Using OMG's Model Driven Architecture (MDA) to Integrate Web Services

by Jon Siegel
Vice President, Technology Transfer
Object Management Group

Since the bloom of networked computing during the last half of the 1990s, the IT industry has seen a parade of new middleware platforms come and stay. (Support costs would be lower if they would come and go, but this never happens.) In spite of advocates' claims that each one was so superior to its predecessors that it would take over the entire middleware space, each has only been able to establish itself in the restricted segment where it offered an advantage. As a result, today's enterprise IT department may have to support applications using CORBA (still the only vendor- and operating-system independent, standardized middleware), COM/DCOM/ActiveX, Java/RMI, EJB, XML/SOAP, and other middleware platforms. To these, we are now adding C#/.Net and the main topic of this article, Web Services (WS). (By the way, we don't believe that C#/.Net and Web Services will be the last platforms that your IT department will have to accommodate, but that is a topic for another article.)

To the enterprise supporting this complex infrastructure, there is pain at two levels: First, *building distributed applications is hard.* Any of these middleware platforms is, by itself, complex enough that programmers skilled in its practice are a scarce and valued resource, and building and maintaining distributed applications is time-consuming and costly. Second, *integrating the multiple applications that work together to run a business is even harder.* In today's IT environment, integration of two (or more!) programs built on different middleware may require one of those rare programmers skilled in programming both or, perhaps even more costly, cooperation by two teams whose skills combine to cover the solution space.

In this white paper, we'll show how OMG's Model Driven Architecture (MDA) reduces the pain at both levels: Starting with a UML model, MDA-based tools automate and thereby simplify most of the building of distributed applications on any middleware. Then, because every MDA-based application comprises both a UML model and a matching implementation, developers can import the model into MDA-enabled tools which refer to it as they design, and the tools implement, integration pathways into new applications (or build wrappers for existing ones), even if these new applications and wrappers are based on different middleware platforms and this interoperability requires cross-platform invocations.

We'll start off by examining WS, emphasizing the characteristics that make them different from other infrastructure platforms, and then set up a B2B scenario with an automated WS client placing orders to an automated WS server. With our problem set up, we'll introduce the MDA and describe how the customer company in our scenario would use it to design and build a WS client, integrating it with the back-end legacy applications that request items for purchase. As we work our way through, we'll point out features of the MDA that simplify the different parts of our scenario, and how they do it. So here we go, with WS characteristics first:

Web Services Characteristics

Here are some of the characteristics of WS that distinguish them from other middleware, and make them suitable for some classes of inter-application integration:

- Going beyond many RPC or distributed object or component platforms, the WS architecture defines a comprehensive registry that potential clients can search when they need a service. (The other architectures include a directory or naming service, but these are rudimentary compared to a WS registry entry which includes detailed information about the service and how to invoke it. CORBA's Trader is nearly the equivalent, but hasn't achieved the industry buy-in and support needed to flourish on the Internet.) At the largest scope, registries are public and services are provided by vendors, for profit. WS may also be provided within a company, perhaps from one division to another, or provided by an IT department for use by everyone in a company.
- There is enough information in a registry entry to allow a programmer or, even better, a suitably-programmed client, to assemble and dispatch an invocation of the service, and properly record or deal with the returned result. The entry also identifies the provider, the cost of using the service, and how to make payment.
- For our discussion, these capabilities and the infrastructure that results are even more interesting than are details about the standards that make them possible. These standards (XML to encode invocations and responses; UDDI for the registry; SOAP over the network) are interesting in their own right, but we'll stay at a higher level in this paper.
- Unlike components working together in a server, WS are coarse-grained and self-contained – you search a registry, find a WS that does what you need, invoke it, get your response, pay, and the interaction is over. Client applications are unlikely to build up a single service from many fine-grained WS, although they might use two or three as they execute an extended application function.
- Loosely-coupled asynchronous invocations (assumed by the most common Internet-based scenario, albeit not compulsory) make WS slow, at least compared to synchronous RFP calls. In the extreme, reliable store-and-forward network transmission allows a WS request to succeed (eventually) even if the server or network is down when the invocation is sent. This is fine for a computerized purchase of something you don't need until next week or next month, but won't work for a telecomms giant making 25,000 invocations per second recording call data into a transactional database.

