerations are similar.

But as we begin to deal with globally scalable process networks, we find that this kind of coordination of activities across critical resources is ill-suited for coordinating the kind of coarse-grained activities of a next generation service-oriented system since such activities could live on for months at a time. Hence the need for additional unit of work semantics. We use XA properties shown below, abbreviated by ACID, as a discussion template:

- (Atomic) Activities with side effects either succeed or fail together. When failure occurs, effects of all activities should be undone, and the state of the execution environment should be rolled back to its previous state;
- (Consistent) Unit of work activities transition the business from one consistent state to another;
- (Isolated) Resource changes effected by unit of work activities are not shared until the unit of work completes; and
- (Durable) Once a unit of work completes, its effects are guaranteed despite any business infrastructure failures.

It is desirable that long-lived units of work also have ACID properties – though it is not possible to make it so without relaxing the definitions of these properties, so the notion of compensation is introduced to represent the kind of undo activities that must be performed to manage exceptions in long-lived units of work as follows:

- (Atomic) Failure in a long-lived unit of work must be dealt with in a compensational manner. Compensation refers to a set of activities that either reverses the effects of essential interaction activities performed up to a point of failure and causes the interaction to halt, or corrects the problem that triggered the exception and causes execution of the unit of work to continue.
- (Consistent) Each unit of work must be expressed in the form of goals it is to achieve, together with participant contracts that enable participant coordination. The long-lived nature of the unit of work underscores the importance of capturing unit of work state and even its production goals so that, if necessary, a human being can understand what has transpired up to and including failure, and to determine how best to fix the problem causing the exception condition in a way that leaves the system in a stable state.
- (Isolated) Unit of work content and state must be managed so that it is not inappropriately shared before the outermost unit of work (nesting must be permitted) completes. There is potentially a need to declare within the definition of a unit of work when certain information can be shared.
- (Durable) Durability of a unit of work means that changes made to the execution environment by a unit of work that successfully completes must be guaranteed despite any business infrastructure failures. Aside from business-related changes, this includes state changes of the unit of work itself. Durability also means that completed unit of work results must be reliably communicated to all business participants who wish to have them, for whatever reason. If these results cannot be reliably communicated, then compensatory activities must be triggered to roll the unit of work back. Infrastructure that oversees the unit of work cannot be

obligated to guarantee that participants successfully process communicated results since results might not be processed on a timely enough basis (e.g. they could be processed on a batch basis).

The concept of compensation is not foreign in the Web Services world. However, we note that its conceptualization as a protocol like XA is nowhere near mature, and acceptance of Web Service standards moving along a 'unit of work' management trajectory is not great. This does not mean there is no way to implement compensation, only that a standard way to do so has not surfaced and become widely adopted.

People as First Class Actors

People in a next generation service-oriented architecture play at least the following roles: exception handler which must compensate for errors in a system in which long lived transactions occur; and workflow manager/data processor that recognize when changes in business the business climate are not accurately reflected in business policy.

Exception Handler

Exception handling vis-à-vis short-lived transactions usually equates to reverting system state to the what it was prior to some system or application failure, and then displaying some dialog box containing a brief description of the problem.

This won't work in long lived business interactions. Instead, faults must be managed with compensation plans that quite likely will require greater human interaction while special cases for handling the specifics of business interactions are devised and standardized. Short-lived transaction rollback is a considerably simpler approach to managing faults since its concern is to rollback any work performed in a transactional resource so that the resource is left in (presumably) the steady state it was in when the failed transaction began. But, while the principles are similar with respect to undoing the effects of a long lived transaction, what does it mean to "undeliver" product once a truck has left a warehouse? It might mean making inventory adjustments such that the truck that left the warehouse moves product to another warehouse. Additionally, it might mean ordering more product to restock the warehouse from which the product was removed given a known increase in demand. It may be possible to define, in advance, what must be done to compensate for exceptions in long lived business interactions or units of work. But even these compensation plans may age such that it behooves a business to make it possible to engage humans in the compensation process.

Further, exception handling must include the notions of business and market change to current business capabilities as well as fault management. Changes of conditions in the market place, while not specifically a fault in some software system, represent a change to policy that may not be expected in a system sense. The value of separating and formally managing policy has been discussed above. But we note that humans usually detect such market changes, and ultimately must manage related policy exceptions, so it is critical that humans should be formally incorporated as exception handling actors in a next generation architecture.

