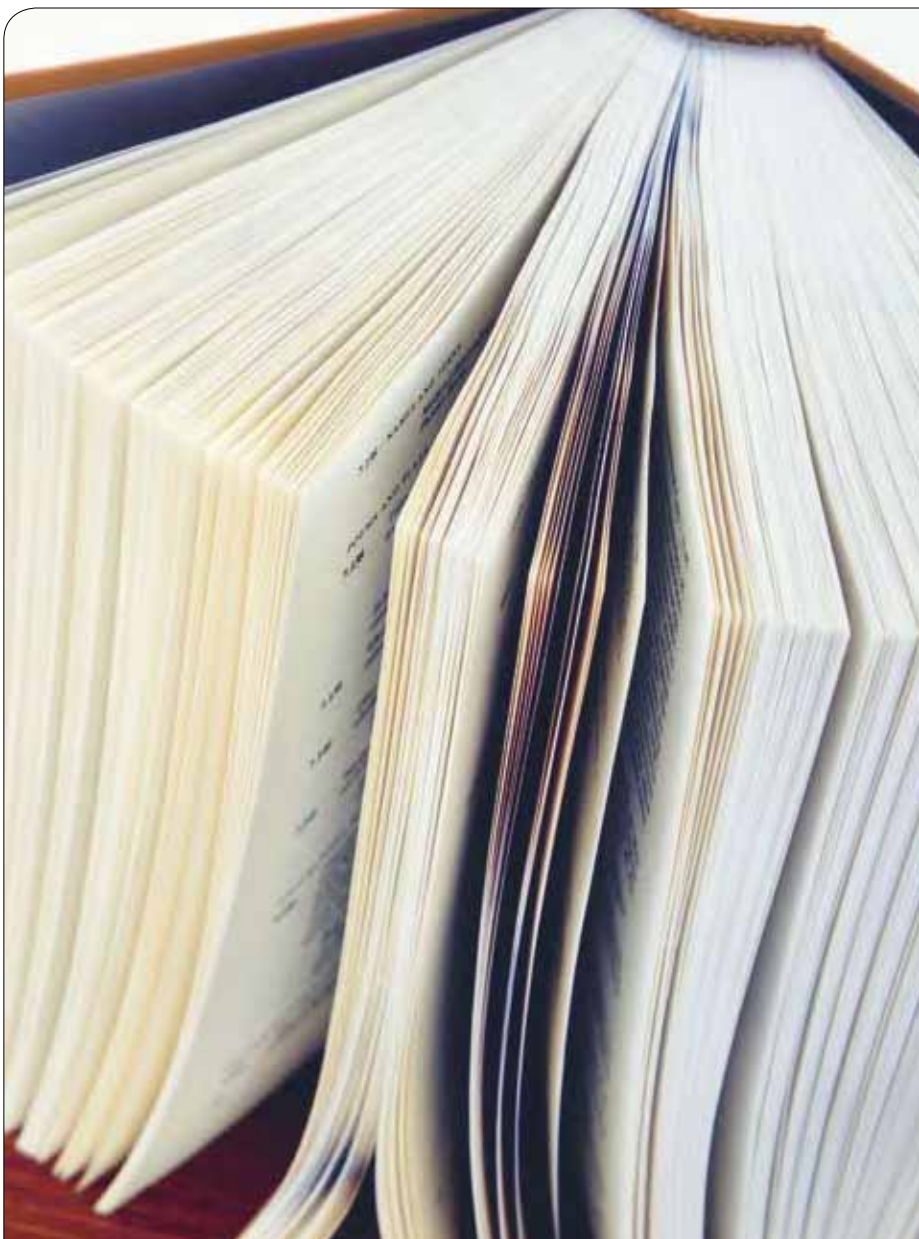
Learn the discipline, pursue the art, and contribute ideas at www.ArchitectureJournal.net
Resources you can build on.

THE ARCHITECTURE JOURNAL™
Input for Better Outcomes
Journal 8

Data by Design



Microsoft®

Contents

THE ARCHITECTURE JOURNAL™
Input for Better Outcomes
Journal 8

Foreword

Dear Architect,

Welcome to Issue 8 of *The Architecture Journal*, the theme of which is Data by Design. As architects, I feel we often undervalue the pervasiveness of data in architecture, especially when we consider how data is used in applications and systems that span multiple geographies, time zones, and organizations.

One analogy I frequently use compares the availability of data within a system to water running through the pipes in a house. When we turn on a faucet, we expect clean, filtered water delivered instantly and usually at a consistent, expected pressure. The pipes are the infrastructure in this analogy, and the water is the data. When I think about data and its relationship to architecture I like to think of the same approach—the data we deliver to users should be clean, filtered, and delivered without delay and as expected—regardless of whether the data is a single e-mail, customer record, or large set of monthly financial data.

Although we will not be covering many plumbing techniques in this issue, we do have a number of great articles from a distinguished group of authors that focus on the importance of data.

We start with Roger Wolter, a solutions architect at Microsoft and author of many whitepapers and books on SQL Server and SQL Server Service Broker (SSB). Roger discusses the importance of reliability for data, especially in the context of designing connected systems.

Tim Ewald and Kimberly Wolk follow with an article that explains some of the models they created for data integration for the MSDN TechNet Publishing System, the next-generation, XML-based system that forms the foundation of MSDN2. Udi Dahan then takes us on a journey for using entity aggregation to get 360-degree views of our data entities and focusing on concrete ways to solve immediate business needs.

Next, another author team, Tom Fuller and Shawn Morgan, share some of their experiences realizing that data replication can be an antipattern for service-oriented architecture (SOA), especially in light of autonomous services and applications. Then Dion Hinchcliffe, the CTO of Sphere of Influence, shares some patterns for data consumption and composition, especially in the areas of mashups and Web 2.0 applications.

Paul Nielsen ends our series of data-related papers with an overview of Nordic, a new object/relational hybrid model that explores greater flexibility and better performance over traditional relational data models.

Rounding out this issue of the Journal, Andrew Filev shares some of his experience combining an agile methodology with a model of outsourcing, and then Arvindra Sehmi and Beat Schwegler return with Part 2 of their Modeling for Connected Systems series. If you missed Part 1, be sure to download Issue 7 of *The Architecture Journal* at www.architecturejournal.net.

Well, that wraps our theme on data. I trust the articles presented in this issue will help you to design systems with the same data availability as water flowing from the faucet in your house—of course, I can't guarantee that you won't get your hands wet!

Simon Guest

Reliability in Connected Systems

by Roger Wolter

Summary

Connected system applications are composed of a number of loosely coupled services often spread over a network; therefore, achieving high levels of reliability and availability for connected systems applications poses a unique set of architectural challenges. For example, if an application stops running because any one of ten services running on ten different servers is unavailable, the failure rate for the application is about ten times the failure rate of the individual services. Data failure makes this issue much more critical because a data source may be used by dozens of services. This article discusses the reliability issues that must be considered in architecting a connected services application and shows how some of the new features in SQL Server 2005 and Microsoft messaging products address these issues.

From hardware, software, and maintenance perspectives reliability is expensive, so it is very important to understand the reliability requirements of your application. While an application with no reliability requirement is rare, implementing an application with more reliability than is required can be a waste of time and resources. For this reason, it is important to understand the reliability issues of connected systems so you can architect a solution that will provide the required level of reliability while not wasting resources by providing reliability that is not required.

In service-oriented architecture (SOA) or connected systems, services communicate with each other through well-defined message formats, which means that the reliability of the connected systems application will be influenced strongly by the reliability of the messaging infrastructure it relies on to communicate between services. We will use the example of services in an automated teller banking application to illustrate the various degrees of messaging reliability and how they are achieved. Message handling between services is generally more complex than client/server messaging because client/server messaging can rely on the user to make some decisions about how to handle various error situations and timeouts, while the server initiating the message exchange in server-to-server messaging must make all the decisions that the end user would normally make in a client/server interaction.

Infrastructure Burden

Many service-oriented applications achieve better throughput by using asynchronous messaging. With *asynchronous messaging* a service sends a message to another service without waiting for a response to the message before continuing. The service processes many more requests because it doesn't waste time waiting for responses to requests it makes to other services, but it puts the burden of ensuring the message is delivered and processed on the messaging infrastructure. The choice of synchronous or asynchronous message handling is usually determined by the business requirements of the application.

For example, if a customer is trying to withdraw money from an automated teller machine (ATM), making an asynchronous request to check the customer's balance and proceeding to dispense the money without waiting for a response from the balance check is generally not a sound business decision. This scenario doesn't necessarily rule out an asynchronous request, however. The ATM might dispatch a balance request as soon as the customer selects the withdrawal option and then let the customer enter an amount while the balance check is proceeding asynchronously. Once the withdrawal amount is entered and the balance is received, the ATM application can make the decision on whether to dispense the cash.

Let's look at how the ATM handles the balance request message. The ATM service will send the balance request message to the account service to get the customer's account balance. At this point, one of three things will happen: the balance will be returned successfully, an error will be returned, or the request will time out because the result was not returned in a timely manner. If the balance is returned, the ATM service continues with the transaction. If an error is returned, the ATM uses its business logic to work with it—maybe using a cached copy of the balance or dispensing or not dispensing the cash depending on the amount requested. The hardest one to work with is the timeout. A timeout may mean the message got lost in transit, the message arrived at the account service but the response was delayed for some reason, or the response was sent and then lost on the way back.

In most cases, the best response is to try again and hope that it will work better next time. If the message was lost on the way the first time, it may get through this time. If the response was slow, the original response might arrive before the second request times out. Unless the account service is unreachable or gone completely, a retry should succeed eventually. Depending on what the problem was, the balance request might have been processed multiple times, or multiple results may be received, but as long as the ATM service is prepared to work with them, these aren't serious issues.

If the message was sent asynchronously, the ATM service might not have the information around to resend the message when a retry is required, so the messaging infrastructure will need to keep a copy of the messages it sends and resend them if required. For this type of simple request message, keeping a copy of the message in memory is probably adequate because if the ATM looses power the customer will have to start over anyway. If the ATM service needs a response even after a power loss, the message will have to be put into some persistent storage, and the messaging system will have to resend it when it restarts. Figure 1 shows three possible outcomes of a balance request.

Message Delivery

For the synchronous version of this request, a simple SOAP Web service will provide the required degree of reliability. Error handling and timeout handling are the responsibility of the ATM service so it will determine the level of reliability for the request. For the asynchronous version of the balance request, either the WS-RM channel in the Windows Communication Foundation (WCF) or express delivery in MSMQ will provide the required reliability if the request doesn't have to survive system failures. If the message has to survive a system restart, MSMQ recoverable delivery is the appropriate choice. These messaging systems will handle the timeouts and retries required to deliver the message and the response so the ATM service doesn't have to include this logic.

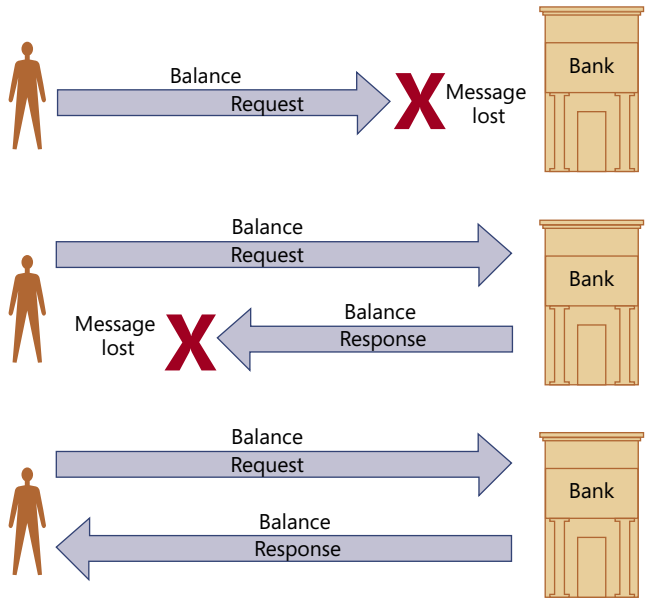
Now that we understand the reliability issues with a balance request, let us move on to the balance change request that happens after the cash is dispensed. The reliability requirements for this message are a lot higher because if it doesn't get delivered successfully, the bank will have given the customer cash without reducing his or her account balance. While the customer is not likely to complain, the bank will not be happy if this situation happens regularly.

The balance change message must get delivered and processed in the face of any number of network and system failures. The memory-based retry mechanism obviously is not adequate for this task because a system failure will lose the copy of the message being kept for retries. Keeping a persisted copy of the message for retries may also be an issue because it is possible for multiple copies of the message to be delivered because of retries. If the message decreases the customer account balance by $200.00, for example, delivering the message multiple times can lead to customer dissatisfaction.

The retries required to ensure reliable message delivery work only if the messages are idempotent. An *idempotent* message can be delivered any number of times without violating the business constraints of the application. Some messages are naturally idempotent. For example, a message that changes a customer's address can be executed multiple times without ill effects. Other messages are not naturally idempotent, so the messaging system must make them idempotent to prevent the damage that can be done when a message is processed more than once (see Figures 2 and 3). This transaction is normally done by keeping a list of messages that have already been processed; if the same message is received more than once it will not be processed more than once.

Most messaging systems don't keep a copy of the incoming messages, but the sender assigns an identifier to each message and the receiver keeps track of the identifiers it has seen before. These identifiers must be kept until the message destination is sure that the source has thrown the message away because if the source fails, the message might be sent again days later when the source restarts. The list

Figure 1 Balance request



of processed messages must also be persistent because the destination service may fail between retries. The destination must also keep track of the message that was sent in response to the incoming message because the source will keep sending the original message until it receives the response and the reason for the retry might be that the original response message was lost.

Choose a Message Infrastructure

Unless you are willing to persist and track the messages in your service logic, you will need a reliable messaging infrastructure to provide this level of reliability. Microsoft has three options for this infrastructure: MSMQ recoverable messaging, SQL Server Service Broker, and BizTalk. Which option you choose depends on your reliability requirements and application design.

Service Broker and BizTalk provide more reliable message storage than MSMQ because they store messages in a SQL Server database while MSMQ stores them in a file. If your application can work with the occasional loss of messages when a file is lost or corrupted, then MSMQ will be adequate for your needs. Some MSMQ applications guard against the potential loss of messages by storing a copy of the message in a database. If you are going to store messages, using Service Broker, which stores messages in the database anyway, is significantly more efficient.

Service Broker and BizTalk provide roughly the same degree of reliability and defense against message loss because they both store messages in the database. Service Broker's message handling is built into the database server logic, so Service Broker talks directly from the SQL Server process to the TCP/IP sockets, which is much more efficient than the BizTalk approach of an external process calling into the database to store messages in a table. While the Service Broker can deliver significantly more messages per second than BizTalk, BizTalk offers a large number of features—message transformation, data-dependent message routing, multiple message transports, orchestration, and so on—that Service Broker doesn't offer.

In general, if reliable transfer of messages between databases is all your application requires, then Service Broker is a better choice because it is more lightweight and efficient at transferring messages than Biz-

Figure 2 Withdraw cash request harmless retry

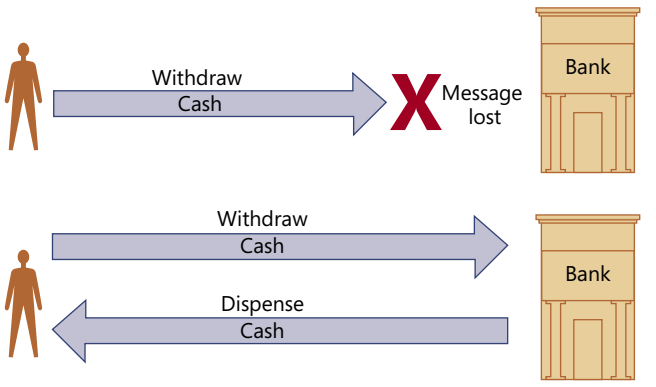
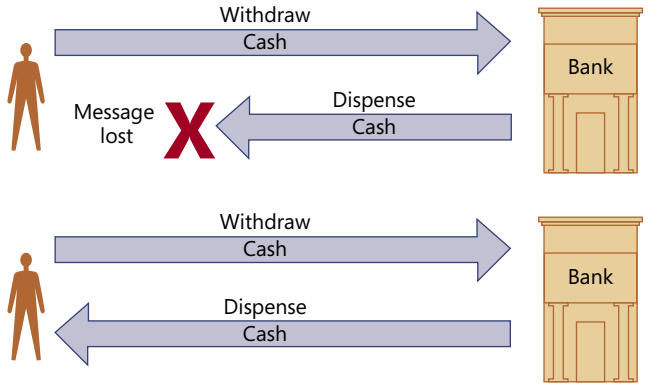


Figure 3 Withdraw cash request retry where the customer loses money



Talk. If on the other hand your application requires the message manipulation, data integrations, or orchestration features that BizTalk provides, then Service Broker is probably not the right solution.

While MSMQ recoverable messaging doesn't provide the reliability levels that the SQL Server-based options provide, it does have the advantage of not requiring SQL Server at both messaging endpoints. If supporting a database at both endpoints is not an option and the application's reliability requirements can be met by MSMQ, then MSMQ is probably the right choice as the messaging infrastructure. If both of the communicating services require data storage, however, the extra reliability of storing messages in the database is worthwhile.

In our ATM example, the ATM service will require local data storage for auditing, local storage of data for offline operation, and storage of reference data, so there will probably be a database in the ATM anyway and one of the SQL Server–based options is appropriate. The choice between Service Broker and BizTalk depends on the nonmessaging requirements of the application, the resources available at the ATM, and the message volume requirements.

Execution Reliability

Previously we discussed reliability in message delivery from one service to another. Not surprisingly, we found that the amount of reliability required depends on what the application is doing and how critical the message data is to the application. Here we will assume that messages are transferred with the required degree of reliability to a service and examine the reliability requirements for the service processing the messages.

One of the unique requirements of a service that processes asynchronous messages is that receiving a message from the queue is a "pull" operation. In other words, when a message arrives in a queue, it will sit there until an application executes a "receive" operation to retrieve and process the message. This requirement means that an asynchronous service must ensure that it executes when there are messages in the queue to be processed. The most common way to achieve this requirement is to make the service a Windows service managed by the Windows Service Control Manager (SCM). The SCM will ensure the service is started when Windows starts and can be configured to restart the service if it fails for some reason.

While this configuration generally provides the required level of reliability and is generally the preferred solution when messages arrive at a constant rate, it can cause problems if the message load varies significantly. If the service is configured with enough resources to handle peak loads, it will be wasting resources when the message load is low; and if it is configured to handle the average load, it will get far behind during peak loads. BizTalk messaging runs as a Windows service so a BizTalk application can rely on the Windows service to be there to handle incoming messages.

MSMQ addresses the message load problem with triggers that start a messaging processing service each time a message arrives in the queue. While this processing works well when messages arrive infrequently, when message load is high, the overhead of starting a thousand copies of the services can be more than the service logic itself.