Three Architectural Levels

In the WS world, an enterprise with an established IT department will probably have three distinct levels of application and infrastructure:

- Monolithic legacy applications, providing basic business functionality (accounting, sales/stock/shipping, etc.)
- Object- or component-based applications servicing browser- or computer-equipped sales staff or customers, or possibly engineering or production facilities and staff
- Web Services, composed from functions in the first two categories plus new capabilities added specifically for the WS market

The legacy business applications may be mainframe-based; even if not they are likely transactional and scale well to the needs of the enterprise after a conversion from batch to on-line operation. This conversion probably exposes the application's functionality to the network through a layer of wrapper code using the middleware platform that was in fashion when the conversion was designed (which, of course, may have been out-of-date by the time the conversion went on-line). Internally, these applications are monolithic, not distributed.

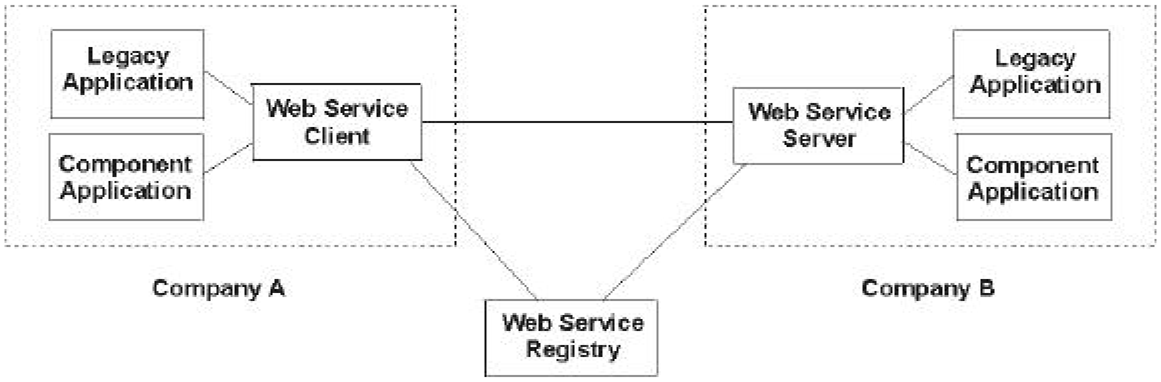
The object- or component-based applications likely run on one or a cluster of CPUs or discrete computers. Handling e-business functions, these applications must scale to high transaction rates without stumbling. Components that make up the internal structure of these applications must be tightly coupled to meet this requirement, precluding shopping for services in a registry via the WS model. Instead, use of established patterns for resource allocation and reference (to components allocated using patterns named session and entity in some component environments) allows references to be cached for instant access. One other difference between these applications and WS: Designed, configured, installed, and invoked by the same end-user company, these applications do not have to deal with the metadata impedance mismatches that arise when independent enterprises try to interoperate and discover that each defines fundamental entities such as "customer" differently enough that software has trouble coping.

WS, unlike applications in these first two categories, are self-contained functions that are invoked, do something useful enough to be paid for, and return the result in an atomic interaction. (For our discussion purposes, we'll regard such "side effects" as the shipping of a purchased product, or streaming of a digital audio or video clip over the network to the customer's computer, as incidental to the computerized interaction, however important they might be to the customer.) The invoking application must shop for the service that it wants, possibly selecting an instance from a list returned by the registry, and retrieve a service identifier in a preliminary step. It must then invoke the service, typically via a loosely-coupled asynchronous transport (which may nevertheless be reliable, if the network implements store-and-forward transmission semantics). WS are, as we already mentioned, unsuitable for rapid multi-use services such as database transactions, or as the internal mechanism linking parts of component-based applications. For the typical enterprise, then, WS are an additional middleware which impose a new set of support responsibilities as they add new capabilities.

As we write this, early adopters of WS are completing their first applications and discovering that primitive WS security standards and implementations do not support secure deployment across the firewall. Instead of waiting for these standards to develop, these early adopters are building WS applications to provide services *within* their enterprises, avoiding the firewall and security problems for now. Still, its registry lookup, asynchronous communications, and verbose message format keep WS from substituting effectively for the more mature and efficient RPC and distributed-object communications protocols within component-based applications, so the effect of this will be only to shrink our three-level application and infrastructure breakdown to the size of an enterprise from its imagined extension to all of the internet.