Workflow Manager/Data Processor

When systems support long lived transactions, it must be possible to keep policies versioned and current to reflect changes in the business environment that they, in some sense, functionally provision. Usually the system actor that first recognizes change is a person, who subsequently must be able to update business policies to reflect a new reality, etc. So embedding a human into an otherwise automated business process requires rethinking of how workflows/process flows implemented as composite web services should be implemented.

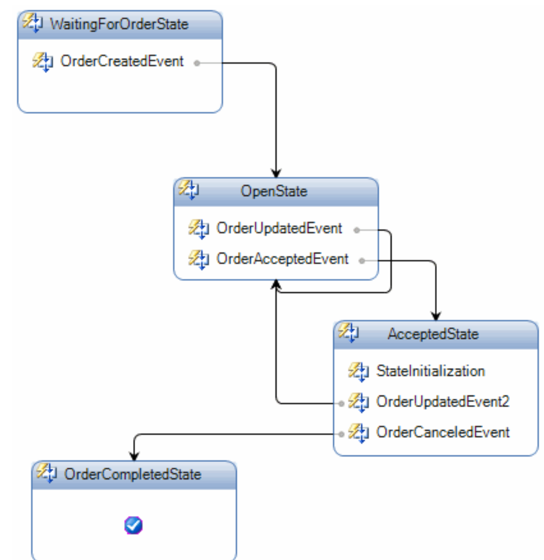
Workflow and other business process management technologies are now well-known within today's enterprise as a means to coordinate sequential activities to perform some business function. The software used to manage workflows, sometimes called a workflow engine, manages directed graphs of activities and coordinates people around performance of these sequenced activities in a way that has enabled software application vendors to make workflow in their respective applications very configurable and flexible.

However, as process networks are deployed beyond current day enterprise boundaries, sequential workflow proves insufficient to flexibly manage coordinated activities for at least the following reasons:

- Sequential vs. State Machine/Policy-driven Workflow: As previously noted, the granularity of functional interfaces must become more coarsely grained and correlate better to the tasks that people perform when conducting business. When such interfaces do become coarse-grained, it is common to see that these interfaces begin to manipulate documents as sets of information that correlate directly with the way that people would exchange information were there no intermediary software system. We note that documents that can be managed by some software system usually have a well defined schema to which constraints – yes, here is a direct link to policy – can be associated. And constraints can be used to orchestrate workflow with a state machine that transitions on the state change of a document – regardless of what mechanism (technology or human) was used to cause the change.
- "Legacy" media types: Information to be processed in business interactions may not be in a digitally parse-able form: content may be in the form of voice files, scanned images, PDF files, or JPEG or TIFF images that contain data most easily processed by a human – whether that human extracts relevant content for future system processing, or that the human directly processes content embedded in such media. Sequential workflows are not terribly useful in this case.

The opportunity to transition away from prescriptively sequential workflows results in a loose coupling between process networks and systems that participate in provisioning the functionality they coordinate. The value of state driven flow is that it can be more explicitly policy driven, and it is better suited to human interaction and mediation.

Another consequence of rethinking how interactions are coordinated and making them policy based is that this process forces us as business process architects to define key events in business interactions that are meaningful to people as well as systems. As state changes occur in long running business interactions, workflows must be instrumented to enable business partners to monitor and actively participate in business interactions beyond basic invocation of platform functionality. This visibility into the life cycle of business interactions enables business partners to tune and refine their business and system processes, both of which are fundamental to evolving their business models to be increasingly global.



Flexible Process Network Definition

One other benefit to state machine oriented and policy driven workflow is the ease with which new roles and role players can be introduced into active process networks.

TradeCard is a New York-based company that provides supply chain management solutions to business partners distributed in ~40 countries world-wide, which we refer to as TradeCard Network Members, or TNMs, for the sake of the subsequent discussion. With its 3500+ client implementations (almost double the number published in July 2007), TradeCard's goal is to optimize business interactions between TNMs by synchronizing them with physical events that occur between issuing a purchase order and delivering a product/service. In turn, this provides TNMs with good visibility to the status of work being conducted in a TradeCard-enabled context and enables efficient and timely triggering of payment and chargeback events.