Service Broker provides a feature called *activation* to solve this problem. When a message arrives in an empty queue, Service Broker will start a stored procedure to handle the message. This stored procedure will wait in a loop for more messages to arrive and continue in this loop until the queue is empty. If Service Broker determines that the stored procedure is not keeping up with the messages coming in, it will start additional copies of the stored procedure until there are enough copies running to keep up. When the message arrival rate decreases, the queue will be empty and the extra copies will exit. Then there will always be approximately the right number of resources available to service the incoming messages. Because Service Broker starts these procedures, it will be notified if one fails and replace the failing copy. If the service is an external application instead of a stored procedure, Service Broker provides events that an external application can subscribe to that will notify the service when more resources are required to process messages in a queue.

Lost Message

The other reliability issue that service execution must work with is a service failure while processing a message. If a service deletes a message from the queue as soon as it has received it and then fails before completely processing the message, that message is effectively lost. Similarly, if the service waits until it has completely processed the message before removing it from the queue, a failure between the processing step and the message removal will result in the message still being in the queue when the service restarts, and it will be processed again.

As mentioned earlier, processing the same message multiple times is not a problem if the message is a balance query, but processing a withdrawal twice can be irritating to the customer involved. The only real way to ensure that each message is processed "exactly once" is for the message processing and the queue deletion to be part of the same transaction. If there is a failure in processing, both the processing changes and the message deletion are rolled back so everything is back to the way it was before the message was received.

A single commit operation commits both the message receive and the message processing actions. Similarly, if processing the message

generates an outgoing message, the "send" of the outgoing message should be part of the transaction to avoid the situation where the service rolls back but the response message is still sent. This type of message processing is called *transactional messaging*. Most reliable messaging infrastructures support transactional messaging.

Because the messages are stored in a different file than the database, MSMQ transactional messaging requires a two-phase commit to ensure both parts of the transaction are committed. Because Service Broker SEND and RECEIVE commands are TSQL commands, the messaging and data update operations in a Service Broker service can be executed from the same SQL Server connection and be part of the same SQL Server transaction. Therefore, a two-phase commit is not required, which makes Service Broker's implementation of transactional messaging significantly more efficient than MSMQ's implementation.

Again, transactional messaging is required to make a non-idempotent service behave like an idempotent service to eliminate the issues caused by message retries. If the service is inherently idempotent, then transactional messaging is not required. If transactional messaging is not used, the service requester must be prepared to receive a response multiple times, sometimes over a long time span.

Data Reliability

Now we will look at the impact of data on service reliability. Most services access data while processing service messages, so data reliability is tightly bound to service reliability.

One of the unique aspects of asynchronous service execution is that in many cases the service messages are valuable business objects. In our ATM service, for example, if balance change messages are lost because of a failure, the account balance is not changed and the bank loses money. For this reason, storing messages in the database so they enjoy the same reliability, redundancy, and availability protections that the rest of the data stored there enjoy makes a lot of sense. If the balance change messages in our example are stored in the same database as the accounts, they will only be lost if the accounts are also lost. The backups, log backups, and Storage Area Network (SAN) features used to ensure that the bank account information is not lost also apply to the balance change messages, making the reliability of the service extremely high. If your message reliability requirements are high, Service Broker or BizTalk have significant reliability advantages because they store messages in the database.

One of the new features of SQL Server 2005 that can improve service reliability is database mirroring (DBM). DBM provides reliability by maintaining a secondary copy of a database that is maintained transactionally consistent with the primary database by applying each transaction that commits on the primary to the secondary copy before returning control to the service. If the primary fails, the secondary copy of the database can become the primary in a few seconds.

Service Broker takes advantage of database mirroring to improve messaging reliability. If the account database is a DBM pair of databases, the Service Broker database in the ATM will open network connections to both the primary and secondary databases and send messages to the primary. If the secondary database becomes the primary, Service Broker is notified immediately and messages are routed to the new primary with no user intervention or interruption.

Highly data–intensive services can take advantage of another feature in SQL Server 2005 to improve reliability. The integration of the Common Language Runtime (CLR) into the SQL Server engine means that the service logic can run into the database also. Therefore, for a Service Broker service the logic, messages, execution environment, security context, and data for a service can be in the same database.

This single-location storage has many advantages in a system with high reliability requirements. Database servers generally have the hardware and software features to maintain the high reliability required by a database. This reliability can now apply to all aspects of the service implementation. In the unlikely event of a service failure, the whole environment of the service can be restored to a transactionally consistent state with the database recovery features. Not only is the data saved, but any operations in progress are rolled back and restarted in the same execution and security environment that they were in at the time of failure.

The loosely coupled, asynchronous nature of service-oriented applications imposes some unique reliability requirements. When architecting services, the level of reliability required for the service must be understood and taken into account. Microsoft offers a variety of infrastructures for implementing services that offer different levels of reliability. Choosing the right infrastructure involves matching the level of reliability required against the capabilities of the infrastructure. The new features of SQL Server 2005 provide a service-hosting infrastructure that offers unprecedented levels of reliability for services that require very high reliability levels. •

Resources

Service Oriented Architecture
http://msdn.microsoft.com/architecture/soa/

"An Overview of SQL Server 2005 for the Database Developer," Matt Nunn (Microsoft Corporation, 2005)
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql_ovyukondev.asp

"Building Reliable, Asynchronous Database Applications Using Service Broker," Roger Wolter (Microsoft Corporation, 2005)
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql2k5_SrvBrk.asp

MSDN Magazine
Distributed .NET – "Learn the ABCs of Programming Windows Communication Foundation," Aaron Skonnard (Microsoft Corporation, 2006)
http://msdn.microsoft.com/msdnmag/issues/06/02/WindowsCommunicationFoundation/

Microsoft Windows Server System – Microsoft BizTalk Server
www.microsoft.com/biztalk/

About the Author

Roger Wolter is a solutions architect on the Microsoft Architecture Strategy Team. Roger has 30 years of experience in various aspects of the computer industry including jobs at Unisys, Infospan, Fourth Shift, and the last seven years as a program manager at Microsoft. His projects at Microsoft include SQLXML, the SOAP Toolkit, the SQL Server Service Broker, and SQL Server Express. His interest in the Service Broker was sparked by a messaging-based manufacturing system he worked on in a previous life. He also wrote *The Rational Guide to SQL Server 2005 Service Broker Beta Preview* (Rational Press, 2005).

A Flexible Model for Data Integration

by Tim Ewald and Kimberly Wolk

Summary

There are many challenges in systems integration for architects and developers, and the industry has focused on XML, Web services, and SOA for solving integration problems by concentrating on communication protocols, particularly in regard to adding advanced features that support message flow in complex network topologies. However, this concentration on communication protocols has taken the focus away from the problem of integrating data. Flexible models for combining data across disparate systems are essential for successful integration. These models are expressed in XML schema (XSD) in Web service–based systems, and instances of the model are represented as XML transmitted in SOAP messages. In our work on the architecture of the MSDN TechNet Publishing System (MTPS) we addressed three pitfalls. We'll look at what those pitfalls are and our solutions to them in the context of a more general problem, that of integrating customer information.

Systems integration presents a wide range of challenges to architects and developers. Over the last several years, the industry has focused on using XML, Web services, and service-oriented architecture (SOA) to solve integration problems. Much of the work done in this space has concentrated on communication protocols, especially on adding advanced features designed to support messages flowing in complex network topologies. While there is undoubtedly some value in this approach, all of this work on communication protocols has taken focus away from the problem of integrating data.

Having flexible models for combining data across disparate systems is essential to a successful integration effort. In Web service–based systems, these models are expressed in XSD. Instances of the model are represented as XML that is transmitted between systems in SOAP messages. Some systems map the XML data into relational databases; some do not. From an integration perspective, the structure of those relational database models is not important. What matters is the shape of the XML data model defined in XSD.

There are three pitfalls that Web service–based data integration projects typically fall into. All three are related to how they define their XML schemas. We confronted all three in our work on the architecture of the MSDN TechNet Publishing System (MTPS), the next-generation XML-based system that serves as the foundation for MSDN2. We will look at our solutions in the context of integrating customer information.

The Essential Data Integration Problem

Imagine you work at a large company. Your company has many outward-facing systems that are used by customers to accomplish a variety of tasks. For instance, one system offers customized product information to registered users who have expressed particular interests. Another system provides membership management tools for customers in your partner program. A third system tracks customers who have registered to come to upcoming events. Unfortunately, the systems were all developed separately, one of them by a separate company that your company acquired a year ago. Each of these systems stores customer-related information in different formats and locations.

> "HAVING FLEXIBLE MODELS FOR COMBINING DATA ACROSS DISPARATE SYSTEMS IS ESSENTIAL TO A SUCCESSFUL INTEGRATION EFFORT"

This setup presents a critical problem for the business: it doesn't have a unified view of a given customer. This problem has two effects. First, the customer's experience suffers because the single company with which they are doing business treats them as different people when they use different systems. For example, a customer who has expressed her desire to receive information about a given product through e-mail whenever it becomes available has to express her interest in that product a second time when she registers for particular talks at an upcoming company-sponsored event. Her experience would be more seamless if the system that registered her for an upcoming event already knew about her interest in a particular product.

Second, the business suffers because it does not have an integrated understanding of its customers. How many customers who are members of a partners' program are also receiving information about products through e-mail? In both cases, the divisions between the systems that the customer works with are limiting how well the company can respond to its customers' needs.

Whether this situation arose because systems were designed and developed individually without any thought to the larger context within which they operate or because different groups of integrated systems were collected through mergers and acquisitions is irrelevant. The problem remains: the business must integrate the systems to both

improve the customer experience and their knowledge of who the customer is and how best to serve them.

The most common approach to solving this problem is to mandate that all systems adopt a single canonical model for a customer. A group of architects gets together and designs the company's single format for representing customer data as XML. The format is defined using a schema written in XSD. To enable systems to share data in the new format, a central team builds a new store that supports it. The XSD data model team and store team deliver their solution to all the teams responsible for systems that interact with customers in some way and require that they adopt it. The essential change is shown in Figures 1 and 2.

Each system is modified to use the underlying customer data store through its Web service interface. They store and retrieve customer information as XML that conforms to the customer schema. All of the systems share the same service instance, the same XSD data model, and the same XML information.

This solution appears to be simple, elegant, and good, but a naïve implementation will typically fail for one of three reasons: demanding too much information, no effective versioning strategy, and no support for system-level extension.

Figure 1 Three separate data stores, one per system

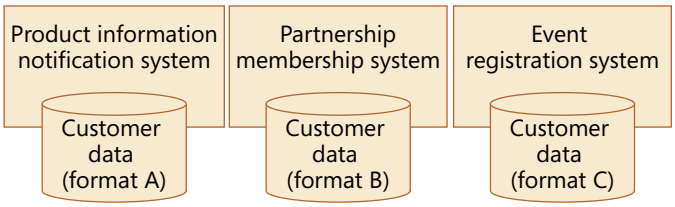
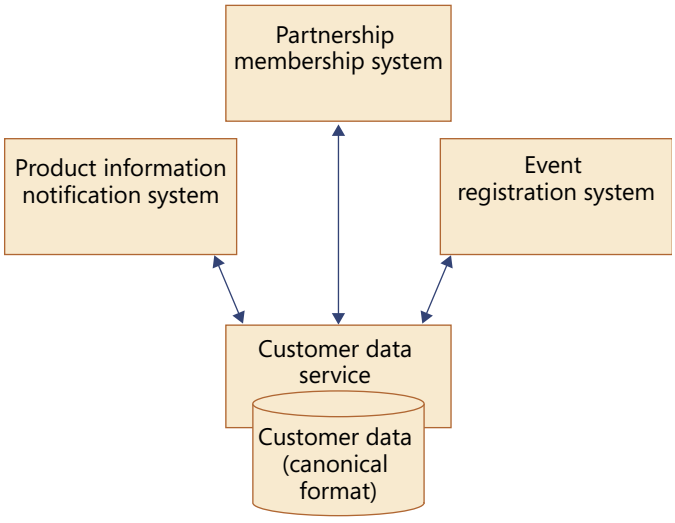


Figure 2 A single data store and format



Demanding Too Much Information

The first potential cause for failure is a schema and store that require too much information. When people build a simple Web service for point-to-point integration, they tend to think of the data their particular service needs. They define a contract that requires that particular data be provided. When a contract is generated from source code, this data can happen implicitly. Most of the tools that map from source code to a Web service contract treat fields of simple value types as required data elements, insisting that a client send it. Even when a contract is created by hand, there is still a tendency to treat all data as required. As soon as the service determines (by schema validation or code) that some required data is not present, it rejects a request. The client gets a service fault.

This approach to defining Web service contracts is too rigid and leads to systems that are very tightly coupled. Any change in the service's requirements forces a change in the contract and in the clients that consume it. To loosen this coupling, you need to separate the definition of the shape of the data a service expects from a service's current processing requirements. More concretely, the data formats defined by your contract should treat everything beyond identity data as optional. The implementation of your service should enforce occurrence requirements internally at run time (either using a dedicated validation schema or code). It should be as forgiving as possible when data is not present in a client request and degrade gracefully.

In the customer information example, it is easy to think of cases where some systems want to work with customers but do not have complete customer information available. For instance, the system that records a customer's interest in a particular product might only collect a customer's name and preferred e-mail address. The event registration system, in contrast, might capture address and credit card information as well. If a common customer data model requires that every valid customer record include name, e-mail, address, and credit card information, neither system can adopt it without either collecting more data than it needs or providing bogus data. Making all the data other than the identity (ID number, e-mail address, and so forth) optional eases adoption of the data model because systems can simply supply the information they have.

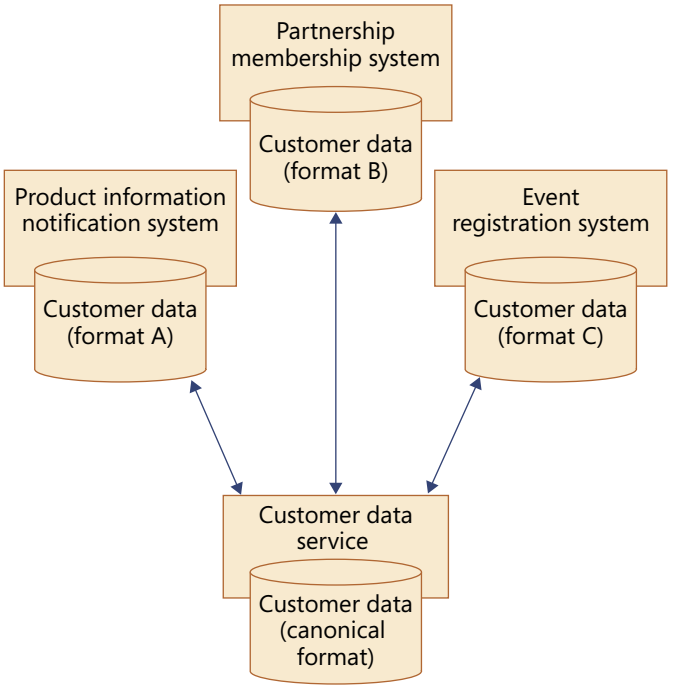
By separating the shape of data from occurrence requirements, you make it easier to manage change in the implementation of a single service. It is also critical when you are defining a common XML schema to be used by multiple services and clients. If too much information is mandatory, every system that wants to use the data model may be missing some required piece of information. That leaves each system with the choice of not adopting the shared model and store or providing bogus data (often the default value of a simple programming language type). Either option can be considered a failure.

You gain a lot of flexibility for systems to adopt the model by loosening the schema's occurrence requirements nearly completely. Each system can contribute as much data as it has available, which makes a common XML schema much easier to adopt. The price is that systems receiving data must be careful to check that the data they really need is present. If it is not present, they should respond accordingly by getting more data from the user or some other store, by downgrading their behavior, or—only in the worst case—generating a fault. What you are really doing is shifting some of the constraints you might normally put in an XML schema into your code, where they will be checked at run time. This shifting gives you room to change those constraints without revising the shared schema.

No Effective Versioning Strategy

The second potential cause for failure is the lack of a versioning strategy. No matter how much time and effort is put into defining an XML

Figure 3 A combination of stores (see Figures 1 and 2)



schema up front, it will need to change over time. If schema, the shared store that supports them, and every system that uses them has to move to a new version all at once, you cannot succeed. Some systems will have to wait for necessary changes because other systems are not at a point where they can adopt a revision. Conversely, some systems will be forced to do extra, unexpected work because other systems need to adopt a new revision. This approach is untenable.

To solve this problem you need to embrace a versioning strategy that allows the schema and store to move ahead independently of the rate at which other systems adopt their revisions. This solution sounds simple and it is, as long as you think about XML schemas the right way.

Systems that integrate using a common XML schema view it as a contract. Lowering the bar for required data by making elements optional makes a contract easier to agree to because systems are committing to less. For versioning, systems also need to be allowed to do more *without changing schema namespace*. What this means in practical terms is that a system should always produce XML data based on the version of the schema it was developed with. It should always consume data based on that same version *with additional information*. This definition is a variation on Postel's Law: "Be liberal in what you accept; conservative in what you send." Arguably, this idea underlies all successful distributed systems technologies, and certainly all loosely coupled ones. If you take this approach, then you can extend a schema without updating clients.

In the customer example, an update to the schema and store might add support for an additional optional element that captures the user's mother's maiden name for security purposes. If systems working with the old version generate customer records without this information, it's okay because the element is optional. If they send those records to other systems that require this information, the request may fail, and that is okay too. If new systems send customer data including the mother's maiden name to old systems, that is also okay because they are designed to ignore it.

Happily, many Web service toolkits support this feature directly in their schema-driven marshaling plumbing. Certainly the .NET object-XML mappers (both the trusty XmlSerializer and the new XmlFormatter/DataContract) handle extra data gracefully. Some Java toolkits do too, and frameworks that support the new JAX-WS 2.0 and JAXB 2.0 specifications will as well. Given that, adopting this approach is pretty easy.

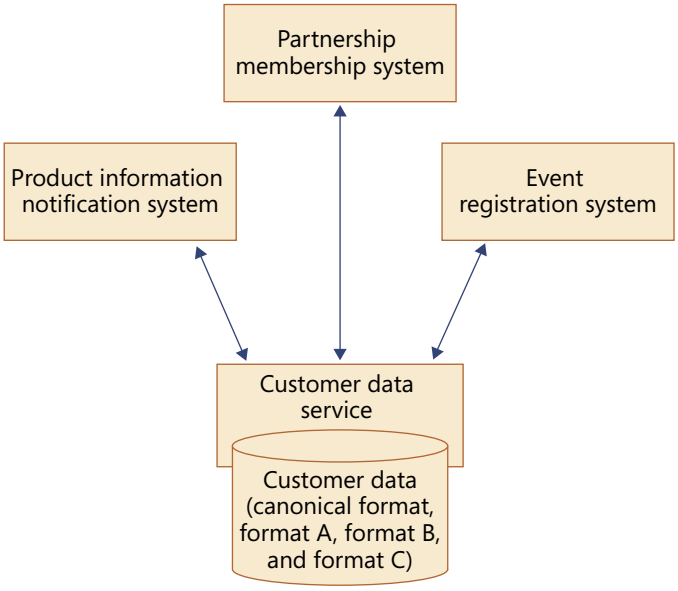
The only real problem with this model is that it introduces multiple definitions of a given schema, each representing a different version. Given a piece of data—an XML fragment captured from a message sent on the wire, for instance—it is impossible to answer the question: "Is this data valid?" The question of validity can only be answered relative to a particular version of the schema. The inability to definitively state whether a given piece of data is valid presents an issue for debugging and possibly also for security. With data models described with XML schema, it is possible to answer a different and more interesting question: "Is this data valid enough?"