A B2B WS Scenario

A company may be a user of WS, a provider, or both. Figure 1 shows a WS provided by Company B, being invoked by a client in Company A. Company A's WS client is driven by some combination of legacy and component-based applications. You may have thought of WS as called mainly by GUI-driven clients, but this will surely not be the case: computer-driven clients can invoke so rapidly that their interactions will dominate the network no matter how diligently your children and their friends spend their allowances on WS-enabled shopping sites. Suppose that Company A runs a factory that produces 1,000 automobiles and trucks per day (one about every 90 seconds), and that its WS client orders parts from a supplier, Company B.



At this production rate, Company A is one of the world's smallest automobile producers but this is a complex scenario even so. If an automobile or truck needs only one each of the parts that Company B supplies (such as a dashboard or steering wheel), then we will have to order 30,000 of these each month, and each may need to be adjusted (size, color, style) for an individual vehicle, and delivered in time to be installed as that vehicle reaches the proper point on the assembly line. If each auto or truck needs many of each part (such as bolts or rivets), we may not need to customize them for each vehicle but now we may need millions of each per month, of many dozens or hundreds of sizes and styles. It costs us money if parts are delivered too soon (if we requested delivery a week before production "just to be sure", we would need a warehouse big enough to store 7,000 cars, in pieces), or if they're delivered too late (which would require shutting down the production line until the parts arrived, and then re-starting it).

Back-end workflow applications coordinate ordering of all of the millions of parts and other resources needed for production. In addition to these ordering details, they notify the company's financial applications to schedule payment precisely on the last day that Company B will allow (it doesn't make sense to pay any sooner; money is worth money. But paying late may risk souring a carefully-cultivated business arrangement. Or maybe not.). Payment, of course, goes via electronic funds transfer and is probably ordered by its own WS but we haven't shown it in the figure. Making payment may require borrowing money, perhaps by invoking a WS provided by the company's bank, but we won't extend this example to cover it. We've made our point: The world of business is complex.

On Company B's side of the diagram, receipt of an order via its WS server triggers a multitude of actions on its back end: A main program schedules production and shipping of the ordered parts, presumably some time in the future. Scheduling and workflow applications check that sufficient stock is on hand for the additional production, and order more if necessary (probably by invoking WS on the stock producers' servers, of course). At the same time, Company B's financial applications are notified to schedule AP for the stock, and the sending of invoices for the shipped parts. If there is a gap between the time that Company B must pay for the stock, and when it will receive payment for the parts, then Company B's financial application will attempt to borrow funds to cover the float between payment of the invoice and receipt of the amount due.

Integrating WS into your Architecture

Clearly, both companies in our scenario had to overcome formidable integration problems to implement this scenario, and most of these headaches involved back-end applications on a plethora of (non-WS) middleware platforms. Instead of focusing in on any one of the different integration aspects, let's zoom out to the overall problem: What is the best way to evaluate the overall integration problem, so that we can visualize and design a solution? We would like to use the *same* set of tools for the overall problem that we do for each individual part, even though many parts use different middleware platforms; in addition, we need the toolset to represent and work with all of our individual applications regardless of middleware platform.

OMG's Model Driven Architecture (MDA), adopted as its base architecture by the group in late 2001, can do this and it's an industry standard implemented in multiple tools from many vendors. Here is a high-level description of the MDA; to fill in details, read the white papers and presentations linked on OMG's web page www.omg.org/mda.

OMG's Model Driven Architecture

OMG's MDA unifies and simplifies modeling, design, implementation, and integration of applications – including large and complex ones – by defining software fundamentally at the model level, expressed in OMG's standard Unified Modeling Language (UML). An MDA application's base model specifies every detail of its business functionality and behavior in a technology-neutral way; in MDA terminology this is the application's

Platform-Independent Model (PIM). Working from the PIM, MDA tools follow an OMG-standard mapping to generate an intermediate model tailored to the target middleware implementation platform. (OMG will standardize mappings to all popular middleware platforms; several have already been adopted.) Termed a Platform-Specific Model (PSM), this intermediate product will reflect non-business, computing-related details (typically affecting performance and resource usage) added to the PIM by your architects, and the version produced by the MDA tool will probably require some hand-tuning before it can be used for the next step. (The amount of hand-tuning required will vary depending on the sophistication of the tool, the complexity of the application, and the maturity of the MDA in your application domain.)