TradeCard's service offering is a combination of technology and people who operate on the boundary between technology-enabled companies and the companies that are not so technology-enabled with whom they partner. The need to download TradeCard software is avoided by making business functionality available through commonly available Internet browser technology. TradeCard also exposes a secure Internet-accessible messaging API that enables its clients to access TradeCard services using their own applications and infrastructure so long as Internet connectivity is available. And, where infrastructure technology is unavailable, TradeCard helps to make it available by provisioning workstations to suppliers, making its staff members available to assist in getting information into the TradeCard system and ensuring business rule compliance, or a combination of both. TradeCard's architecture, like Cassatt's and Rearden's described elsewhere in this document, treats policy as a first class citizen. Policy is composed of international trade-related policies, policy that TradeCard calls rules of engagement which determines how TNMs interact in a TradeCard business context, and local TNM policies. These policies are harmonized, though TradeCard itself is responsible for doing so as opposed to having some semi- or fully-automated means to do … TradeCard develops policies for its clients and ensures their harmonization with TradeCard and regulatory policies.

Business interactions in a TradeCard environment are defined using roles that specify TNM responsibilities. A role is a named set of business functions that a TNM agrees to perform in a business interaction context, and is analogous to the modern day software concepts of protocols and interfaces. TradeCard's platform implementation permits role players (TNMs) to be linked to a specified role either before or after starting a business interaction.

As noted earlier, TradeCard has an extensive client base with which it has direct relationships, in the sense that client business is conducted using TradeCard's hosted services and staff members (where human services are required). Because TradeCard directly participates with TNMs in their business and has full visibility to the details of all TNM business interactions, and because it is able to late bind role players to roles in a business interaction, it is uniquely positioned to add new value-added services to its service platform. This capability is fundamental to accommodating long lived business interactions in which policies might change, or new TNMs begin participating in already running business interactions. For example: TradeCard became aware of the impact of a change in a buyer's payment strategy relating to a specific supplier where buyer wanted to pay invoices on a 45-day basis, but supplier needed to be paid on a 30-day or better basis. TradeCard was able to broker a private relationship between the supplier and a financing agency such that the financing agency would pay the supplier, charging the supplier a fee for early payment, and TradeCard subsequently would direct the buyer's payment to the financing agency instead of the supplier without making the buyer aware of the redirection. Despite the fact that TradeCard introduced a new (and private) role (played by a human) into an already running workflow, no workflow restart was necessary because of the way TradeCard has implemented its platform.

Forming an Architecture Point of View

To this point, we've discussed various challenges that most certainly will be encountered when pursuing the potential of next generation Web Services-oriented architectures. We now would like to flesh out the need of an architecture point of view that helps us to implement a next generation architecture that satisfies the requirements noted earlier in this paper.

When playing the role of software architect, people look at application and technology systems from various points of view to understand what important functions should exist in a system, how they should be organized and made available for consumption, what user roles are permitted to use which functionality, how system components couple, and so forth. The terms bottom-up or top-down are software industry terms that relate to compositional and decompositional points of view to do exactly this. Architects make certain assumptions about the deployment context of systems when examining system behavior from either of these points of view. And since the vast majority of applications developed today are developed to support the enterprise, many of the assumptions architects make are predicated on current best practice enterprise concepts relating to application system structure, application integration strategy, network topology, the specific realities of the current IT spending model, etc.

But as we consider the ways that business will be conducted in the future, these assumptions are invalidated. For example: it may well be the case that there is no single point of control in a loosely coupled enterprise of the future. Who plays which roles and has what responsibilities most likely will be variable across the set of interactions in which business partners collaborate, and business rules governing how business is conducted using software systems will have to resemble the real world rather than be prescribed by a best of breed or bespoke application – minimally because there could be no standardized set of applications.

Because traditional assumptions are invalid in this future state, it is clear there will be impact to the way that business software functionality is constructed. So it is necessary to form a point of view – maybe many, ultimately – on how to build out service-based platforms around this new set of assumptions and norms. There appears to be at least two possibilities:

- Assume that all enterprises could be made to look like some standard virtual enterprise model to which enterprises wishing to collaborate must integrate - presumably through a Web Services layer, and that service-based applications should be developed for the virtual enterprise; or
- Assume today's enterprise is an invalid foundation on which a future enterprise should be based, that determining a standard definition of an enterprise is an unachievable goal, that the only sensible interoperability between business entities is realized at the business functional level (in the context of a business process network), that the technical means to interoperate is Web Services based, and that interoperability must be policy-driven.