This question is really what systems care about, and you can answer it using the validity information most XML-based schema validators provide, which is a reasonable path to take in cases where schema validation is required, and it can be implemented with today's schema validation frameworks.

No Support for System-Level Extension

The third potential cause for failure is lack of support for system-specific extensions to a schema. The versioning strategy based on the notion that a schema's definition changes over time is necessary to promote adoption, but it is not sufficient. While it frees systems from having to adopt the latest schema revision immediately, it does nothing to help systems that are waiting for specific schema updates. Delays in revisions can also make a schema too expensive for a system to adopt. The solution to this last problem is to allow systems that adopt a common schema to extend it with additional information of their own. The extension information can be stored locally in a system-level store (see Figure 3).

Figure 4 The same store (see Figure 3) with data formats in a shared store



In this case, each system is modified to write customer data to both its dedicated store using its own data model and to the shared store using the canonical schema. It is also modified to read customer data from both its dedicated store and the shared store. Depending on what it finds, it knows whether a customer is already known to the company and to the system. Table 1 summarizes the three possibilities.

The system can use this information to decide how much information it needs to gather about a customer. If the customer is new to the company, the system will add as much information as it can to the canonical store. That information becomes available for other systems that work with customers. It may also store data in its dedicated store to meet its own needs. This model can be further expanded so that system-specific data is stored in the shared store as well (see Figure 4).

This solution makes it possible for systems to integrate with one another using extension data that is beyond the scope of the canonical schema. To work successfully, the store and other systems need to have visibility into the extension data. In other words, it cannot be opaque. The easiest way to solve this problem is to make the extension data itself XML. The system providing the data defines a schema for the extension data so that other systems can process it reliably. The shared store keeps track of extension schemas so it can ensure that the extension data is valid, even if it does not know explicitly what the extension contains. In the most extreme case, a system might choose to store an entire customer record in a system-specific format as XML extension data. Other systems that understand that format can use it. Systems that do not understand it rely on the canonical representation instead.

> "THE INABILITY TO DEFINITIVELY STATE WHETHER A GIVEN PIECE OF DATA IS VALID PRESENTS AN ISSUE FOR DEBUGGING AND POSSIBLY ALSO FOR SECURITY"

When systems are independent, they each control their own destiny. They can capture and store whatever information they need in whatever format and location they prefer. The move to a single common schema and store changes that. If adopting a common XML data format restricts a system's freedom to deliver required functionality, you are doomed to fail.

Using a combination of typed XML extension data in either a system-level or the shared store adds complexity because you have to keep data synchronized. But it also provides tremendous flexibility. You can align systems around whatever combination of the canonical schema and alternate schemas you want. You can drive toward one XML format over time, but you always have the flexibility to deviate from that format to meet new requirements. This extra freedom is worth a lot in the uncertain world of the enterprise.

A further, subtle benefit of this model is that it allows the team defining the common schema to slow their work on revisions. Systems can use extension data to meet new requirements between revisions. The team working on the canonical model can mine those extensions for input into their revision process. This feedback loop helps ensure that model changes are driven by real system requirements.

Table 1 The three possible cases of customer data

Record in shared store	Record in system store	Meaning
No	No	Customer is new to the company. Collect all common and system-specific data.
Yes	No	Customer is known to the company but new to the system. Collect system-specific data.
Yes	Yes	Customer is known to the company and to the system.

Mitigate the Risks

Lots of organizations are working on integrating systems using XML data described using XML schema and exchanged through Web services. In this discussion we presented three common causes for failure in these data-centric integration projects: demanding too much information, no effective versioning strategy, and no support for system-level extensions. To mitigate these risks:

- Make schema elements optional, and encode system-specific occurrence requirements as part of that system's implementation.
- Build systems that produce data according to their version of a shared schema but consume so that systems can adopt schema revisions at different rates without changing schema namespace.
- Allow systems to extend shared schemas with their own data to meet new requirements independent of data-model revisions.

All of these solutions are based on one core idea: to integrate successfully without sacrificing the agility that systems need to be able to agree on as little as possible and still get things done. So does it all work? The answer is yes. These techniques are core to the design of the MSDN/TechNet Publishing System, which underlies MSDN2. •

Resources

MSDN Magazine
http://msdn.microsoft.com/msdnmag/05/02/InsideMSDN/

MSDN2 Library
http://msdn2.microsoft.com/en-us/library/

About the Authors

Tim Ewald is a principal architect at Foliage Software Systems where he helps customers design and build applications ranging from enterprise IT to medical devices. Prior to joining Foliage, Tim worked at Mindreef, a leading supplier of Web services diagnostics tools, and before that Tim was a program manager lead at MSDN, where he worked with Kim Wolk as co-architects of MTPS, the XML and Web service–based publishing engine behind MSDN2. Tim is an internationally recognized speaker and author.

Kim Wolk is the development manager for MSDN and the driving force behind MTPS, the XML and Web services-based publishing engine behind MSDN2. Previously, she worked as MTPS co-architect and lead developer. Before joining MSDN, Kim spent many years as a consultant, both independently and with Microsoft Consulting Services, where she worked on a wide range of mission-critical enterprise systems.

Autonomous Services and Enterprise Entity Aggregation

by Udi Dahan

Summary

Enterprises today depend on heterogeneous systems and applications to function. Each of these systems manages its own data and often doesn't explicitly expose it for external consumption. Many of these systems depend on the same basic concepts like *customer* and *employee* and, as a result, these entities have been defined in multiple places in slightly different ways. Entity aggregation embodies the business need to get a 360-degree view of those entities in one place. However, this business need is only one symptom of the larger issue: business/IT alignment. Service-oriented architectures (SOAs) have been hailed as the glue that would bring IT closer to business, yet the hype is already fading. We'll take a look at concrete ways that autonomous services can be used to transform the way we develop systems to more closely match business processes and solve immediate entity aggregation needs.

The term SOA has gained popularity over the past year and has become the buzzword de jour. Everything these days is service-oriented, SOA-enabled, or "the key to your SOA success." The industry continues to struggle with what defines a service; however, various properties of services do appear to be well accepted. Microsoft's tenets of service orientation define four such properties: services are autonomous; services have explicit boundaries; services share contract and schema, not class or type; and service compatibility is based on policy.

While these tenets act as important architectural guidelines for developing complex software, there is obviously more to be said. Many run-time aspects of services like scalability, availability, and robustness are not addressed by service orientation. It is exactly these run-time aspects that are the focus of autonomous services.

Service Autonomy

The phrase "autonomous services" seems to be a simple rewording of the first tenet, and yet the two have very different meanings. "Services are autonomous" means that teams developing cooperating services could operate independently—to a degree of course. When taken together with the tenet about contract and schema, it is clear that there need be no binary dependencies among those teams. The development of each service could be done on a different platform, using different languages and tools. An *autonomous service*, on the other hand, is a service whose ability to function is not controlled or inhibited by other services.

The word *autonomous* has many definitions including: self-governing, self-controlling, independent, self-contained, and free from external control and constraint. In the light of these definitions of autonomy, we will examine two kinds of service interaction: synchronous and asynchronous communication.

In Figure 1 we can see that Service A needs to actively hold runtime resources (the calling thread) until Service B replies. The time it takes Service A to respond to a single request depends on its interaction with Service B. Service A is affected if the network is slow or if Service B is unavailable or slow. Therefore, it does not appear that Service A is "free from external constraint" in this case.

Another issue to consider here is coupling. While Service A and Service B may be loosely coupled in that they were developed separately by different teams on different platforms sharing only a WSDL file, we can see that they are tightly coupled in time. This temporal coupling

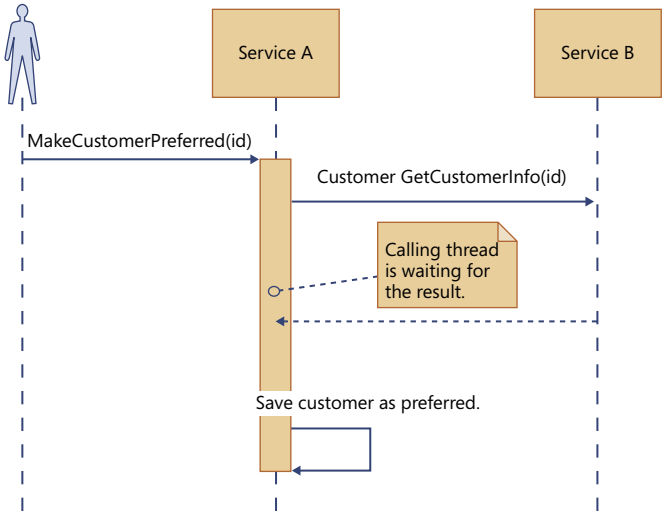
> "MANY RUN-TIME ASPECTS OF SERVICES LIKE SCALABILITY, AVAILABILITY, AND ROBUSTNESS ARE NOT ADDRESSED BY SERVICE ORIENTATION"

can be seen in that the time it takes Service A to respond to a request includes Service B's processing time. It is exactly this coupling that causes undesired failures in Service B to ripple into Service A.

There are two ways that we can break this temporal coupling. One way is for Service A to poll Service B for the result. Unfortunately, polling leads to undue load on Service B (as a function of number of consumers and the polling interval), and consumers get the requested information later than the time available. Figure 2 shows the complexity of this solution.

If we continue our analysis, we'll find that spawning a new thread, or even using the thread pool to handle the polling for each request we send, is going to drain Service A of its resources fairly quickly. A more performant (and even more complex) solution would involve a single thread that manages polling for all the requests sent. That thread would marshal the results back as they became available to a different thread, which would finish the processing of the original

Figure 1 Synchronous communication



request. We would do well to heed Occam's razor before continuing down this path.

A different solution to the problem of temporal coupling that avoids the issues stated previously is to use asynchronous communication between these services (see Figure 3). In this case Service A subscribes to events published by Service B about changes to its internal state. When these events occur, Service A stores the data that it considers relevant. Thus, when Service A receives a request it is no longer dependent on external parties for processing leaving its availability unaffected. Notice that the load on Service B is even lower than in the original synchronous communication example since it no longer receives those requests. Modern publish/subscribe and messaging infrastructure can keep the load near constant no matter how many consumers there are. Data freshness is also improved with asynchronous communication; Service A receives the data much closer to the time that Service B made it available. We need not make trade-offs on load (as a result of the polling interval) against data freshness.

Technical Boundaries and Data Replication

Although the second tenet seems clear at first glance, the nature of a boundary isn't at all obvious. Are the boundaries of Service A and Service B in the previous examples any different in the synchronous and asynchronous cases? It does not appear so, but there is one major technical difference: transactions.

To handle a single request properly, in cases where that request causes data to be changed, the handling of the request should be done within the context of a transaction so that the state of the service stays consistent. If that service has to interact with other services to handle the request, and as a result those services change their data as well, should those changes occur within the original transaction context?

When service interaction is synchronous, the division of responsibility between services may often require that changes across service be performed within a single transaction. When services interaction is asynchronous (or synchronous with polling) we avoid changing data in other services altogether, so there is no need to have transactions cross service boundaries. Obviously, if a transaction starting in one service were to lock resources in other services, this would require

Figure 2 Synchronous communication with polling

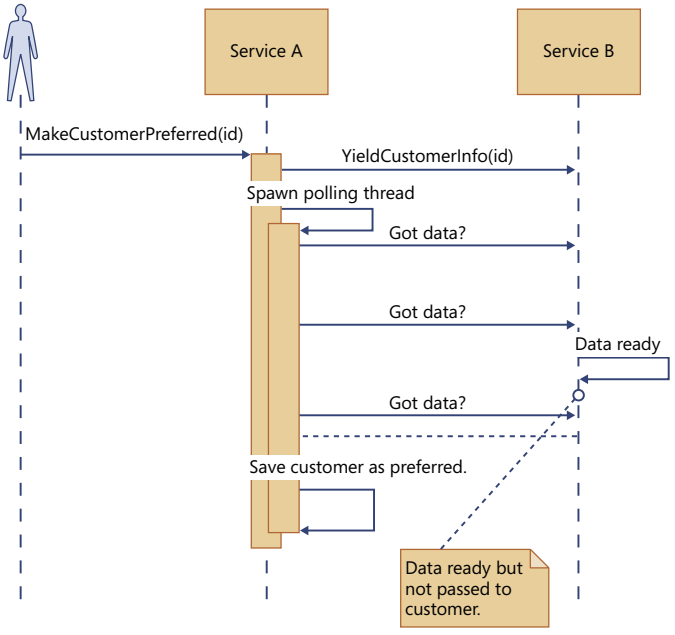
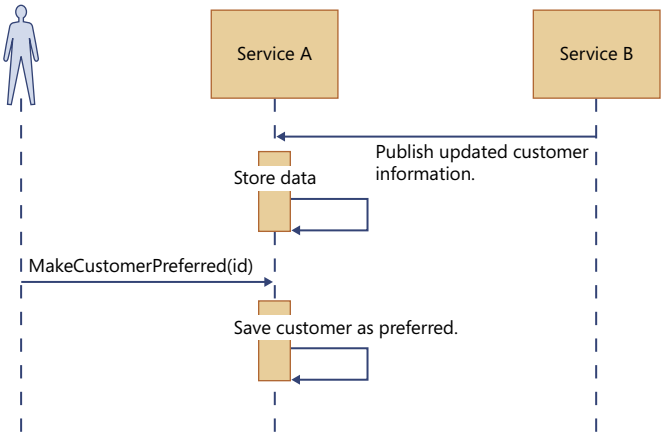


Figure 3 Asynchronous communication

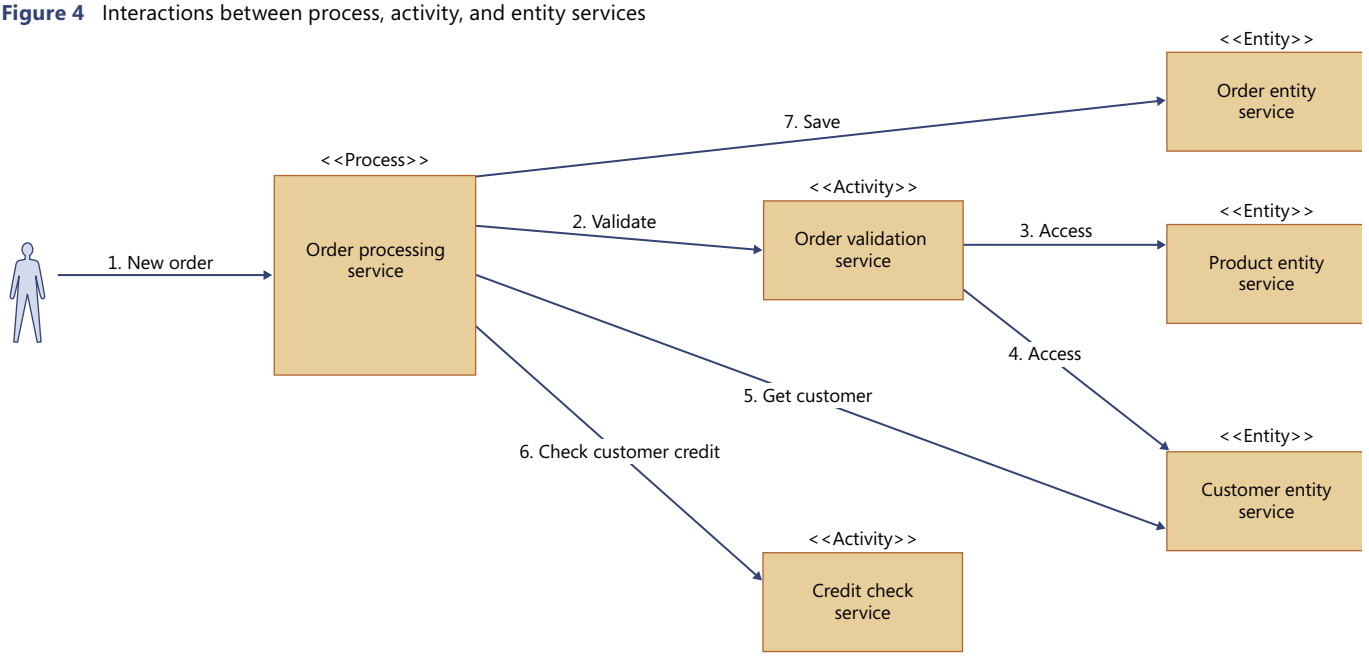


high levels of trust between all involved services and blur the distinction of where one service ended and another began.

Let us define autonomous services. It is clear that autonomous services span much more than low-level communications, encompassing many aspects including trust and reliability. However, we have seen that constraining interactions between services to asynchronous messaging has guided our architecture in such a way that autonomous, loosely coupled services have crystallized. In fact, autonomous services appear to expand on service orientation (or constrain its degrees of freedom) by adding a new tenet: *a service interacts with other services using asynchronous communication patterns*.

Note that while a service may consume other services asynchronously, this consumption does not necessarily mean that it cannot expose a synchronous interface. Google and Amazon do exactly that. The Web services that they expose are synchronous in nature but it has no effect on their autonomy.

Figure 4 Interactions between process, activity, and entity services



At first glance it appears that the use of publish/subscribe communications leads to data duplication between services similar to what happens when doing data replication. Data replication techniques such as extract, transform, and load (ETL) or file transfer handle transferring data between low-level data sources like databases and directories. This transfer bypasses higher-level logical constructs for managing that data coherently, which often leads to duplicating those same logical constructs at the consumer end of the transfer.

> "ENTITY AGGREGATION REPRESENTS THE BUSINESS NEED TO GET A GLOBAL VIEW OF DATA ACROSS THE ENTERPRISE"

The data that flows through a publish/subscribe interaction behaves differently. When a service publishes a message, that message must be part of the service contract—that contract is independent of the underlying data store's schema. The process of building the message—retrieving the appropriate data from the data store and transforming it to the message schema—goes through all the service layers. Services that consume these messages do not need to implement the same logic. Furthermore, the decision of when to publish the message is a logical decision for which code has been written; it is not a low-level detail of when the ETL script was scheduled to run.

The most important difference between simple data replication and autonomous service interaction is that the consumer service decides to save only the data that it needs. There is no longer a "back door" into the service database. When the consumer service receives the message that was published, the data within the message does not bypass any of the layers in the service until it reaches the database (see Resources).