Extremely detailed, the PSM contains the same information as a fully-coded application but in the form of a UML model instead of as code. In the final development step, working from the PSM, MDA tools generate interface definitions, application code, makefiles, and configuration files for the PSM's middleware platform.

Because the PIM is middleware-neutral and conversion to the PSM and then to the implementation is mostly automatic, it is practical to produce equivalent implementations of MDA-based applications on multiple target platforms. In addition, tools can generate cross-platform invocations, allowing easy interworking among suites of MDA-based applications wherever they reside. Another benefit of the MDA: because industry standards defined as an MDA PIM are platform-independent, they can be implemented on multiple targets and then used by every enterprise even in industries that haven't converged on a single middleware platform.

Based on UML, automation, and sound architectural principles, the MDA supports applications over their full lifecycle starting with design and moving on to coding, testing, and deployment, through maintenance, and eventually to evolution to a new platform when an application's existing platform becomes obsolete. The MDA became the base architecture for OMG standards in September 2001.

Modeling Integration with the MDA

Represented as PIMs, the set of applications used by a company or industry can be imported together into an MDA-based UML modeling tool and viewed as a group. Zooming out to show the least detail about each application allows us to view and model how they do, or could, work together. One important advantage: Modeling at this high level allows us to focus on business functionality and behavior without worrying about (or even seeing, on our model) technical aspects.

Let's assume that our automobile company had already imported its production and workflow applications, scheduling, sales, and financials as an group into an MDA development tool to model and automate their interactions. Using this view, company architects have analyzed all of the (sometimes labor-intensive and primitive) ways that data produced by one program had been input to another in the "good old days" before reliable high-speed networks became commonplace. Working in the MDA, these architects have designed and built interfaces and wrappers that allow these legacy applications to couple over the network, transforming a batch system into a nimble (all things considered) suite of networked applications that has been running reliably for some time.

Now, with the advent of WS, it's time to interface this suite of applications to the outside world. To the model, company architects can now add an MDA-compliant UML model of a supplier's WS-based stock-ordering application, either provided by the vendor directly or built (mostly automatically albeit with some help from an architect) from the information in the WS registry entry.

Using this comprehensive business-level view, business experts and IT architects can analyze and design the business functionality and behavior of a WS client application that orders parts from the supplier's WS server. Driven by sales and production scheduling, this application would order parts and schedule delivery, notify receiving when to expect them (and to raise a red flag if they don't come in), and notify the accounting application that invoices for the amount due will have to be paid on a certain date. If a reply from the supplier cautions that a part will not be available on time, this application is able to reschedule production of a vehicle on a later date. If multiple companies can supply a part (such as nuts and bolts, or rivets), the application can search the registry for the lowest price or fastest shipping; if a preferred supplier is out of stock, the application could select an alternate. (Once this system covers all of the significant vehicle parts, it could be made available to dealers who could enter "what-if" queries and quote delivery dates to customers for custom-designed vehicles.)

This model must be very detailed – middleware and networked computing technical information will be part of the MDA tool's database, but automobile production details will not! It must also reflect a sound overall design: Handed a good design, an MDA tool will produce an implementation based on best practice that will perform as well as, and probably better than, a hand-coded version. However, faced with a poorly-designed UML model of the same business functionality, the MDA tool will produce a poor implementation if it is able to produce one at all. MDA development *leverages* the skills of architects and designers: IT shops will be able to produce more applications quicker with fewer skilled architects and designers, but will not be able to dispense with them entirely.

Technical details are added to the model only after all business functionality and behavior has been incorporated and reviewed for correctness by business experts. Application architects mark up the PIM, adding hints about resource usage patterns and performance. This is where you input to your MDA-based development tool that one of your back-end legacy applications still runs on DCOM (if this information isn't in the model of that application already, or in the tool's site configuration file), and couple the data in the new WS application to your corporate database, or a selected image of the data. If you're using a component-based environment, you may label component types with resource-usage patterns. (These may have generic names in your model, which will be translated to names like *session, entity,* or *process* if the target platform is Enterprise JavaBeans or CORBA Components.)