For ease of discussion we name the first of these points of view inside-out and the second outside-in, we characterize them further below, and we argue that the outside-in point of view should be preferred over inside-out when considering an architecture point of view for tomorrow's enterprise architecture.

Inside-Out Architectures

We define inside-out as the point of view that assumes today's enterprise is a reasonable foundation on which tomorrow's enterprise could be based, and we can see the outcome of taking this point of view by considering past efforts in the enterprise application integration (EAI) market place to scale an enterprise application platform beyond traditional enterprise boundaries.

EAI infrastructure has been used by enterprises to integrate their best of breed applications used in front and back offices. Usually some form of Event or Message Broker is put into place to broadcast changes to important/common data entities, and application adapters – that expose fine-grained programming APIs of these applications in event or message forms – serve as publishers of and subscribers to events that coordinate data synchronization.

Applications constructed using EAI technologies start as basic data synchronization applications that recognize when some data entity changes in an application

database and ensures other forms of the same data entity in other application databases are synchronized. These basic applications are composited into more complex applications that filter and correlate multiple events into an aggregated or higher order business event. As more complex event-processing applications are constructed, ordered collections of these applications resemble workflows, process flows, or business services.

XA transactional semantics (n-phase commits/rollbacks) may or may not be supported depending upon underlying messaging transports, and then, if XA transactional support does exist, it usually only applies to message or event delivery (it is up to an event subscriber to process an event and inform the application system if a processing exception must be raised). Actual rollback of the effects of publishing events or messages must be implemented using compensational semantics – i.e., for critical "do" activities there must be compensational "undo" activities.

Anyone who has implemented application integrations within a single enterprise using EAI technologies, especially an enterprise having a large application portfolio, is familiar with the challenges of building them. The list of challenges includes but is not limited to the following:

- The need to designate one application system as a source of record for each important data entity that triggers integration activities, thus defining a kind of canonical data schema.
- Harmonizing unique identifier schemes across application boundaries.
- Capturing the business context in which a data entity changed so that appropriate business rules can be used to determine how the integration application should function.

Extending EAI across today's enterprise boundaries is doable. Many EAI vendors have added web service technology components to their offerings to support doing so. When all members of a partnership deploy the same infrastructure (suggesting there are no significant infrastructure or application impedance mismatches because application APIs and business processes are the same, or at least very similar), this approach to service enablement can work. However, there are limits to this type of architecture, and these should be considered carefully before attempting to take an inside-out point of view where infrastructure, and information, service and process models are not common:

- Data synchronization across enterprise boundaries becomes complicated due to technology choices that partnering enterprises have made. Technology differences often introduce semantic mismatches relating to important data entities that must be reconciled.
- Differences in policies/business rules embedded in best of breed applications and in integration applications must be harmonized. Today's enterprise application systems often embed business policies in them, and make them difficult (at best) to modify/adjust to address policy differences between partners. Integration of applications with embedded policies usually requires custom code or compromises regarding policy because there usually is no (virtually) centralized means to manage policy.
- Making it possible for applications to interface with EAI middleware is sometimes difficult.
- Hybrid application architectures often expose functionality in the form of objects, components, entity beans, XML interfaces, SQL, event handlers, etc. The state and life cycle management of these different functional entities are difficult to harmonize, and it is unusual to see a built-out homogenous abstraction layer around all these different technologies complete enough to make their management transparent.
- There usually is a higher than desirable degree of technology coupling when enterprise applications are integrated in this type of architecture, decreasing maintainability and the potential for reuse.
- People are difficult to integrate into the integrated system since most user interfaces in such an architecture have been developed for the applications that are integrated. This makes human involvement in long lived transactions difficult at best.

Outside-In Architectures

Even with the challenges noted above, it is entirely possible to implement what some might call a service oriented architecture. However, we believe that the assumption that today's enterprise model serves as a

model for distributed enterprises that forms around process networks is fundamentally flawed because a central locus of control is inherent in the model. It is quite probable that businesses in the future will interoperate in a model that is served much better by a model that is fundamentally distributed.