When using these kinds of asynchronous service interactions, we often find that our services tend to be larger and coarser grained, often containing databases of their own and hosted on their own servers or datacenters. By keeping transaction contexts constrained to the scope of a single service, the responsibilities of that service tend to expand to the level of a business function or department—the natural boundary found in the business domain. This effect is quite understandable when we view the levels of coupling at different levels of business. Departments are loosely coupled to each other, collaborating without intimate knowledge of each other's inner workings. Groups internal to a department often require much deeper understanding into the workings of parallel groups to get the job done.

We can see that this architectural style in no way contradicts any of the four tenets, yet familiar service types found when using service orientation no longer fit.

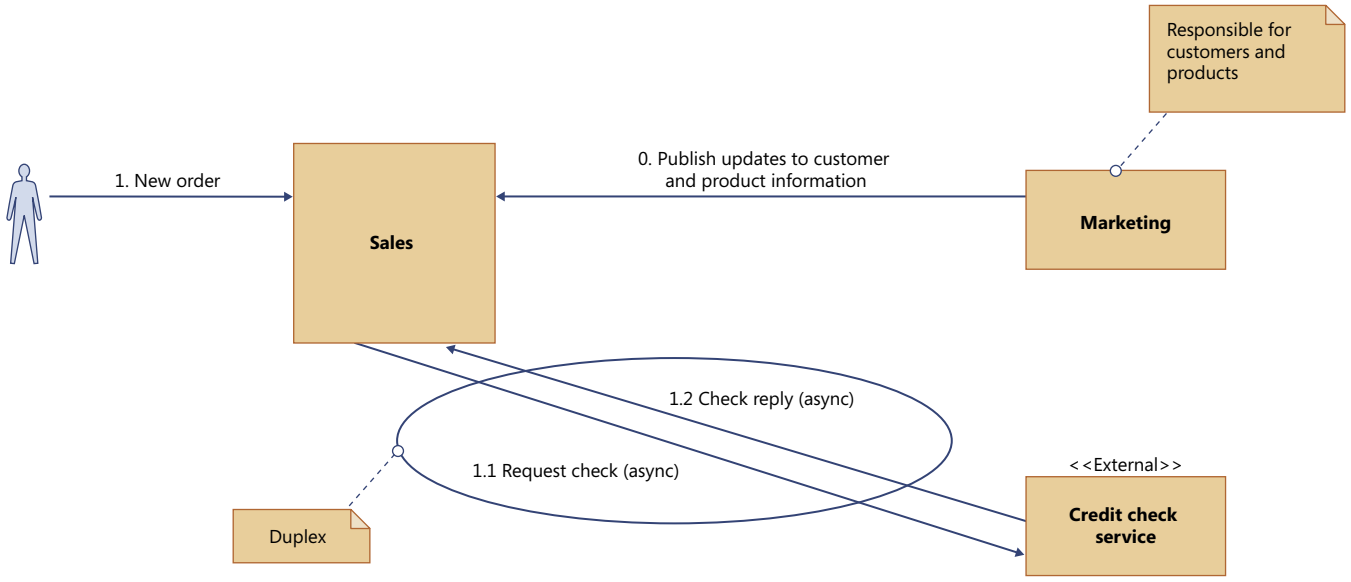
Common Boundary Pitfalls

Process services, activity services, and entity services were once proposed as the way to do service orientation. Process services manage long-running business processes. Activity services manage atomic operations that encapsulate interaction with more than one entity service. Entity services manage interaction with a single business entity. In this model entity aggregation occurs at the entity service level. The problem with this choice of service boundaries manifests itself in the synchronous interservice flows (see Figure 4).

Although it is quite possible to model the same interaction with asynchronous messaging (specifically with respect to an external credit check service), the value in separating order processing into three services is unclear. It is unlikely that a different team would be working on each of the order services or that they would run on different platforms. The tight coupling between these services is inherent. All of them work with order processing; not sharing a common Order class would probably cause code duplication, but that duplication would go against the third tenet of service orientation. There is only one conclusion: we cannot separate our services along the process/activity/entity boundaries. Consider the result of modeling this business process using autonomous services (see Figure 5).

The autonomous services present in Figure 5 represent one possible division correlating to a given organization; different enterprises would

Figure 5 Interactions between autonomous services



have different groups responsible for different business processes. There are two important differences to note here. The first is that the sales service stores the customer and product data it needs internally so that when it needs to process an order, it already has all the data it needs. The second difference is that the division of order handling between separate services does not occur here. Although it is likely that the sales service has different layers and components for handling the various steps of order processing, and possibly some kind of workflow engine to manage those steps, these are implementation details of the service.

The code duplication that arises by following the "schema/class" tenet in the previous case does not occur here, simply because the tenet does not apply within a service boundary. (The issue here is that they are all the same entity [Order] probably with the same fields. Before the entity service goes to save the order, it too must perform validation: code that's already been written for the activity service validation.) If we can no longer use entity services, then how and where does entity aggregation occur when using autonomous services?

Business-Level Entity Aggregation Requirements

Before we can delve into the technical details of the solution, we must better define the problem. As was first stated, entity aggregation represents the business need to get a global view of data across the enterprise. Various business departments require different data elements of a given entity, but seldom require all the data for that entity. This case is one in which one business department requires data that is owned by another department. For instance, the marketing department needs to know total order value per customer per quarter—data that is managed by the sales department. In this case, we are aggregating data from two existing systems into one of those systems, which is a different style of entity aggregation than what is most commonly viewed as entity aggregation; yet it is the most common case seen in the field.

Intersystem OLTP aggregation. We will start by examining a concrete example. In a typical order processing scenario we have two systems: one accepts orders from our Web site; the other is a homegrown customer relationship management (CRM) system. Marketing has a new requirement that "preferred customers" should receive a 10 percent discount on all orders. A preferred customer is defined as a customer living in the United States who has done at least $50,000 of business with the company over the last quarter, but this definition is expected to change.

After analyzing this requirement we can see that it has two parts: who is a preferred customer, and what do we do with that information? The first decision that has to be made has to do with boundaries and responsibilities: which system will be responsible for *preferred customers*, and which system will be responsible for what we do with preferred customers? In this case there is no reason for the CRM system not to take on the first responsibility and the order processing system taking on the second. The next decision that has to be made is how these systems will interact, synchronously or asynchronously.

Synchronous OLTP aggregation. The first way to handle this requirement is to add to the order processing system a query method that retrieves customers whose total orders amounted at least X over a period Y. This method gives us the ability to handle various sums and time periods as the marketing department changes their rules. The CRM system would fulfill requests for which customers are preferred customers by querying the order processing system (see Figure 6).

Autonomous OLTP aggregation. In the second approach, we view each of the systems as autonomous services that notify external consumers of meaningful events. In this case, interactions between services are more event driven. In Figure 7, when the order processing system needs to know if a given customer is a preferred customer, it does not have to communicate with the CRM system. The same goes for the order data and the CRM system. The data from these two systems have been aggregated by design.

> "MANAGING A SINGLE SERVICE THAT HANDLES ENTERPRISE-WIDE ENTITY AGGREGATION NEEDS IS MUCH MORE COST-EFFECTIVE THAN MANAGING MULTIPLE ENTITY SERVICES"

Let us explore this example a bit further as the marketing department changes the rules that define preferred customers. Preferred customers

Figure 6 Aggregating data from different systems without autonomous services

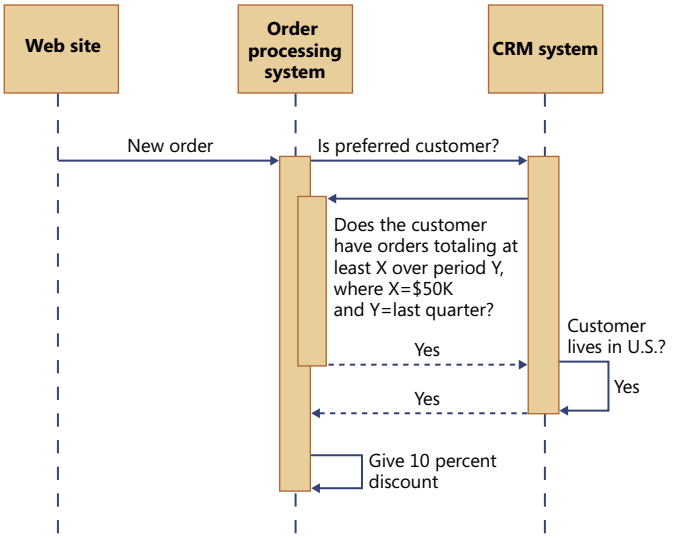


Figure 7 Aggregating data from different systems with autonomous services

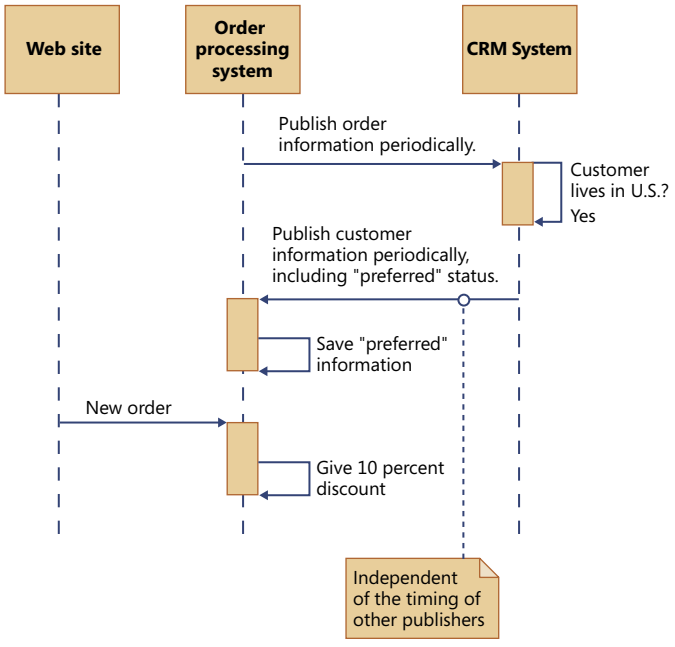


Figure 8 Changes made when not using autonomous services

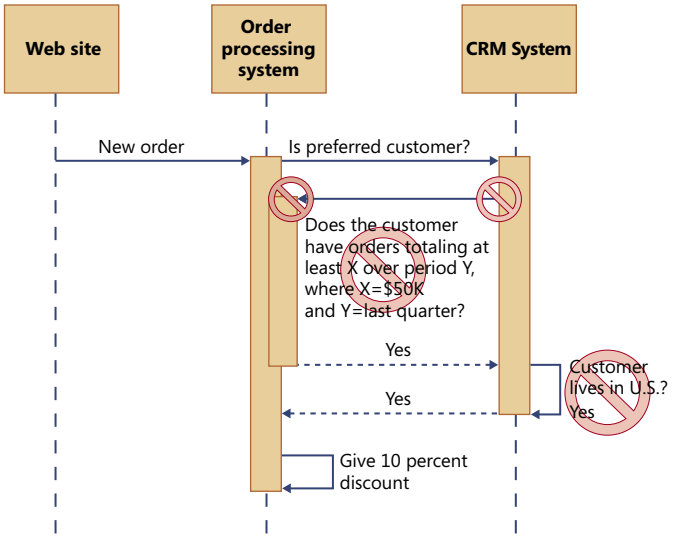
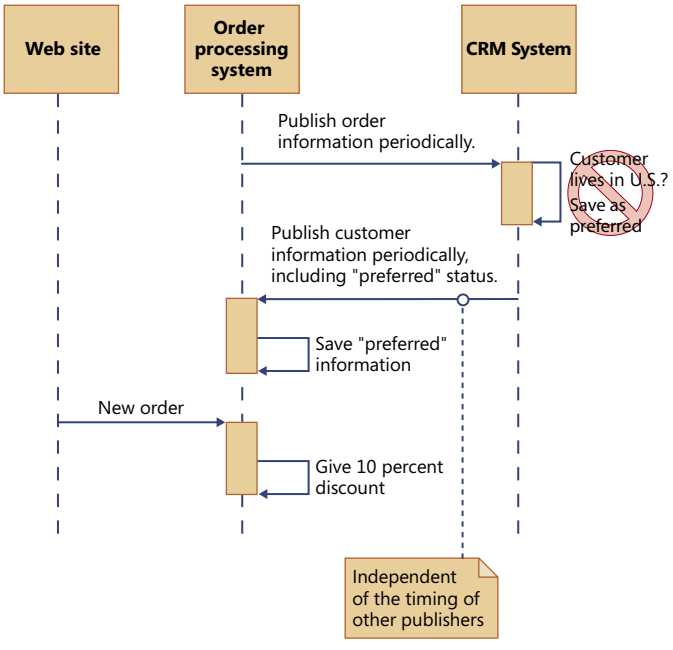


Figure 9 Changes made when using autonomous services



are now those customers living in all of North America who have placed at least three orders in the last quarter, each totaling $15,000 or more.

In our first approach, the original query we added is no longer relevant. We now need to support a different kind of query (get customers with at least X orders over period Y, with each order totaling Z or more, where X = 3, Y = last quarter, and Z = $15,000), changing the order processing system's interface, some of its implementation (to support the new interface), and the code that activates it in the CRM system (see Figure 8). In the second approach, the only changes we need to make are internal to the CRM system. Neither the interface nor the implementation of the order processing system need to be modified (see Figure 9).

The term entity aggregation brings to mind an active process of collecting data from disparate sources and merging the data together to form a cohesive whole. While the dynamics of the first approach map very well to this process, the second approach does not. However, it is clear that the second approach maintains a lower level of coupling between these two services, even as requirements change. Keep this point in mind when designing services; loose coupling between services needs to occur from both the data and the logic perspective.

Entity Aggregation for Business Intelligence

In the previous case, the aggregated data was critical to the functioning of the business department(s) involved, participating in day-to-

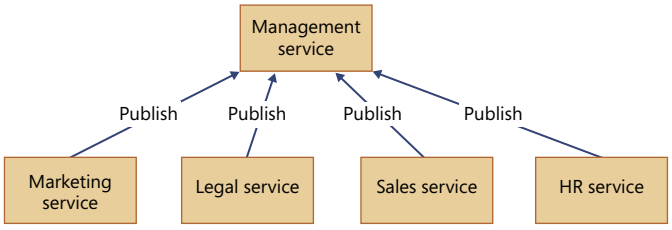
day transaction processing. However, entity aggregation is most often discussed in the management context—business wants to get a 360 degree view of the enterprise data. This context is quite different from the previous one in that the aggregated data is mainly used for decision support and business intelligence—in other words, primarily read-only usage scenarios.

A simple and effective way of modeling this context is by instituting a management service (see Figure 10). This service does not perform operations and provisioning management for other services, but rather pulls together the data published by all other services and stores it in a format optimized for its usage scenarios.

Another common business requirement that comes up in the context of entity aggregation is historical analysis. While it may not make sense for the marketing service to maintain historical data about past products no longer being offered, users of the management service may find if invaluable to compare sales and profit figures of current and past products. It would be the responsibility of the management service to manage these historical trends.

One benefit of using a management service as opposed to entity services is exactly in the operations management area. Managing a single service that handles enterprise-wide entity aggregation needs is much more cost-effective than managing multiple entity services—one for each entity that needs to be aggregated. Internal changes to any of these services that do not affect their contract may not even impact the management service. Changes to the contract that may affect numerous entities result in corresponding changes only in the management service, not in each one of the entity services that previously aggregated those entities.

Figure 10 Relationships between the management service and other autonomous services



Services may seem to be the new unit of reuse in the enterprise, but this does not comply with the tenet of autonomy, not to mention that the constraints on how we interact with a service make it distasteful. For instance, while it may seem that the management service could easily provide audit trail tracking and storage for all other services, this choice could break those services' autonomy. Should the management service go down for any reason, other services shouldn't be allowed to continue processing without auditing. Furthermore, an autonomous service could not trust another service to provide this core capability, which is not to say that you cannot encapsulate audit trail tracking (or other specific functionalities) into a component that all services reuse. Regulatory compliance issues must be taken care of within each service.

By recognizing that the requirements for OLTP and OLAP entity aggregation are different, we have been able to identify two separate, yet simple solutions using a single communication paradigm. Asynchronous messaging patterns enable the creation of autonomous, loosely coupled services that more closely resemble the business processes they model. As a result, often all that is needed to respond to changing business requirements is a local change to a single service. These small-scale changes do not affect interservice contracts and can be performed with greater certainty that other systems will not be affected and therefore in less time. Aligning IT with business has much more to do with interpersonal communications and understanding than technology, but that does not mean that technology cannot help. •

Resources

Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Gregor Hohpe and Bobby Woolf (Addison-Wesley Professional, 2003)

"Data on the Outside vs. Data on the Inside," Pat Helland (Microsoft Corporation)
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/dataoutsideinside.asp

"Dealing with Concurrency: Designing Interaction Between Services and Their Agents," Maarten Mullender (Microsoft Corporation, 2004)
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/concurev4M.asp

Microsoft Events
"MSDN Webcast: Why You Can't Do SOA Without Messaging (Level 300)," Udi Dahan (Microsoft Corporation, 2006)
http://msevents.microsoft.com/cui/WebCastEventDetails.aspx?EventID=1032273610&EventCategory=5&culture=en-US&CountryCode=US

MSDN – Channel 9 Forums
"ARCast – Autonomous Services," Udi Dahan (Microsoft Corporation, 2006) http://channel9.msdn.com/Showpost.aspx?postid=163201

"ARCast – Service Orientation and Workflow," Udi Dahan (Microsoft Corporation, 2006) http://channel9.msdn.com/ShowPost.aspx?PostID=163471

"SOA Challenges: Entity Aggregation," Ramkumar Kothandaraman (Microsoft Corporation, 2004)
http://msdn.microsoft.com/architecture/soa/default.aspx?pull=/library/en-us/dnbda/html/dngrfsoachallenges-entityaggregation.asp

About the Author

Udi Dahan is a Microsoft solutions architect MVP, a recognized .NET development expert, and the chief IT architect and C4ISR product line manager at KorenTec. Udi is known as a primary authority on SOAs in Israel and consults on the architecture and design of large-scale, mission-critical systems developed all over the country. His experience spans technologies related to command and control systems, real-time applications, and high-availability Internet services. For more information, please visit www.UdiDahan.com.

Data Replication as an Enterprise SOA Antipattern

by Tom Fuller and Shawn Morgan

Summary

As applications are envisioned and delivered throughout any organization, the desire to use data replication as a "quick-fix" integration strategy often wins favor. This path of least resistance breeds redundancy and inconsistency. Regularly applying this antipattern within an enterprise dilutes the original long-term goals of a service-oriented architecture (SOA). However, depending on the context in which it surfaces, data replication can be viewed either positively or negatively. To successfully deliver to a service-oriented strategy, enterprise architects need to draw from the successes and failures of other architectures. Architecture patterns, when discovered and documented, can provide the best technique for managing the onerous job of influencing change in your enterprise. Weíll use both an antipattern and a pattern to describe how data replication can impact your enterprise architecture.