Using these hints, the MDA-based tool applies an OMG-standard mapping to the selected target middleware platform to generate a PSM for the application. Because our example application is mainly a WS client, we will probably choose WS as our target platform. Still, our tool will have no trouble generating cross-platform invocations to the multiple other middlewares in our execution environment. Architects examine the PSM produced by the tool and tune it up where necessary, before it is used as the input to the next step.

Once the PSM has been examined and tuned up where necessary, our MDA-enabled development tool uses it to generate interface files, programming language code, configuration files, and all of the other artifacts required to produce a running application. Compiling and linking the files produced by this step yields the deployable application. Designed and coded by MDA-enabled tools, this Web Service client integrates smoothly with the enterprise applications that drive it over RPC and distributed-object protocols, while simultaneously interacting with the company's supplier using Web Services. Faster, better, cheaper – this is the way to build Web Services applications!

MDA architects do not expect that the first version of this or any application will be perfect. The MDA development process assumes that developers will iterate from design to implementation and back several times before settling on a final version. MDA architectural specifications call for automated "round tripping", propagating edits to the running code back into the PSM and then the PIM. You can already buy several tools that reverse-engineer source and even object code into UML models; the automated round-tripping is not that much of a stretch from there. You can expect basic round-tripping capability to be present in some early versions of MDA tools, and for this capability to mature rapidly.

Although our discussion dealt specifically with the client side, this was mainly for concreteness – In our automated-client system, the WS client and server play similar roles in orchestrating and integrating the interaction among all of our enterprise applications. (If we had used a GUI-based client for the example, with no connections to our enterprise back end, the client and server ends would be quite different.) So, we won't discuss the server end separately here – it's already done.

Conclusions

The MDA environment is extremely powerful, and broadly scoped: Per application, the PIM isolates business functionality and behavior, allowing analysis and design of core business aspects undistorted by technical concerns. The PIM then carries through to at least two development pathways: First, from it MDA tools generate one or more PSMs and an applications, on one or more middleware platforms. Second, PIMs for an enterprise's entire application suite can be collected into a model of its total computing environment, allowing their combined business functionality to be viewed, modeled, analyzed, and integrated.

Web Service applications, typically intended to interact across the firewall to an enterprise's customers and suppliers, are inherently multi-platform and broadly integrated. By designing these complex applications in the MDA, and letting MDA-enabled tools build them with help from skilled architectural and programming staff, the computer-savvy enterprise can move into the world of Web Services better, faster, and cheaper than by doing it any other way.

References:

For more information on the MDA, see www.omg.org/mda; for information on OMG see www.omg.org. OMG specifications are adopted at the group's meetings, listed up to a year in advance at www.omg.org/news/schedule/upcoming.htm. To register for a meeting, members may fill out the form at http://www.omg.org/registration/registration-info.htm. Non-members may also attend; see www.omg.org/news/meetings/tc/guest.htm. To check if your company is a member, see http://cgi.omg.org/cgi-bin/membersearch.pl. Address other questions about OMG membership and programs to info@omg.org; send technical questions directly to the author at siegel@omg.org.

About the Author:

Dr. Jon Siegel, OMG's Vice President of Technology Transfer, heads OMG's technology transfer program with the goal of teaching the technical aspects and benefits of the Model Driven Architecture (MDA) based on OMG's modeling specifications UML, the MOF, XMI and CWM. Siegel's scope also includes OMG's industry-standard middleware, the Common Object Request Broker Architecture (CORBA) and the Object Management Architecture (OMA) comprised of the CORBAservices, the CORBAfacilities, and the Domain specifications in vertical markets ranging from healthcare, life sciences, and telecommunications to manufacturing and financial systems. In this capacity, he presents tutorials, seminars, and company briefings around the world, and writes magazine articles and books including the popular "CORBA 3 Fundamentals and Programming" and "Quick CORBA 3". With OMG since 1993, Siegel previously chaired the Domain Technology Committee responsible for OMG specifications in the vertical domains.