So, as an alternative to the inside-out point of view, we consider an outside-in point of view which:

- Stresses functional decomposition that aligns business services with web services rather than requiring that business services conform to existing business interfaces and capabilities;
- Pushes business services as far down into an architecture as possible (all the way if feasible), forcing architecture simplification and encapsulation of any provisioning technologies beneath a service-based abstraction layer, instead of exposing and pushing upward hardwired business processes, data structures, and technology decisions associated with underlying technologies; Encapsulation of provisioning technologies decreases technology coupling, increases maintainability, increases reuse potential, and simplifies integration and interoperability;
- Factors policy from these services and separately manages it to more easily accommodate changes over time, make it feasible to reconcile business policies to infrastructure management policies, make policy temporal, and version policy.

Applications built on an outside-in architecture are services. Services may be primitive (i.e., their methods may not be decomposable into services), or they may be composite. Ideally, services are stateless, independent and self-contained as possible with respect to the control that they hold over their underlying logic. The functionality of business services is only accessed through its service layer to ensure very loose coupling between services. Even when leveraging legacy application systems to provision services, the goals of loose coupling and self-containment are pursued to enable replacement of service implementations when this is desirable.

We believe that the assumption that today's enterprise model serves as a model for a distributed enterprise that forms around process networks is fundamentally flawed because a central locus of control is inherent in the model.

XA transactionality of technologies that provision web services is entirely encapsulated, since this is viewed as an implementation concern. XA transactional semantics over web services and compensational semantics must be supported in a web service standards-compliant way – and should be limited in their use … transactional semantics arguably can be viewed as exposing the specifics of implementation choices in many cases (we exclude n-phase commit business cases such as would be common in financial trading and banking applications), and such exposure should be viewed as architecturally unacceptable.

Pushing services far down into an architecture stack, as close to technology that provisions the business service as possible, simplifies an architecture often by obviating the need of middleware that has become superfluous over time as commercial application vendors have begun to service enable their products. Business functionality natively designed for use as services is ideal. Services provisioned using 3rd party applications often can be written to leverage vendor APIs to construct web service-enabled adapters that fully encapsulate fine-grained application API calls without the need of legacy middleware products. Requests made of an adapter constructed in this fashion usually are accompanied by information in larger/coarse and complete enough chunks that make it possible for the adapter to minimize out-of-process/over-the-network requests for additional information it requires, so the end result of pushing services down into an architecture stack can be the removal of (now) superfluous middleware components, and less network chattiness.

An outside-in architecture fosters agility in the context of multi-party business interactions. Services in an outside-in architecture are constructed to be self-contained and very loosely coupled vis-à-vis technology and infrastructure dependencies. This results in the ability for participants in multi-party interactions to change roles as long as they implement services corresponding

to the roles they intend to play. While this could be accomplished in an inside-out architecture, an outside-in architecture is less encumbered/constrained by existing roles and business process flows that are built into underlying/existing applications. And the ability to more explicitly and conveniently manage policy enables business rules to be changed rapidly, and in a way that minimizes code-level impact.

Our view is that an outside-in point of view is isomorphic to Service Oriented Architectures that are properly constructed ...

Outside-in architectures sidestep the types of challenges of an inside-out architecture because of the fact that business context is a part of each business interaction:

- Data synchronization is not the aim of an outside-in architecture. However, because information that is shared between interaction participants is complete in the sense that it does not leverage insider IT application information about business entities but, instead, is formed as complete information sets, data synchronization is a nice side effect.
- Policy is not embedded in the technologies used to provision business services, making it simpler to reconcile policies (or determine a compensation plan when reconciliation is not possible).
- Outside-in applications are service-oriented from the start. Whether an service oriented platform is constructed greenfield, or with legacy technologies, taking an outside-in point of view on architecture requires build out of the architecture such that limits of the underlying technologies are entirely encapsulated - or they are replaced.

Pushing services far down into an outside-in architecture stack, as close to technology that provisions the business service as possible, simplifies an architecture often by obviating the need of middleware that has become superfluous over time ... eliminating, in many cases, the need for out-of-process network communication.

Our view is that an outside-in point of view is isomorphic to Web Services implemented to realize the goals of future enterprises. The result of taking an outside-in point of view is that the architecture put into place is service (and only service) oriented, whereas an inside-out architecture is a hybrid architecture which is difficult to manage. Because of this, we also advocate that it may make sense to transition to an outside-in architecture even when architectural goals are not to provision tomorrow's enterprise since doing so can simplify the runtime by eliminating superfluous components as well as simplify application integrations. Architecture simplification is a prerequisite to effectively using it to serve the needs of multiple lines of business even within a single enterprise.