The drive toward service-orientation is still in its infancy. For SOA to achieve the lofty goals visualized by the industry, architects will need to rely on documented best practices. This is where patterns and antipatterns will help bridge the gap between conceptualization and reality. Patterns represent a proven repeatable strategy for achieving expected results. Software architecture patterns and antipatterns provide documented examples of successes and failures in attempting to apply those strategies in IT. Through successful identification of architecture abstractions, SOA will begin to find implementation strategies that ensure successful delivery.

We'll describe why data replication, when used as a foundational pattern in your enterprise SOA, can lead to a complicated and costly architecture and provide you with a refactoring strategy to move from data replication (antipattern) to direct services (pattern). Then we'll offer a brief explanation of where data replication can be used to successfully solve some specific problem domains (replication as a pattern) without compromising other enterprise architecture initiatives. Architects will take away from this discussion a strategy for facilitating the shift away from architectural malfeasance.

To promote consistency, the software industry has settled on a template for documenting design patterns. This template, however, seemed insufficient to describe architectural antipatterns and their eventual refactored solutions. A hybrid collection of pattern sections from various resources were pulled together to help define a structure for the antipattern and pattern described here (see Resources). Table 1 shows the superset of sections and whether or not they are applied to a pattern, antipattern, or both.

Data Replication: An SOA Architectural Antipattern

Let's begin with a detailed description of how data replication is an antipattern when applied in an SOA, using the antipattern template to explain the context specifics and forces that validate this architecture. An example illustrates the problems when applying this antipattern. In closing, you will see the refactored solution and the benefits that can be seen when the pattern is applied in place of the antipattern.

Context. This antipattern describes a common scenario that architects face when attempting to build enterprise SOA solutions in a distributed environment. In the move from the mainframe environment to the common distributed environment, a shift has occurred in the management of our important master data (the data that drives our enterprise on a daily basis). We decide to build our new distributed applications using a silo model, where an application is defined as a set of business functionality exposed through a user interface (UI), and that business functionality is built within its own context. This decision encourages architectures that favor extreme isolation and unnecessary entity aggregation.

"TO PROMOTE CONSISTENCY, THE SOFTWARE INDUSTRY HAS SETTLED ON A TEMPLATE FOR DOCUMENTING DESIGN PATTERNS. THIS TEMPLATE, HOWEVER, SEEMED INSUFFICIENT TO DESCRIBE ARCHITECTURAL ANTIPATTERNS AND THEIR EVENTUAL REFACTORED SOLUTIONS"

Forces. This antipattern occurs in an SOA because of the comfort that architects and designers have with this model. Even when it is explained as an antipattern within the context of SOA, architects and designers continue to use this antipattern for these reasons:

- *The architect is familiar with the data replication strategy.* The techniques used for replicating data have been fine tuned since

Table 1 Correlating sections with patterns and antipatterns

Section	Description	Pattern	Antipattern
Name	A useful short name that can help to quickly describe the pattern/antipattern.	X	X
Context	The interesting background information where the pattern/antipattern will be applied.	X	X
Forces	There are a set of reasons why this solution is being used. Very often in an antipattern these reasons are made up of misconceptions or lack of knowledge.	X	X
Solution	This section describes the proposed solution to the problem based on the context and forces documented and represents the strategy or intelligence of the pattern.	X	
Poor solution (Antipattern)	This section will describe the antipattern or solution to a problem that appears to have benefits but in fact will generate negative consequences when employed.		X
Consequences	Consequences are the side effects of the implemented solution. In the case of an antipattern, consequences are always going to be made up of negative impacts that outweigh the benefit of the pattern itself.	X	X
Refactored solution	Every antipattern should have an inverse that would in fact be a solution or pattern, which may be documented as patterns.		X
Benefits	The benefits will be the same for a documented pattern and the refactored solution of an antipattern.	X	X
Example	A real-world example of where the pattern/antipattern is applied to highlight/solve a problem.	X	X

the earliest days of batch-oriented development. This fine tuning creates a comfort level for the architect that has been designing systems with data replication for years, using either file-based transfers or extract, transform, and load (ETL tools). It seems simple enough to continue to use this strategy going forward.

- *The architect is concerned about the performance of a service exposed by the master data system.* For example, accessing a data store remotely through services may slow down the new system over a solution that uses a local database, which is the one-to-many data scenario.
- *Separate services may each have slightly different views of the entity.* Combining the different views into an aggregated entity and maintaining service autonomy is seen as too difficult, so the data from the services is replicated into a store, which is the many-to-one data scenario.
- *The architect is concerned that the new system's durability could be compromised.* For example, if the new system will access the master data through a service that exists on the network, network failures or master data system failures could cause downtime for the new system. Having the data available locally seems to mitigate this concern.
- *Applications are built using a silo model.* An application is defined as a set of business functionality exposed through a user interface, and that business functionality is built within its own context without an enterprise architect helping to guide the application developers toward the reuse of available services.
- *Resources for delivering a solution using data replication are abundant.* The skills required to implement a solution using data replication are typically easy to find. Building applications that leverage new or existing services takes a rarer skill set, requiring someone with a strong knowledge of distributed systems, Web services, object orientation, and other modern mechanisms for delivering solutions to your business.

Blurring Lines of Control

Poor Solution (Antipattern). To accomplish this antipattern, master-data information is replicated from the master-data source to a new database created to expose the functionality of the new application (see Figure 1). This exposure may be accomplished through many different mechanisms, such as the use of an ETL tool or through more basic mechanisms such as a file transfer of the data from one system to the other. Master-data information may also be augmented by the new systems, adding additional entity attributes to the original entity.

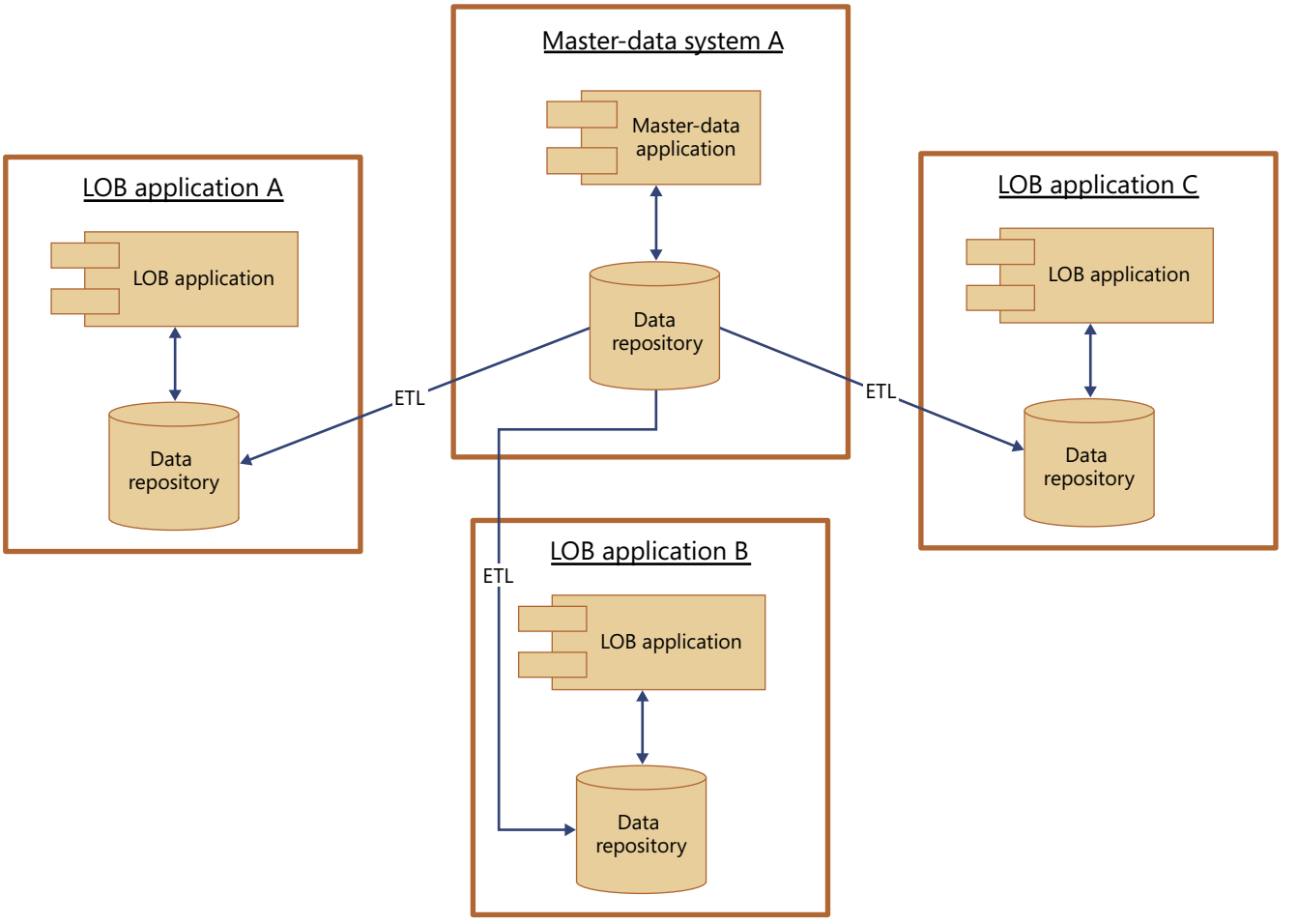
Consequences. The consequences of this pattern applied incorrectly within an SOA are not immediately visible. When data is replicated from the owning master-data system, in a fractal way, to many different application databases, your service orientation becomes very hard to manage. Replicated data becomes pervasive across the enterprise. Unclear responsibility for the data blurs the lines of control. Multiple services exist to manipulate the same (or slightly augmented) data. The "business view" of the data becomes disparate.

This unclear responsibility for the data and loss of control over the master data is disconcerting to any architect. As data is proliferated and applications begin to add their own *spin* to the data the complexity begins to mount. The new fields that are added to the data, which may be important to the enterprise as well, may not get added back to the master-data system. Of course, with the compressed timelines and budgets of most enterprise projects, a project team rarely will take on the extra effort of creating new data elements within the master-data system, exposing those new elements through services extended off of the master-data system, or creating an aggregation services layer (thus replacing the initial master-data service).

They may in fact extend services off of their system that will expose this new master data. Also, they may create new services that mimic some of the functionality of the master-data system to expose the data that they have replicated to their new system. This exposure creates a large headache for the enterprise architects to manage. The architect now needs to prevent new systems from consuming services exposed using replicated data, which leads to confusion within the enterprise architecture of the true ownership of the business object.

Replication also carries with it a huge responsibility on the replicating system to duplicate the business logic of the master-data system. Many times data is massaged by the master-data system before being exposed as a service to other applications. In these cases, the *quick-hit* method of replicating the data to the new system carries with it the burden of also replicating the business

Figure 1 Proliferation of master data leads to distributed ownership of various permutations.



logic used to massage the data. Unless good analysis is done on the master-data system, this logic may never get implemented, or may get implemented incorrectly. An even more detrimental scenario is one in which no business logic surrounding the retrieval of the master data initially exists. The logic is instead added later

> "THE BENEFITS OF AN SOA WILL ONLY BE FULLY REALIZED BY AN ORGANIZATION THAT IS COGNIZANT OF THE MIND-SET CHANGE REQUIRED TO DELIVER SOLUTIONS USING NEW ARCHITECTURAL PATTERNS"

in a project that needs to upgrade the master-data system. This scenario can be painful for an enterprise that has replicated data many times. Each system must now be analyzed for the changes required to the system to meet the new demands of the enterprise data.

Widespread Redundancies

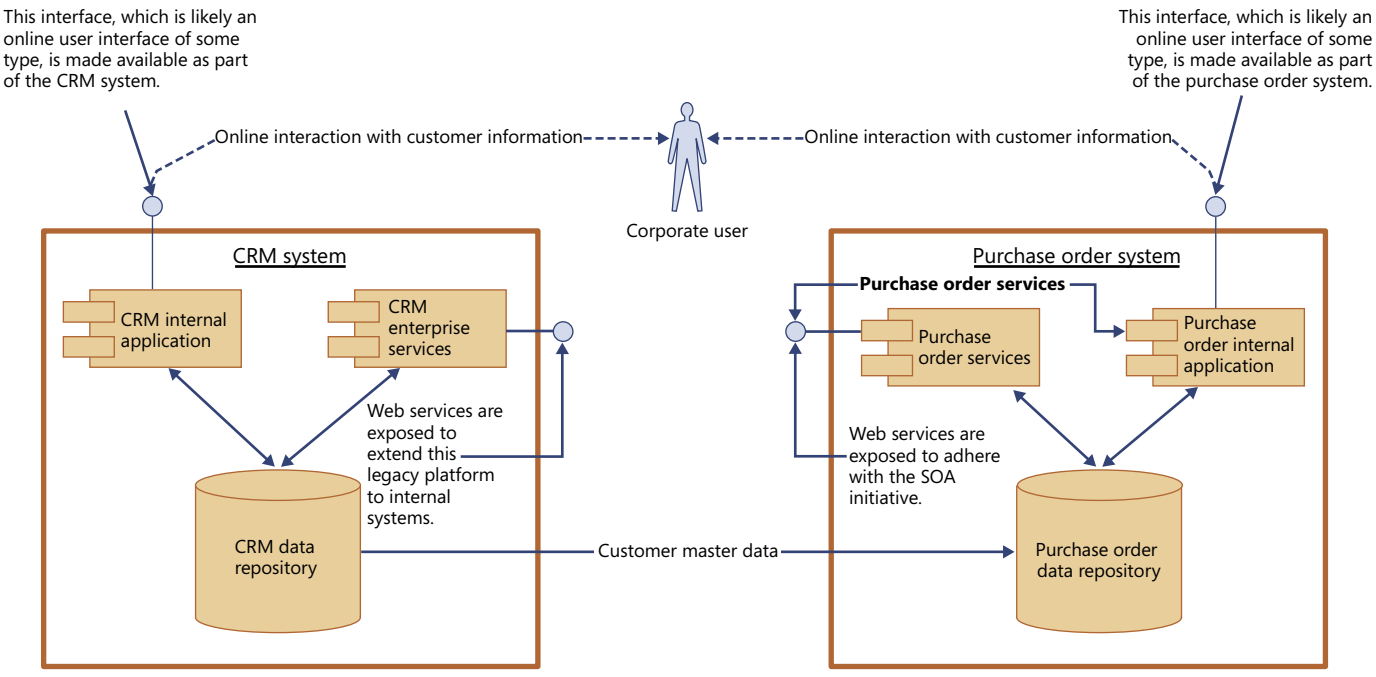
It is often easy to dismiss the consequences of using data replication because, on the surface, they seem superficial. Issues like disk space, reusable assets, and labor reduction seem to be manageable with or without moving toward service orientation. The consequence that overshadows them all is the ability to minimize complexity and deliver solutions that can quickly adapt to change. Data replication adds complexity, brittleness, and inflexibility because of the widespread redundancies it creates in your enterprise architecture.

Let's look at an example. A new application is being built to handle the management of a purchase order. The architecture demands that business services be created for the new system and that the UI, whether a portal or Web-based implementation, consumes those business services.

The business requirements dictate that the system should perform the functions of creation, modification, and viewing of purchase orders. Part of the purchase order is customer information, which is stored in a customer relationship management (CRM) system. The CRM system exposes services to retrieve customer information, but the decision is made by the architect to replicate the customer master data to the purchase order systems based on the forces discussed previously.

A requirement of the new purchase order system is to create an account number for each customer. This new data is attached to the replicated customer data in the PO system's local database. The UI

Figure 2 Data redundancy in the antipattern



displays customer information through the use of a new Web service created by the narrow vertical purchase order system. Other business services are developed for creation, modification, and viewing of purchase orders that are then consumed by the PO system's UI. Any data manipulation or massaging that the master-data system does when a user accesses its data services must be replicated into the new system. Any changes of that data going forward must also be replicated to the new system, as well as any changes to any business logic to access that data.

The diagram shown in Figure 2 makes it clear that the data replication results in duplicate services. Each application now has its own view of the customer entity, with its own way to access the data through a service that was created for that purpose. The purchase order system copies and extends the customer data through its own set of services, which effectively creates redundancy and confusion for any future systems looking to consume customer services.

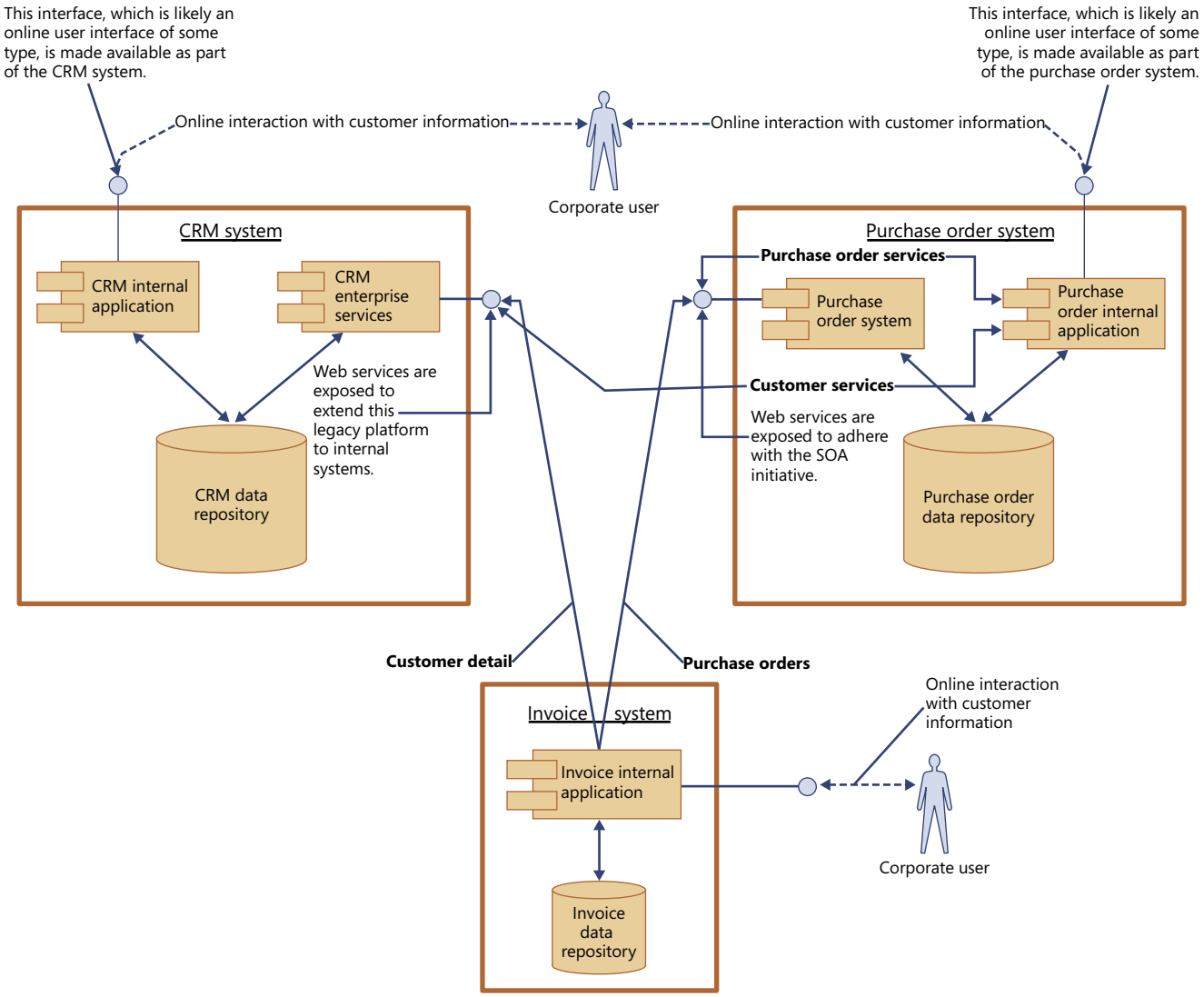
Now the next application comes along, a system that manages invoicing/accounting. This system also needs the customer data, including the account number. There are now several choices that the system designers have to make concerning customer data. Do they go get the customer data master information from the CRM system and get the customer account number from the PO system, or do they—since the data all exists in the PO system—just go to the PO system to get all of the customer information? In fact, because durability is a big concern (a force in this antipattern), a decision is made to replicate the customer data from the PO system to the invoicing system. Now three systems have a view of customer data in certain degrees of completeness. Three systems are forced to build any business logic around the access of that data. Each of them expose customer data through services (because we are building a "service-oriented architecture"), and the ability for the enterprise to manage customer data in an agile way continues to break down.