The How Do We Start Conundrum

After reading the preceding section, the reader could ask the question "How do we start?" and criticize us in our discussion of the outside-in point of view for not really providing an answer to that question. Our understanding of technologies, methodologies, and architectures relating to the inside-out point of view allows us to intuit a starting point when taking that point of view, but the reader could suggest that a starting point for an outside-in point of view is not even suggested.

We tackle this head on by suggesting that there are at least two starting points worthy of consideration:

1. Start exactly where you'd start if adopting an inside-out point of view … but don't stop until, like Rearden, you have fully transformed your architecture.
2. Start from scratch!

The Evolutionary Path

Embarking on a program that modernizes and transforms a non-service oriented architecture to one that is could be a multi-year endeavor depending upon the size of an enterprise's application portfolio. And grappling with where to start service enablement requires detailed knowledge of what is in the portfolio – from what the applications functionally do, to how specific application functionality can be linked to a web service technology stack and the cost of doing same. For legacy applications (e.g., some Cobol/CICS application running on an AS/400), unconventional means to expose functionality as services may be required. For example: it may be necessary to implement some IBM MQ-based or some SQL adapter-styled interface to an application running on an AS/400 in order to surface behavior to a service level, or it might be expedient to use some binary to XML transformation (either software or hardware based) to do such. Or, it might be possible to actually compute web service interfaces and implementations by exploiting the metadata rich environments that either already exist or that could be created in today's enterprise IT environment.

The word metadata means data or information about other data. Modern application development and runtime environments contain enough metadata to support the automatic or nearly automatic generation of services, making their implementation faster, easier, of higher quality quickly, and easier to support. And it is feasible to hand construct this metadata where it may not exist in older environments.

Consider the kinds of application infrastructure components commonly found in enterprise application development and runtime environments today, and the metadata associated with them:

- Relational databases contain metadata about the structure of tables, triggers, views, users, stored procedures, and functions that make it possible to map database entities, data types and functionality to their analogs in C#, Java, or other modern programming languages.
- XML contains both content and structure information (especially when accompanied by XML Schema definitions) that make it possible to map content both to relational structures, or objects manipulated in C#, Java, or other modern programming languages. XML also may be mapped to HTML and stylized with style sheets for a standardized rendering that appears to be custom.
- CASE tools that are used to model behavior in application systems contain metadata.
- Runtime environments or modern programming languages like C# and Java are dynamic and can be introspected. This means that class structure can be examined while service applications run, or new classes can be created on the fly and dynamically loaded into the runtime to perform some task, or object methods can be invoked in some generic fashion.
- Code libraries, today called jar, war, ear, assembly, or DLL files, can be introspected to understand what code does (e.g., it invokes SQL database functionality or requires XML/XSLT functionality), and what code does not do (useful in test coverage analysis).
- Source code can be annotated both for document generation purposes, and also runtime attribute management purposes. .NET, in particular, makes it possible to implement custom attributes on classes or methods, and this metadata can be introspected as noted above at design/implementation/test time, or introspected at runtime.
- Application infrastructure like HTTP or J2EE servers, or Notification Event Brokers often are now configured using XML. XML can be easily read to understand

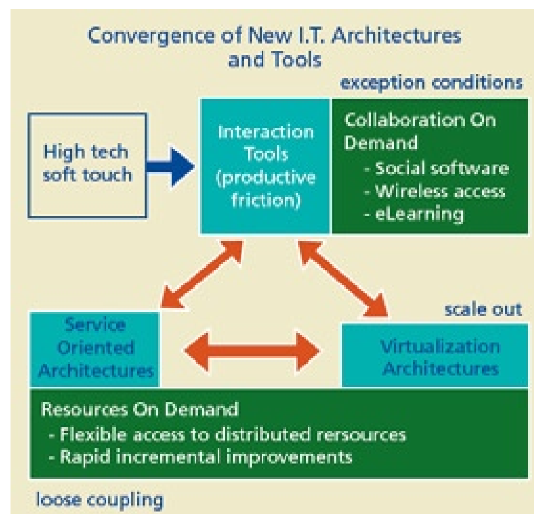
specifics of configurations, and to take advantage of them.

- Web Services are described using WSDL, and runtime messaging is performed using SOAP – both XML. This type of XML could be modified to include important security information, enrich data, or even redirect a request.
- Modern development environments, often called SDEs or IDEs, use ANT, NANT, or MSBUILD makefiles. Microsoft has built out a .NET framework around MSBUILD such that new code build tasks can be programmatically introduced when the presence of certain code artifacts is detected.
- Workflow IDEs often serialize workflow graphs and relating rule bases using XML.

Today's development and deployment environments contain a treasure trove of information about components in an application platform such that it becomes trivial in certain (many) cases to generate web services automatically. For instance: it is straightforward to leverage the metadata about stored procedures (e.g., data types, procedure signatures) in a database to automatically generate simple database client-styled web services that can be deployed in a web service-enabled web application (http) server and transparently wrapped with a common security policy. The same metadata also can be used to automatically generate web service client application code. Code generation as the phrase is used here is template-based, straightforward to implement and maintain, and can be provisioned using 3rd party or bespoke code generation tools.

Modern technology platforms are instrumented in ways that prove useful in constructing new or transforming existing architectures to service oriented ones. Template-based code generation techniques can be used to generate web services from existing data models and middleware all the way up to and including user interfaces.

It is possible to implement template-based code generators to generate Web Services code above database or 3rd party WSDL (or other) functionality, parts of the User Interface, build files, event publishers, fault managers, stubs for event handlers, workflow and transformation (e.g. XSLT) graphs, and configurations



for deployment into simple to sophisticated operational environments. Such techniques make it possible to optimize the use of people, so their skills can be applied to real problems or so that fewer resources are required. They also enable standardized use of infrastructure services and functionality developed for reuse. These work together to optimize code fault resolution, improving time to market with flexible and high quality software solutions - the basis of a sound ROI model.

This approach yields a web service wrapper around existing platform functionality which serves as a layer of encapsulation around applications so that, as determined by an application portfolio management strategy, applications can be modified to be service oriented. Such an encapsulation layer represents the first step in transforming an existing platform into a service oriented one over time and in very controlled steps.

Start From Scratch?

Many CIOs and IT executives hope that the costs and risks of transforming a non-service oriented architecture to a service oriented one can be amortized over time, and who can blame them. Most have probably spent a considerable sum developing the current architecture, so the last thing any IT executive wants to ask for is new budget sufficient to fund still more infrastructure-level activities or require their companies to choose between new functionality or resolved infrastructure issues.

Rearden is a kind of existence proof that demonstrates there are strategies to successfully transform a non-service oriented architecture to one that is service oriented. A rather nice existence proof, too! But taking the second option is not as drastic as it sounds.

We have experienced many changes in the technology world during the last 20 years that have been driven by Moore's Law, and we've seen no signs that Moore's law and other "laws" forcing or enabling significant change won't continue to hold true for the next 20 years:

- We see an increase in broadband capacity so significant and offered at commodity prices that we are able to install no less than T1s at our homes – provisioned by the local cable company – making it possible to run a business on redundant Dell or HP servers racked in a small home closet (a.k.a. data room), together with a modicum of battery backup sufficient to see these systems through relatively infrequent power outages.
- We've seen changes in the economics of disk storage so that companies like Amazon offer distributed storage solutions for the nominal monthly price of US 15¢/gigabyte.[1] These economics, applied to network bandwidth, memory, CPU speed and disk storage lead us to take a cavalier attitude toward software performance and scalability, which we frequently improve simply by adding another Linux box, or running to the electronics store to buy another gigabyte of RAM.
- We've also seen a growing enterprise willingness to outsource IT services that do not represent strategic value or core competency. EDI value-added networks have evolved and become less important, but they still exist. Web application and email hosting is now commonplace, as is managed application hosting. We've seen enterprise application vendors significantly invest to service enable their offerings, and specialize them into business domains as a means to evolve application hosting to the Software as a Service (SaaS) model. Global business process outsourcing companies provide companies with opportunities to outsource business processes permanently, or to outsource them while they invest to develop next generation processes. Pervasive- and utility-styled managed computing capabilities are maturing to the point that establishing a service grid that hosts business service-oriented applications is a reality that is well within reach.