Cognizant of Change

Refactored solution. The benefits of an SOA will only be fully realized by an organization that is cognizant of the mind-set change required to deliver solutions using new architectural patterns. SOA will not deliver the kinds of benefits that are possible until architects realize that only by providing a single point of control over the business services and data can the organization truly become the responsive, flexible, and scalable enterprise that SOA promises. To successfully refactor the solution, you must address the forces that create the antipattern:

- *The architect is familiar with the data replication strategy.* Use the key performance indicators (KPI) that drive an SOA implementation to build a case to not replicate data unnecessarily. Training on SOA implementations that reduce data replication needs can also mitigate this force.
- *The architect is concerned about the performance of a service exposed by the master-data system.* Test to validate the performance of an exposed service. Through testing and an understanding of the service-level agreement (SLA) for the new application, many will find that the use of a new or existing service that exposes master data will perform within the SLA demands.
- *Separate services may each have slightly different views of the entity.* The skills and technologies to combine the different views into an aggregated entity and maintain service autonomy are already well established and can usually be done at run time.
- *The architect is concerned that the new system's durability could be compromised.* Address the durability of the exposed business

Figure 3 The refactored system



services through available infrastructure mechanisms such as clustering.

- *Applications are built using a silo model.* Many projects require the involvement of an architect with an eye toward new and existing services and the functionality they provide. In fact, applications as we know them today must morph from a simple box on a diagram to an assembly of new and existing services.
- *Resources for delivering a solution using data replication are abundant.* Currently available patterns, templates, standards, and tools allow true SOA with "mid-level" designers and developers.

Following a detailed analysis of these issues, you can decide your architectural strategy. The shift toward service orientation requires us to think differently about what are the critical questions when making these types of decisions. Instead of focusing so much time on the architecture of my "application" architects should focus on the architecture of the enterprise services. Services should be built on the system that maintains and manages the data, or aggregating services should be built to provide the data, eliminating the need to replicate the data to another application to provide a service.

The enterprise view of an application needs to fundamentally change. To make SOA work, we must shift from building applications—a stand-alone system that provides user functionality—to building products: an array of features to be delivered to the customer. We should be building services that provide this business functionality. The product is nothing more than a composition or orchestration of one or more services and provides one or more pieces of business functionality. It is the aggregation of solved use cases.

Eliminate Ambiguity

Once this mind-set becomes clear, and these forces have been mitigated, direct access to the line-of-business (LOB) services becomes a reality. The UIs of new systems can call existing services as needed, without any data indirection.

Following a refactoring of the architecture to direct services the ambiguity of data ownership is eliminated. New applications like the Invoice system (see Figure 3) are able to retrieve data from the master systems of record.

Benefits. Architects are constantly called on to evaluate the trade-offs of using a certain pattern within a certain solution. When the enterprise architectural environment attempts to move toward an SOA, for all of the reasons that SOA is an important architecture, the trade-offs of replicating data to enhance the speed of the application or the durability of the application must be evaluated against the benefits of the clarity and simplicity gained from the single-point ownership and governance that a service architecture can bring to your organization.

Additional benefits of using the direct services pattern can be found in the KPI that push us toward an SOA initiative in the first place. Service orientation is the latest attempt to create renewable application deliverables. Extreme cost savings can be found through the reuse of services, both in storage space and labor costs. While data replication can appear to satisfy the functional requirements, the goals of the enterprise architecture vision are not being met.

Another benefit can be found in the efficiencies gained by retrieving only the data you need. Architectures that leverage event-driven data retrieval are sure to be more efficient than antiquated batch-oriented processes. Replicating all of the data during a time-constrained batch window creates costly overhead. Without a mechanism for retrieving the right data at the right time using services,

> "ANOTHER BENEFIT CAN BE FOUND IN THE EFFICIENCIES GAINED BY RETRIEVING ONLY THE DATA YOU NEED. ARCHITECTURES THAT LEVERAGE EVENT-DRIVEN DATA RETRIEVAL ARE SURE TO BE MORE EFFICIENT THAN ANTIQUATED BATCH-ORIENTED PROCESSES"

you use additional processing cycles to move data that may have little or no use depending on the activities that depend on that data. The direct services pattern will ensure that you do not perform any wasted processing.

Ultimately, the goal of any enterprise data service is to give as complete a view of an entity as possible without sacrificing the benefits of the enterprise architecture initiatives. In cases where core pieces of that entity are fragmented across multiple systems, the design of the service may require some level of aggregation. However, this doesn't eliminate the option of using the direct services pattern to retrieve those subcomponents of our entity.

An architecture that effectively implements and consumes direct services will minimize the brittleness of the enterprise solution portfolio by eliminating redundancy. Through centralization and reuse, the services built using this pattern will provide the best possible approach for remaining agile in a climate of change.

Data Replication as a Pattern

All enterprises have complicated systems and requirements that require architects to remain pragmatic about the patterns they

Figure 4 Data movement building block

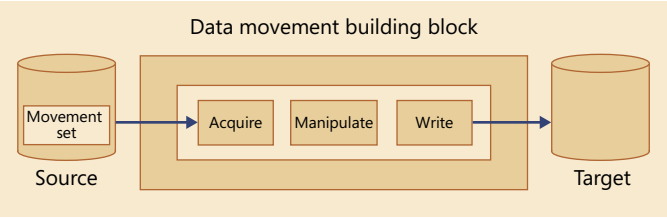
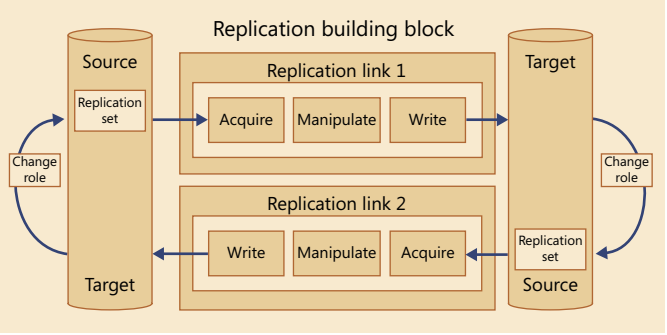


Figure 5 A master-master replication scenario



endorse. In these scenarios an architect must control the widespread adoption of replication patterns through good enterprise governance. If that doesn't happen, the enterprise is at risk of overusing this pattern. Because of its perceived low barrier of entry for developers, designers, and architects, the data replication strategy should only be considered a pattern in rare situations.

Context. The context where data replication is a pattern is slightly different than the context where it is an antipattern. It still stems from a distributed model where you wish to have a copy of the master data maintained in a system external to the master-data system. In some situations, you may not be able to avoid replication. Some examples include where network latency is a problem, such as in a WAN; where the master-data system is unreliable or has incompatible maintenance windows; when offline capabilities are a key requirement of the application; and when the risks of not having services is mitigated through the expected usage of the application.

Forces. The forces that influence the decision to make a copy of the master data and may justify the complexity involved in duplicating the data are:

- *Data availability on the master-data system doesn't match the requirements for the new system.* For example, the master-data system must be taken down for maintenance between certain hours where the new system must be available (and the data is critical to the use cases delivered by the new system).
- *The network is unreliable or too slow.* For example, on a WAN where the network consistently is losing connection to the master-data system, copying the data may alleviate this problem. Another scenario is that the network is too slow to support direct consumption of the data when required.
- *The system is expected to have offline capabilities.* This force is often a requirement found in systems that will have very little

or no connectivity to the enterprise services that are providing the data.

- *The data will be copied without any reusable business logic for the purpose of analysis only.* There are scenarios where data would need to be copied to a data warehouse for use in analytical reports. In most cases, this type of data transfer is most effective when an ETL tool is used to simply copy the data once it is deemed ready for archiving.

Solution. One solution to these forces is data replication. Data replication can be performed using an architectural building block known as a *data movement building block*. The data movement building block consists of a source, movement link, and destination.

The diagram shown in Figure 4 displays the basic building block of many other data movement patterns (see Resources). Depending on the solution, you may need multiple data movement building blocks to handle a master-master scenario, where the data can be updated in both the source and destination (see Figure 5).

Benefits. Much of the work described in the patterns discussed previously can be done with tools and nondevelopment resources. This feature is one of the primary benefits of this approach because you will often see a lower initial total cost of ownership (TCO). This benefit doesn't tell the whole story though. Often the long-term maintenance and versioning of these types of tools and strategies are very costly. If the primary focus is time to market and the road map for the solution is relatively short, then this may be a viable pattern with a high degree of upside. Keep in mind you will trade brittleness and possible refactoring costs down the road. Sometimes such refactoring can be done by wrapping a legacy system that doesn't meet the tenets of SOA with an interface that is SOA compliant. In this way, the enterprise architecture can move toward SOA as a whole, without having to refactor every application in its domain.

Document Best Practices

An unavoidable aspect of innovation in technology is change. Application architects must remain focused on mitigating the risk of change. Using patterns and antipatterns to document best practices has proven to be successful since the earliest OO patterns. When applied in a timely fashion, architectural patterns are the most viable weapon in the battle against change.

Resources

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, et al. (Addison-Wesley Professional, 1995)

Microsoft Developer Network: "Data Patterns – Microsoft Patterns & Practices," Philip Teale, Christopher Etz, Michael Kiel, and Carsten Zeitz (Microsoft Corporation, 2003)

"Principles of Service Design: Service Patterns and Anti-Patterns" (Microsoft Corporation, 2005)

"SOA Antipatterns" (IBM, November, 2005)

Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Jack Greenfield, et al. (Wiley & Sons, 2004)

The pattern and antipattern discussed here focused on data replication and its role in an enterprise with an SOA initiative. By using a template for pattern description, you can see why this technique can have a positive or negative impact depending on the context in which it is applied. Using patterns to solve this architecture mismatch illustrates exactly how successful adopting an architecture-driven methodology can be.

As the software industry continues to evolve, the role of architecture and patterns will become even more critical. Patterns like data replication that are common within current architectures may now be detrimental to your enterprise's architectural progression. Organizations that can see the shift from applications to services will be the ones who will make the move much more quickly and see the full potential of SOA.

"LONG-TERM MAINTENANCE AND VERSIONING OF THESE TYPES OF TOOLS AND STRATEGIES ARE VERY COSTLY. IF THE PRIMARY FOCUS IS TIME TO MARKET AND THE ROAD MAP FOR THE SOLUTION IS RELATIVELY SHORT, THEN THIS VIABLE PATTERN MAY HAVE A HIGH DEGREE OF UPSIDE."

The seemingly unattainable goals of software industrialization are quickly becoming a reality. Patterns are the building blocks of consistency and agility in enterprise architecture. Treating them as first-class architecture artifacts will help to accelerate their discovery and adoption. In time, an enterprise that creates a reusable library of these patterns will have increased responsiveness to their business sponsors and gained an edge over any competition. •

About the Authors

Tom Fuller is CTO and senior SOA consultant at Blue Arch Solutions Inc. (www.BlueArchSolutions.com), an architectural consulting, training, and solution provider in Tampa, Florida. Tom was recently given the Microsoft Most Valuable Professional (MVP) award in the Visual Developer ñ Solution Architect discipline. He is also the current president of the Tampa Bay chapter of the International Association of Software Architects (IASA), holds a MCSD.NET certification, and manages a community site dedicated to SOA, Web services, and Windows Communication Foundation (formerly "Indigo"). He has a series of articles published recently on the Active Software Professional Alliance and in SQL Server Standard magazine. Tom speaks at many of the user groups in the southeastern U.S. Visit www.SOApitstop.com for more information, and contact Tom at tom.fuller@soapitstop.com.

Shawn Morgan is CEO and senior architect at Blue Arch Solutions, Inc. (www.BlueArchSolutions.com). Shawn has architected solutions for many Fortune 500 companies including Federal Express, Beverly Enterprises, and Publix Super Markets. Shawn focuses on enabling enterprises to deliver IT solutions through architecture. Contact Shawn at shawn.morgan@bluearchsolutions.com.

Patterns for High-Integrity Data Consumption and Composition

by Dion Hinchcliffe

Summary

The challenge is to consume and manage data from multiple underlying sources in different formats while participating in a federated information ecosystem and while maintaining integrity and loose coupling with good performance. Here are some emerging patterns for the growing world of mashups and composition applications.

These days the field of software development is just as much about assembly and composition of preexisting services and data as it is about creating brand new functionality. The world of mashups on the Web and composite applications in the world of service-oriented architecture (SOA) are demanding that data—in very different forms and from virtually any source—be pushed together, interact cleanly, and then go about their separate ways. And these approaches demand that this interaction happens and occurs efficiently, all without losing fidelity or integrity.

The considerable variety of data these days includes an extensive array of XML-based formats, as well as increasingly widespread, lighter weight data formats such as the JavaScript Object Notation (JSON) and microformats. Applications also have to increasingly work with rich forms of data such as images, audio, and video, and we can't forget the older, everyday workhorse formats such as text, EDI, and even native objects. All of these formats are increasingly being woven and blended together at the same time systems become more interconnected and integrated into vast supply chains, enterprise service buses (ESBs), SOAs, and Web mashups. Maintaining order in such data chaos is more important than ever. The good news is that first-order rules for handling all of this data heterogeneity are beginning to emerge.

Web services, SOA, and especially the Web itself are made of open standards that make it possible for systems to speak together—the ubiquitous and essential HTTP being a prime example. However, this communication does not allow one to assume what sort of data your software will have to consume in the applications of the future. While it is still a good bet that you will probably be facing some form of XML, it's now just as likely you will be faced with one of the lightweight data formats growing in popularity such as JSON. However, increasingly, it could just as easily be simple, plain text; a tree of native objects; or even a deeply-layered, WS-* style Web services stack that will be provided by the forthcoming Windows Communication Foundation (WCF).

Developers are therefore faced with serious challenges in terms of creating good data modeling, architecture, and integration techniques that can encompass this diversity. The heterogeneity of data representation forms can be quite daunting in environments with high levels of system integration. Never mind that in very loosely coupled, highly federated systems, as many applications are increasingly becoming, the likelihood of frequent change is high. Consequently, it is up to the application development community to create a body of knowledge on best practices for the low-barrier consumption of data from many different sources and formats, with matching techniques for integrating, merging, and relating them. The community also needs to ensure that integrity is maintained while remaining highly resilient to change and even the inevitable evolution of the underlying data formats.

While the patterns presented here are not authoritative and intended primarily to be guidance, they are based on the well-accepted concept of the interface contracts. Interface contracts are now common in the Web services world with WSDL as well as in many programming languages but particularly in design by contract. In an interface contract, a provider and a supplier come together and agree on a set of interactions, usually described in terms of methods or services. These interactions will cause data of interest to pass back and forth across the interface boundary.

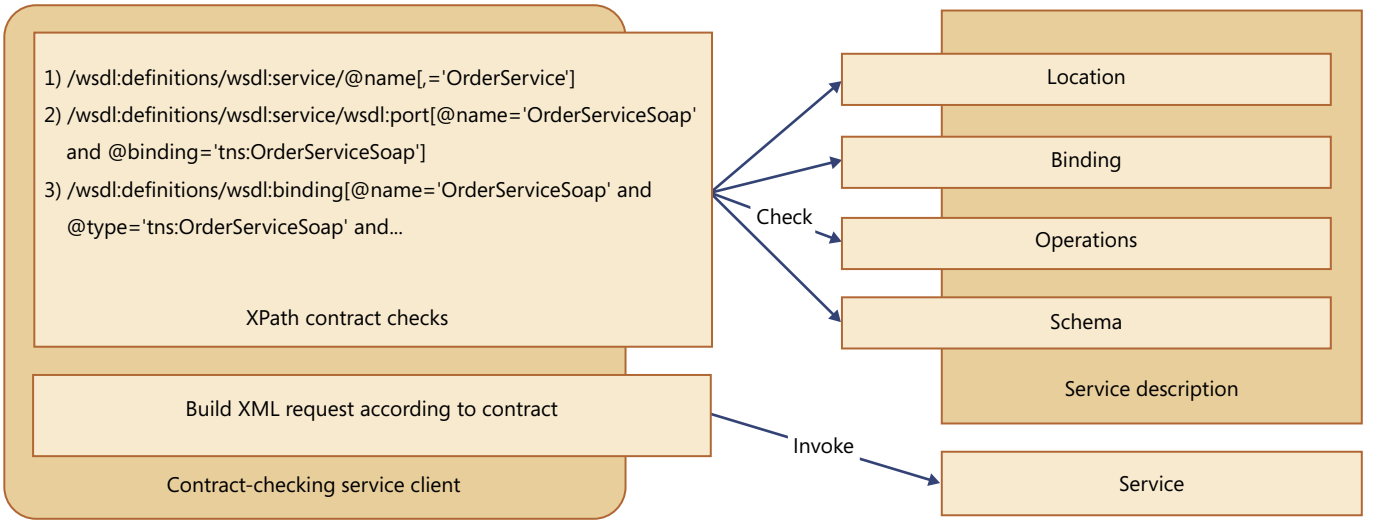
The exact definition of the interactions, including their locations and protocols, are precisely defined in the contract, which is preferably machine readable. Of particular interest are the data structures that are passed back and forth between supplier and client as parameters during each interaction. And it is these data structures that are really the center of the mashup/composite application model. Because it is here, when these highly diverse data structures meet, that we need to put principles of data integration and recombination into the sharpest relief.

Forces and Constraints of Consumption and Composition

Along these lines then, we can get a general sense of the forces that are shaping data consumption and composition in modern distributed software systems. In rough order, these are:

The interface contract as driver of data consumption and composition. When using data structures from any external source or service, such as a Web service, data structures must be validated against the interface contract provided by the source service. This validation can

Figure 1 Schemas embedded in an interface contract are usually the largest client dependency.



often be done at design time for data sources that are known to be highly stable and reliable. But in federated systems, especially those not under the direct control of the consumer, careful consideration must be given to doing contract checking at run time. Run-time contract checking in loosely coupled systems is an emerging technique, and some interesting options are at the disposal of the service consumer to avoid problems up front. The bottom line is that the interface contract is the principal artifact that drives data consumption and composition.

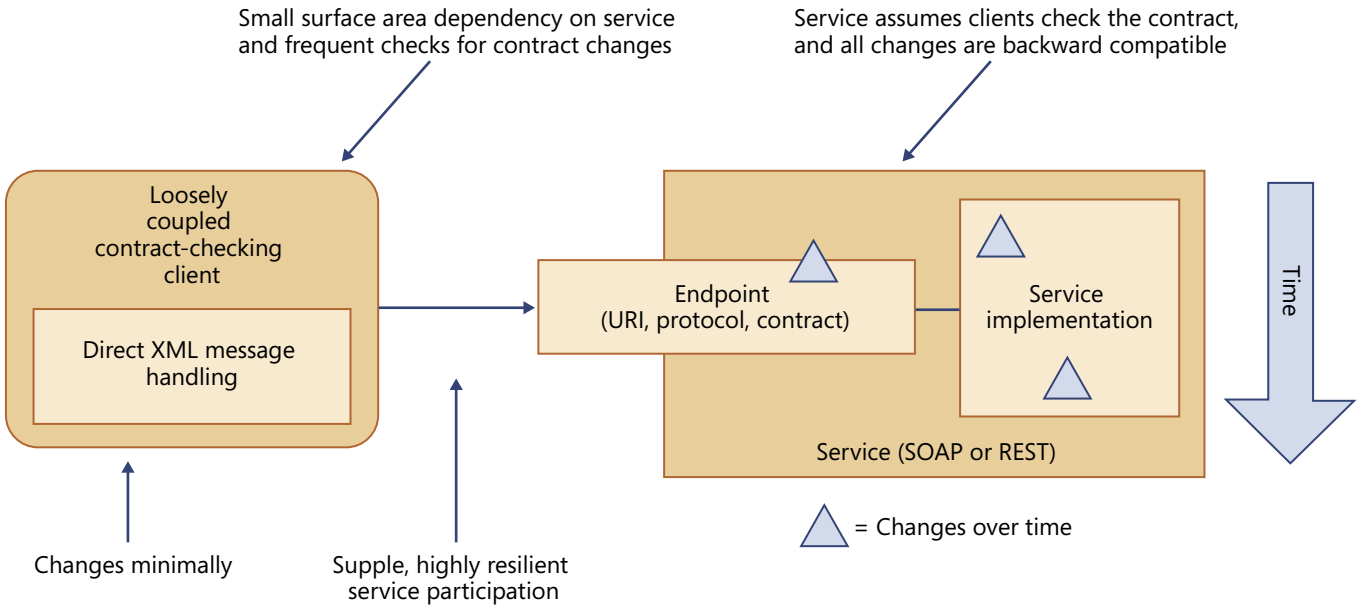
Abstraction impedance disrupts data consumption and composition. The physical point at which data integration occurs is increasingly outside the classic control of databases, and data-access libraries convert everything to a unified data abstraction.

> "WHEN USING DATA STRUCTURES FROM ANY EXTERNAL SOURCE OR SERVICE, SUCH AS A WEB SERVICE, DATA STRUCTURES MUST BE VALIDATED AGAINST THE INTERFACE CONTRACT PROVIDED BY THE SOURCE SERVICE"

Data retrieved from multiple external services and in multiple forms can and do collide within the browser, inside front-end clients, or on the server side's inside code and outside the database. The data structures from these underlying services, depending on the software application, range from native objects, XML, JSON, and SOAP document/fragments to text, binary, and multimedia data. Though abstraction impedance has been a well-known problem in software for a long time, it has only been exacerbated by the proliferation of the growing number of available models for representing information. When consuming and compositing these data structures into usable aggregate representations and then back out to their source services, a number of major problems appear.

- *Data format variability*. The fundamental formats of the underlying data structures are often highly incompatible, and good local facilities for processing all the forms that might be encountered are not usually at hand. Particularly problematic are extremely complex formats that require sophisticated protocol stacks to handle, such as higher-order Web services with WS-Policy requirements.
- *Data identity*. Keys and unique object/data identifiers often are not in common formats and are not the facilities available for checking and enforcing them consistently across abstractions.
- *Data relationships*. Establishing relationships and associations among data structures of highly differing formats can be problematic, both for efficiency and performance, for a number of reasons. Converting all data to a common format can be expensive, both in terms of maintenance and run-time cost. It also introduces the problem of maintaining the provenance of blended data and extracting it back out as it's modified. Keeping data structures in native forms assumes good mechanisms for compositing, relating, and manipulating them.
- *Data provenance*. Maintaining the originating source service of a chunk of data is essential for maintaining valid context for ongoing conversations with the source and for changes to the data in particular, such as updates, additions, and deletions of the data.
- *Data integrity*. Modifying source data, ensuring that changes are valid according to the underlying service's interface contract, and not violating schema definitions are important. With well-defined XML documents that are supported by rich XML schemas, these validations are less of an issue. However, lighter weight programming models such as JSON and others like it often do not have any machine-readable schemas. These types of data formats, though often only available from a federated service, are at the highest risk of integrity problems since the rules for changing and otherwise manipulating them are not as well defined as the other formats.
- *Data conversion*. Often, there are no well-specified conversions between data formats, or, even worse, multiple choices present themselves that alter the data in subtle ways. The most frequently affected data are numeric representations, but these conversions can also affect a data source in multiple character sets, audio/video data, and GIS data almost as often.

Figure 2 Run-time checking of the interface contract on Web services



Large interface contract surface areas. Extremely complex schemas and interface structures are growing into one of the leading problems in distributed systems data architecture. Experience has shown time and again that *simpler* is more reliable and of higher quality. Yet distributed services interface contracts often have large and complex XML schemas and WSDL definitions. The likelihood that changes will occur, inadvertently or not, increases with the complexity of an interface contract and its embedded schemas. While simple XML and REST approaches to services tend to encourage the desired simplicity, sometimes certain application domains are not "simple," and these service models are subject to additional problems because of the lack of interface contract standards. The bottom line is that the likelihood that changes to the contract without a remote supplier being aware of the changes increases with the size of the overall contract, because changes to large schemas are more likely to go undetected than changes to a smaller schema.

Implicit version changes. Even in well-controlled environments, changes to services and their interface contracts can happen without notifying all dependent parties. Barring changes to the way data services work currently, this pattern will only become more common in the highly federated environments of the present and those of the future. Without any change, the services you depend on will change without your knowledge or prior notification, and you will have to accept this fact as practical, inevitable, and unfortunate. Consequently, developing conscious strategies to detect version changes and handle them effectively is essential to maintain the quality of a mashup or composite application's data and functionality.

Patterns of Data Consumption and Composition

The patterns described here are intended to present a lightweight stance on data modeling and application architecture. These observations are a result of observing the dynamic world of Web mashups and composition applications that have been thriving on the Web for several years now. The nimble, minimalist approaches used by mashups in particular are an inspiration but in themselves often are not sufficient to create quality software. These patterns are intended to capture the spirit of mashups, put them in context with good software engineering practices, and to start a discussion and debate in the software community about these lean methods for connecting services and data.

Pattern 1 – Minimal surface area dependency on interface contract. This pattern is the front-end aspect of the well-known software design maxim: "Be liberal in what you accept, conservative in what you send." Most Web service toolkits, relational database frameworks, and multimedia libraries encourage dependencies on too much of the interface contract, often the entire contract, regardless of what the software depends on. For a great many of consumption and composition scenarios, a small surface area dependency is really all that is necessary. A full contract dependency introduces a tremendous amount of brittleness because changes that have no relation to the data in the data structure that is depended on will still break the client. While some data consumption toolkits are already forgiving of this dependency, there is often little control over what parts the toolkits care about.

For many applications, only direct dependencies of the needed internal data elements are appropriate. All other dependencies should be actively elided from the dependency relationship within the source data. The upshot here is that changes to the interface contract that are not of importance to the data consumer must *not* prevent the use of the data. The converse must be true as well; changes to the contract that matter must immediately become apparent to prevent incorrect behavior and use of the data. Examples of minimal dependencies include: XML, key paths through the schemas to data elements; relational data, only the tables, types, columns, and indexes used by the client; and multimedia, only the portions of the media structure that are required such as the specific channels, bit rates, image portion, video segment, and so on.

Pattern 2 – Run-time contract checking for changes. The services that mashups and composite applications depend on for data are subject to unexpected change at any time. This change could be because of moving the endpoint to a new location, stopping use of a previously supported protocol, changes to underlying schema, or even just outright deliberate interface changes or internal service failures. The reason leads to the necessity of checking the contract to detect changes to it. It also requires being able to handle many contract changes gracefully since they may very well not affect the part of the contract you care about.

In reality, there are two types of run-time contract check. One is to *check the contract specification itself*, if one exists. This specification is often the WSDL or other metadata that is officially published in conjunction with the service itself. These can be checked easily and quickly with a variety of programmatic techniques, including XPath queries for XML-based schemas or other relatively lightweight techniques. Hard-coded contract checks in traditional programming languages are not as easy to implement or change and should be a second choice.

The second contract check is to do validation of the contract against the instances of data that the service provides. Interface contracts often have their own abstraction impedance with their delivery mechanisms, and it can be surprising how often inconsistencies will creep in between the data provided and the interface contract, even with otherwise high-quality toolkits.

The biggest dilemma presented by this pattern is how often to check the "check the contract" itself. Checking the contract and the instance data with each retrieval of data from its source can be time-consuming and is usually unnecessary. Ultimately, determining the frequency of contract checking is largely dependent on the application requirements and the nature of the data and its tolerance for inaccuracy and incorrect behavior. For many applications, checking synchronously may not even be an option, and it may make sense to set up an appropriately periodic background process to identify contract changes, which will inevitably occur.

Pattern 3 – Reducing structures to a common data abstraction format. In reality, the carnival of data structures and formats that mashups and composite applications have to work with will only continue to grow. Software can either manipulate them in their native data formats, which means losing opportunities to maintain relationships and enforce business rules, or it can convert all of them to a common abstraction. This conversion is the approach that data libraries like ADO.NET use with its DataSets and XML does with other non-XML data sources. Subsuming the differences in source data by converting them into and out of a common data abstraction that provides a single unified model to work with can be appealing for a number of reasons. First, relationships between the various underlying data sources can be checked and enforced as the data is manipulated and changed. Second, views into the data can take advantage of the abstraction mechanism making it less necessary to build custom Model-View-Controller (MVC) mechanisms and use existing libraries and frameworks that can process the abstraction.

Not all sets of heterogeneously formatted data structures are a good candidate for this pattern, and it may make little sense for some data formats that have dramatic levels of abstraction impedance—image data with non-image data, for example. There is also a potentially nontrivial cost for data transformation and conversion into and out of the common format. For many applications, however, this cost is entirely acceptable.

Within certain data-consumption environments, like the Web browser, there are limited choices for maintaining a common data format, and JSON and the XML DOM tend to be quite popular for browser-based solutions. On the server side the choices are far richer but are often platform dependent. XML structures, O/R libraries like Hibernate, relational databases, and even native object graphs often make excellent common abstraction models depending on the application. But the very different nature of hierarchical data like XML and object graphs is one of the classic impedance problems in computer science, and care must be taken when using this pattern.

> "THE WEB ITSELF IS BECOMING THE MOST IMPORTANT SUPPLIER OF HIGHLY FEDERATED DATA, A SOURCE THAT WILL ONLY GROW AND BECOME MORE IMPORTANT IN THE NEXT FEW YEARS"

The bottom line: if performance is not absolutely critical and the underlying data formats and schemas are amenable, this pattern can be very powerful for working with federated data sources. The disadvantage is that the common abstraction approach can certainly involve more maintenance and take on brittleness because mapping and metadata must be maintained.

Pattern 4 – Native structures mediated with Model-View-Controller (MVC). Converting all source data into a common format will not be an option in many situations because 1) the processing overhead is excessive since much of the data might not be used, or 2) duplicating it in the local environment might be prohibitive in terms of resources. Or it could be because there is no common format that makes sense for all the underlying datatypes. In this case, creating an MVC that mediates the access, translation, and data integrity with the underlying native structures can provide the best options for both performance and data storage.

MVC is a powerful design pattern in its own right that has been proven time and again as an excellent strategy for the separation of the concerns in application software. Its use is particularly appropriate when there are multiple underlying data models in a given application. Good software design dictates that offering a unified view of the source data provides a single, clean, consistent interface that makes it easy to view, interact with, and modify the underlying data. Access to the underlying data with the MVC approach is also relatively efficient since only the data needed must be processed to satisfy most requests.

While there are many advantages with MVC, however, the disadvantages are similar to pattern 3 in that the maintenance of the MVC code can be enormous. Certainly, there are an increasing number of off-the-shelf libraries that can help software designers build MVC on both the client and the server. Be warned, though: mapping code is brittle and tedious.

Pattern 5 – Direct native data structure access. For many applications, especially simpler, browser-based ones, converting source data into common formats or building sophisticated MVC architec-

Table 1 Common data abstractions

Abstraction	Contract standard	Advantages	Disadvantages
Text, JSON, binary	Informal, textual	Relatively efficient	Not self-describing, not ideally efficient
Native objects	Class definition	Unified behavior and data, encapsulation, high-level abstraction, and composition	Requires conversion of most data into objects, requiring a mapping technique
XML	XML schemas (XSD), Relax NG, WSDL, and many others	Self-describing, rich schema description, and extensible without breaking backward compatibility	Very size inefficient, no way to distribute behavior, and schema descriptions are limited even with XSD
Images	Specifications for JPEG, TIFF, GIF, BMP, PNG, and many others	N/A	N/A
Audio	Specifications for WAV, WMA, MP3, and AAC	N/A	N/A
Video	Specifications for AVI, QuickTime, MPEG, and WMF	N/A	N/A

tures just isn't a good option. Direct access to the data makes more sense, and the decision is often a commonsense one, having to do with the libraries at hand. Plus, as mentioned before, *simpler* is often higher quality and better because there is less to break or maintain.

In this pattern, which works best with less highly structured data, the native data structures are used directly without an intermediary or any data conversion, which means data stores in text, XML, JSON, and so on are manipulated natively. The drawback, of course, is that you cannot rely on the facilities that data abstraction libraries can give you to enforce integrity or track changes. However, this pattern often requires the least amount of processing or learning third-party libraries. It can be easy to develop, and because there is no conversion or data access layers to go through, it's also quite fast.

Pattern 6 – All data modification is atomic. More sophisticated views of data and services prescribe properties known as ACID, for atomicity, concurrency, isolation, and durability. Usually ascribed to database systems, ACID is an excellent and practical rule of thumb for almost any concurrent data access system. Unfortunately, almost no one yet in the world of Web services and mashups has the notion of transactions in their protocols, which would confer many of the properties of ACID to the modification of data. Far from ignoring the problem, mashup and composite application developers must be acutely aware of working without a tightrope as they work with their data.

This awareness introduces significant issues with complex data modification scenarios, one being that any data retrieval and storage that is dependent on an extended conversation with underlying services will almost certainly not be protected by the same transaction boundary and related ACID properties. Software code must expect that data integrity problems will occur, especially in an extended conversation that may never finish to completion or fail partway through. Avoiding long-running conversations altogether is one option. Making software operation dependent, as much as is possible, in individual atomic interaction with underlying services is another good way. Each step in the interaction is a discrete, visible success that allows the software to offer its users clear options when a conversation with a supplier data service fails. These two options allow software developers to respect the ACID rule of thumb.

A Return to Simplicity

The Web itself is becoming the most important supplier of highly federated data, a source that will only grow and become more important in the next few years. While XML and lightweight data formats will likely be the dominant data structure that most software will have to work with in the foreseeable future, curve balls are en route. These curve balls include much needed optimizations in XML such as binary XML, advances in microformats, new multimedia codecs that will suddenly change an entire audio/visual landscape, and completely new transport protocols such as Bittorrent, which will make many of the patterns here problematic, to say the least.

A return to simplicity in data design is back in vogue, exemplified by the rise in the interest in formats like JSON, microformats, and dynamic languages like PHP and Ruby. This design simplicity can make data more malleable and easier to connect together. It also allows for less difficulty in writing software to care for and manage it. While the patterns presented here are overarching ones that can lead to less brittle, more loosely coupled, and high-integrity data consumption and composition, the story is really just beginning. As the Web becomes less about visual Web pages and more about services and pure data and content, becoming adept at being a nimble consumer and supplier of the information ecosystem will be an increasingly critical success factor. •

Resources

microformats
http://microformats.org

Wikipedia
http://en.wikipedia.org/wiki/Design_by_contract

"The Impedance Mismatch Between Conceptual Models and Implementation Environments," Scott N. Woodfield, Computer Science Department, Brigham Young Univerisity (ER'97 and Scott N. Woodfield, 1997)
http://osm7.cs.byu.edu/ER97/workshop4/sw.html

Hewlett-Packard Development Company
Technical Reports
"Rethinking the Java SOAP Stack," Steve Loughran and Edmund Smith
www.hpl.hp.com/techreports/2005/HPL-2005-83.html

About the Author

Dion Hinchcliffe is chief technology officer at Sphere of Influence Inc.

The Nordic Object/Relational Database Design

by Paul Nielsen

Summary

The New Object/Relational Database Design (Nordic), like many tools, is not the most fitting or expedient solution for every kind of database problem. However, the object/relational hybrid model can provide more power, greater flexibility, better performance, and even higher data integrity than traditional relational models, particularly for databases that benefit from inheritance, creative data mining, flexible class interactions, or workflow constraints. Discover some of the innovations that are possible when object-oriented technology is modeled using today's mature relational databases.

The differences between object-oriented development and the relational database model create a tension often called the *object-relational impedance mismatch*. Inheritance just does not translate well into a relational schema. The technical impedance mismatch is aggravated by the cultural disconnect between application coders and database administrators (DBAs). Often neither side fully understands nor respects the other's lexicon. A sure way to get under a DBA's skin is to refer to the database as the "object persistence utility." That relationship is unfortunate because each side brings a unique set of advantages to the data architecture problem.

In many ways object-oriented design is superior to the relational model. For example, designing inherence between classes can be accomplished using the relational supertype/subtype pattern, but the object-oriented design is a more elegant solution. Also, nearly all application code is object oriented and an object-oriented database would interface with the application more easily than would a relational database.