These kinds of radical changes, maturing of application platforms, and acceptance of outsourced managed services suggest that while architecture transformation could be feasible, a good return on investment could also result from starting from scratch (almost starting from scratch … breathe deeply and relax) by building out a service oriented architecture using the services of 3rd party application platform products (some, like SAP, are, themselves, being transformed to being service oriented) and leveraging the innovations of modern operating system, development platforms and the services of external business partners.

TradeCard indicates it sees a growing acceptance by its clients to leverage services external to their own IT and business platforms. Rather than investing in one-off portal-based applications to integrate various business partners together, TradeCard clients are increasingly willing to leverage the now mature service API TradeCard has implemented, and use it as their own API to enable interoperability across their supply chain networks. TradeCard's market penetration in specific industry verticals shows that the ability to reuse workflows within these verticals is great, which underscores the point that starting from scratch does not really mean starting from nothing.

1 Jungle Disk, http://www.jungledisk.com/, Feb 2008

In Summary

While we do not subscribe to the belief that the benefits of a service oriented architecture are hype, neither do we believe that architects and lead technologists have thought properly about the requirements that make implementation of a service oriented architecture successful.

We believe one reason that we don't see the success we'd expect is that service orientation too often is viewed as an add-on to existing architectures … a kind of wrapper put around existing functionality solely to simplify systems integration. An inside-out approach has become a kind of path of least resistance in the pursuit of Web Services promises, but wrapping an existing architecture with a Web Services interface layer does not transform the architecture underneath to a service oriented one.

A second reason is that policy is not considered a strong architecture driver that is critical to provisioning process networks for future enterprises. The architectural implications of making it such challenge the conventional wisdom of architecting what we know today as enterprise software platforms, and demand that we rethink our best practices.

Taking an outside-in point of view requires that we separate concerns from the start. Our application platforms must be distributed from their beginning, rather than become so by attaching some distribution layer to our platform as we grow to require it. We must understand how we have permitted past limits of our particular business organizational models to be built into our architectures and how, now that technology innovations enable us to challenge these limits, we must remove them from our computing platforms to effectively leverage infrastructure innovations like Cassatt's, and build out business application platforms like Rearden's and TradeCard's that respond to change rather than impede ability to manage it. The mythical cost of over-engineering an architecture as it relates to such gains should be questioned since architecture simplification could more than self-fund properly building out an architecture the first time.

Giving proper focus to policy within an architecture - across all architecture layers - makes interoperability at the business layers of an architecture feasible, and this has tremendous organizational implications and is key to forming new process networks in response to market shifts. Further, the ability to drive architectural behavior as a function of policy is prerequisite to distributing software-based business capabilities to a grid of services (e.g., a service grid) or to a utility computing platform, and managing infrastructure in a way that well surpasses the goals that terms like software as a service and cloud computing have come to mean. The ability to dynamically re-provision a system while it is running as a function of policy that directly aligns with the wishes of a business community of practice exposes a set of new opportunities to collaborate in business on scales that current day architectures cannot support.

About us

Tom Winans

John Seely Brown
Center for Edge Innovation

Tom Winans and John Seely Brown

Tom Winans is the principal consultant of Concentrum Inc., a professional software engineering and technology diligence consultancy. His client base includes Warburg Pincus, LLC and the Deloitte LLP Center for Edge Innovation. Tom may be reached through website at http://www.concentrum.com.

John Seely Brown is the independent co-chairman of the Deloitte LLP Center for Edge Innovation where he and his Deloitte colleagues explore what executives can learn from innovation emerging on various forms of edges, including the edges of institutions, markets, geographies and generations. John may be reached through Deloitte LLP, or his web site at http://www.johnseelybrown.com.

A copy of this paper can be found on either of the authors' web sites.

About Deloitte
Deloitte refers to one or more of Deloitte Touche Tohmatsu, a Swiss Verein, and its network of member firms, each of which is a legally separate and independent entity. Please see www.deloitte.com/about for a detailed description of the legal structure of Deloitte Touche Tohmatsu and its member firms. Please see ww.deloitte.com/us/about for a detailed description of the legal structure of Deloitte LLP and its subsidiaries.

Copyright © 2009 Deloitte Development LLC. All rights reserved.
Member of Deliotte Touche Tohmatsu