As sophisticated as object-oriented technology is at modeling reality, the relational side is not without significant advantages. Relational database engines offer performance, scalability, and high-availability options, and the financial muscle to ensure the database platform will still be here in a few decades. Relational technology is well understood, and relational databases offer more powerful query and reporting tools than object databases. The few pure object database companies available simply do not have the resources to compete with Microsoft, Oracle, or IBM.

The conundrum of the object-relational impedance mismatch then is how to best embrace the elegance of object-oriented technologies while retaining the power, flexibility, and long-term stability of a mature relational database engine. Since application programmers are generally the most interested in solving this problem, and human nature tends to solve problems using the most comfortable skill set, it is not surprising that most solutions are implemented in a mapping layer between the database and the application code that translates objects into relational tables for persisting objects.

The Nordic Proposal

I propose that a relational model, designed to emulate object-oriented features, can perform extremely well within today's relational database engines, and that manipulating the class inheritance and complex associations directly in the database, close to the data, is in fact very efficient. This efficiency was not always the case. I was hired to optimize a Transact-SQL (T-SQL)-intensive, object/relational (O/R) database design implemented with SQL Server 6.5 and failed. Development of the Nordic Object/Relational Database Design involved a year's worth of iterations, a simplified metadata schema, and SQL Server's maturing T-SQL.

"AS SOPHISTICATED AS OBJECT-ORIENTED TECHNOLOGY IS AT MODELING REALITY, THE RELATIONAL SIDE IS NOT WITHOUT SIGNIFICANT ADVANTAGES"

As with any database project, a strictly enforced data abstraction layer that encapsulates the database is necessary for long-term extensibility. For an O/R hybrid database, the data abstraction layer also provides the façade for the object-oriented features. Behind the façade's code are the metadata schema and code generation for the classes, objects, and associations (see Figure 1). When implementing this solution there are a few key design decisions.

Class management. Within the relational schema, class and attribute metadata are easily modeled using a common one-to-many relationship. The superclass/subclass relationship is modeled as a hierarchical tree, using either the more common adjacency list pattern or the more efficient materialized path pattern.

Navigating up and down the class hierarchy with user-defined functions enables SQL queries to join easily with any class's decedent or ancestor classes. These user-defined functions are lever-

aged throughout the façade layer. For instance, when selecting all the properties for a class, joining with the superclasses() user-defined function returns all superclasses, and the query can then select all of a given class's properties including properties inherited from superclasses.

In the context of working with persisted objects, *polymorphism* refers to the select method's ability to retrieve not only the current class's objects but all subclass objects as well. For example, selecting all contacts should select not only objects of the contact class but also objects of the customer and major customer subclasses. A user-defined function that returns a table variable of all subclasses of a given class makes writing this query and stored procedure efficient and reusable.

Object management. Objects are best modeled using a single object table that stores the object's common data such as the unique objectid, object's class, audit data, and a few search attributes common to nearly every class, like name, a date attribute, and so forth. Additional attributes are stored in custom class tables that use an objectid foreign key to relate the custom attributes to the object table. The createclass and other class-management, façade-stored procedures execute the Data Definition Language (DDL) code to create or modify the custom class tables and generate the custom façade code for selecting, inserting, and updating objects.

The major design decision for modeling how objects are stored is how to represent the custom attribute data. There are three possible methods: the value-pair pattern, concrete custom class tables, and cascading custom class tables. The *value-pair pattern*, also called the *generic pattern*, uses a diamond-shaped pattern consisting of class, property, object, and value. The value table uses a single column to store all values. This long, narrow table uses one row for every attribute. Ten million objects with fifteen attributes would use 150 million rows in the value table. SQL Server is more than capable of working with large tables—that's not a problem. This model appears to offer the most flexibility because attributes can be added without modifying the relational schema; however, this model suffers from nonexistent, or at best awkward, data typing and is difficult to query using SQL.

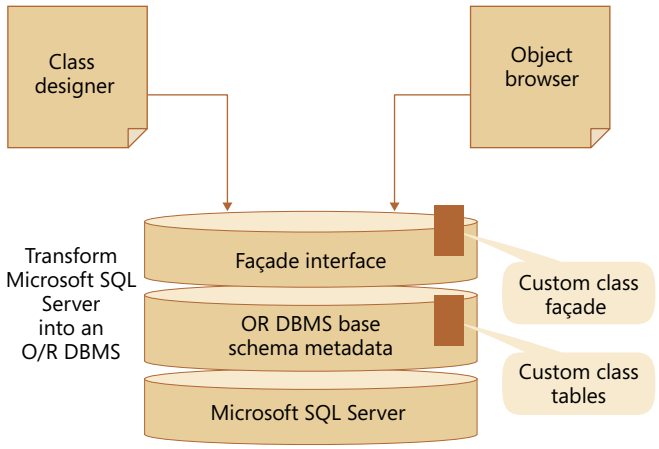
A Table for Every Class

The *concrete custom class* model uses a table for each class with columns for every custom attribute including nonabstract inherited attributes. An object exists in only two tables—the object table and the concrete custom class table—while attributes are replicated in every subclass's concrete custom class tables. Therefore, if the animal class has a birthdate attribute, and the mammal subclass has a gender attribute (since some animals do not have a gender), then the mammal custom class table includes objectid, birthdate, and gender columns.

This pattern has the advantage that selecting all attributes for a given class requires, only joining the object metadata table with a single custom class table. The disadvantage is implementing polymorphism; selecting all animals requires performing a union of every subclass and either eliminating subclass attributes or adding surrogate superclass attributes so all selects in the union have compatible columns.

An improvement over the value-pair pattern, the concrete custom class tables use a relational column for each attribute so attribute data typing can easily implement the native data types of the host relational database.

Figure 1 The O/R hybrid design uses a façade to encapsulate the object-oriented functionality executed within a relational database schema.



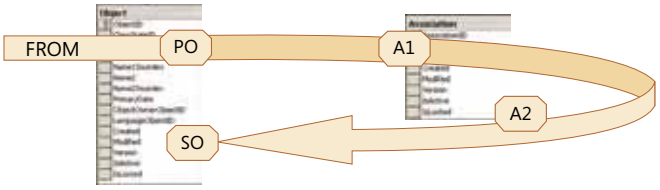
The third option implemented in the Nordic Object/Relational Database Design, *cascading custom class* tables, uses a table for each class like the concrete class solution. However, instead of replicating attributes, each attribute is represented only once in its own class, and every object is represented once in every cascading class. Using the mammal and animal example, the animal table contains objectid and birthdate, and the mammal table consists of objectid and gender. An instance of the mammal object is stored in the object metadata, the animal table, and the mammal table. Polymorphism is very easy in this option, however, more joins are required to select all attributes from subclasses. Since SQL Server is optimized for joins, cascading custom class tables perform very well. As with concrete custom class tables, strong data typing is supported by the host database.

Association management. Object-oriented technology's associations are very similar to relational database technology's foreign key constraints, and it is certainly possible to define associations within an O/R hybrid model by adding foreign key attributes and assigning relational declarative referential integrity constraints to the custom class tables.

However, storing every object in a single table offers some exciting alternatives for modeling associations. Where a normalized relational database might contain dozens of foreign keys, each with a different foreign key table and column, and each references a different primary key, an O/R hybrid association table needs to reference only a single table. Every foreign key relationship in the database can be generalized into a single association.objectid to object.objectid foreign key.

The association table can be designed as a paired list (ObjectA_id, ObjectB_id), but this design is too limiting for complex collections, and queries must identify objectA and objectB. The more flexible alternative uses an object-association list consisting of a single objectid to reference the object, and an associationid to group associated objects. An associationtypeid column can reference association metadata that describes the association and might provide association constraints.

Figure 2 A generic three-join query locates all associated objects regardless of the class.




```
SELECT PO.ObjectCode, SO.ObjectCode
  FROM Object PO
    JOIN Association A1
      ON PO.ObjectID = A1.ObjectID
    JOIN Association A2
      ON A1.AssociationID = A2.AssociationID
    JOIN Object SO
      ON SO.ObjectID = A2.ObjectID
```

A single table for every object and another table for every association sounds radically different than a normalized schema, and it is. The physical structure is not a problem though; SQL Server excels at long narrow tables, and this design lends itself to clustered and nonclustered covering index tuning, which yields high performance.

Find All Associations

For the database architect the object-association list pattern provides amazing possibilities. First, joining an object with *n* number of other objects of any class always uses the same query (see Figure 2). Adding additional tables to a relational query adds *n*-1 joins, but the object-association list consistently uses the same three joins regardless of the number of classes involved. Depending on the application, this style of relating objects can scale considerably better than a normalized relational model. As new classes are added to the data model or to the association they are automatically included in "find all associations" queries without modifying any existing code.

Finding all associations between classes, or all objects not participating in any association with another class, or other creative yet powerful data mining queries are all trivial and reusable set-based queries. SQL Server user-defined functions can encapsulate working with associations and the many logical combinations of objects that do or do not participate in associations.

Complex collections are also possible with an object-association list. For instance, a classroom collection might include one classroom, one or more instructors, one or more desks, curriculum, and one or more students as defined in the association metadata.

Generalized associations open up more possibilities. Web pages are also linked using a generalized method; every hyperlink uses an anchor tag and a URL. It is essentially an object-association list embedded within HTML code. It is trivial to graphically map a Web page and display the navigation between Web pages regardless of the content of the Web page. Likewise, it is trivial to bounce between the object and association tables and instantly locate objects associated by several degrees of separation regardless of class.

I am currently developing a child-sponsorship management database to help organizations fight poverty. Using the object-association list, a single user-defined function that finds associations for any object can find that Joe sponsors a child in Peru, is scheduled to attend a meeting about poverty, wrote three letters to the child, sent a gift last year, and inquired about a child in Russia.

Spidering the association-object list for multiple degrees of separation also reveals that Greg is scheduled to visit the town in Peru where Joe's child lives, 14 others attended the same poverty meeting as Joe, and three of those sponsor children in Peru. This flexibility for any object with one query is impossible with a relational design.

Object Workflow State

The generality of the object-association list lends itself to another database innovation—integrating object workflow state into the database. While workflow state does not apply to all classes, for some classes, workflow is a dimension of data integrity missing from the relational database model.

A typical workflow for an order might be shopping cart, order confirmed, payment confirmed, inventory allocated, in process, ready to ship, and shipped. A relational foreign key only constrains the secondary table to referencing a valid primary key value. Using a relational database, a shipdetail row can be created that references the *order* regardless of the workflow state of the order. Custom code must validate that the order has completed certain steps prior to shipping.

> "WITH INHERITABLE WORKFLOW STATES DEFINED AS PART OF THE CLASS METADATA, THE ASSOCIATION METADATA CAN RESTRICT THE OBJECTS TO CLASS AND WORKFLOW STATE INTEGRATING WORKFLOW INTO THE OBJECT DATA"

With inheritable workflow states defined as part of the class metadata, the association metadata can restrict the objects to class and workflow state integrating workflow into the object data.

I have highlighted some of the innovations possible when object-oriented technology is modeled using today's mature relational databases. As with any tool, the Nordic Object/Relational Database Design is not the best solution for every database problem; however, for databases that benefit from inheritance, creative data mining, flexible class interactions, or workflow constraints, the O/R hybrid model can provide more power, flexibility, performance, and even data integrity than traditional relational models. •

About the Author

Paul Nielsen is a SQL Server MVP, author of the *SQL Server Bible* series (Wiley, 2002), and is writing *Nordic Object/Relational Design In Action* (Manning), which is due to be published later this year. He offers workshops on database design and optimization and may be contacted though his Web site, www.SQLServerBible.com.

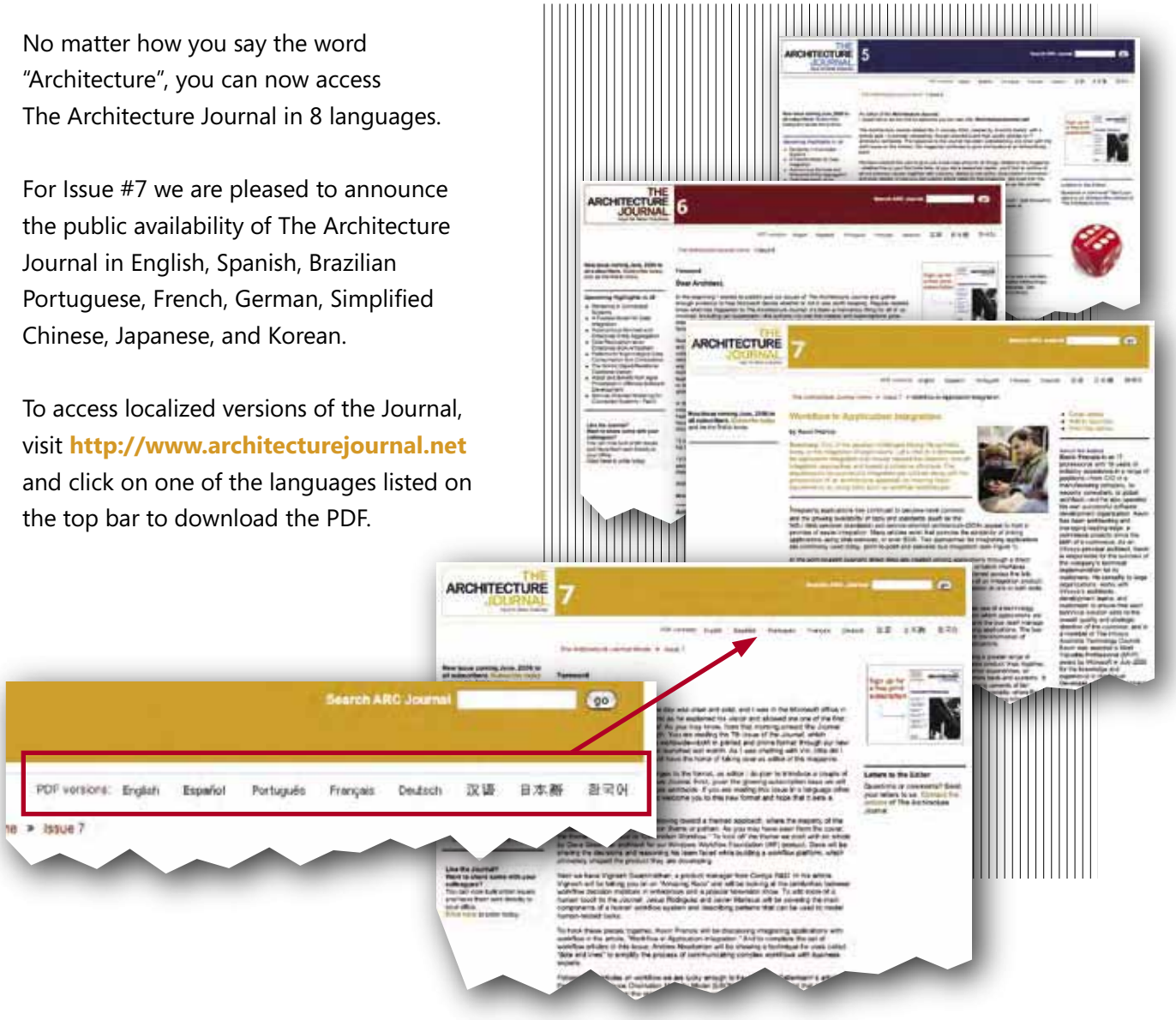
THE ARCHITECTURE JOURNAL™

Input for Better Outcomes

No matter how you say the word "Architecture", you can now access The Architecture Journal in 8 languages.

For Issue #7 we are pleased to announce the public availability of The Architecture Journal in English, Spanish, Brazilian Portuguese, French, German, Simplified Chinese, Japanese, and Korean.

To access localized versions of the Journal, visit **http://www.architecturejournal.net** and click on one of the languages listed on the top bar to download the PDF.



Now live at www.ArchitectureJournal.net!

Microsoft®

ARC

Adopt and Benefit from Agile Processes in Offshore Software Development

by Andrew Filev

Summary

In modern software development there are two trends that allow people to get more for less: agile development and offshore outsourcing. Let's look at how and when to successfully combine both to raise the competitiveness of your business.

During the post-bubble era, IT budgets were cut more than were demands for their services, which prompted managers to search for more cost-effective solutions and empowered the trend for outsourcing software development to emerging market countries (offshore development). The economic driving force is not the only force for this trend. The recent rapid growth triggered by an improved communications infrastructure also plays a major role.

With respect to working in distributed teams in general and to offshore outsourcing in particular, usable voice over Internet protocol (VoIP) software, instant messengers, e-mail clients, and wikis have made online communications easier. Moreover, now it is often preferable to use online tools such as wikis over personal communications because these tools not only help to communicate the information, but also help to structure and store it. These tools are also effective when distributing information to many recipients.

These globally available, fast Internet connections power other tools, which adds to this trend. Modeling tools help to make documentation more self-explanatory in distributed teams. Bug trackers, source control servers, Web portals, and online collaboration tools all help coordinate the distributed projects. Terminal services and virtual machines facilitate remote testing and administration.

The Internet also brought emerging market countries onto the track of high technologies. Because the Internet bypasses political borders, thousands of young people in developing countries like Russia and China use it to learn technologies that are cutting edge and improve their English skills. This new wave of Internet-educated software engineers came just in time to reinforce the offshoring trend.

The recent rise of offshore outsourcing made it big enough to become the target for political debates. For this discussion assume offshore development is an existing reality, and we'll focus on maximizing return from such outsourcing engagements. We'll skip the politics, but consult the list of resources for McKinsey Global Institute's research that quantifies the benefits of offshoring for the U.S. economy and debunks several myths about it.

Agile Software Development Trends

Let's return back onshore. Here the hearts and minds of many managers and engineers are conquered by another modern trend, agile software development. Slow *heavyweight* methods didn't prove themselves in today's dynamic business environment. Slender budgets demand more results, while bureaucracy was never the best choice in terms of return on investment (ROI). The power of agile methods lies in collaboration, flexibility, and dedication to the business value of software as reflected in core principles of the Agile Manifesto: *individuals and interactions* over processes and tools, *working software* over comprehensive documentation, *customer collaboration* over contract negotiation, and *responding to change* over following a plan (see Resources).

Agile methods greatly suit the new wave of Internet-based start-ups (often called Web 2.0). Agile software development lets some of these start-ups achieve more for less and release significant projects with small teams and tiny budgets. The short iterations and working software principles are reflected in a practice called *constant beta* after several Google products with the word "beta" incorporated in their logo.

> "AGILE METHODS ARE NOT ONE SIZE FITS ALL. THEY WORK WELL FOR SMALL CO-LOCATED TEAMS FACING RAPIDLY CHANGING CONDITIONS"

However, agile methods are not *one size fits all*. They work well for small co-located teams facing rapidly changing conditions. While there are cases where agile software development's applicability is subject to question—such as in distributed development with offshore outsourcing—my successful five years of experience applying principles of agile development in distributed teams demonstrates that it is possible and that it gives great returns when used properly.

There are other scenarios where the use of agile software development processes remains questionable. Examples are large devel-

Figure 1 Point-to-point and service bus integration

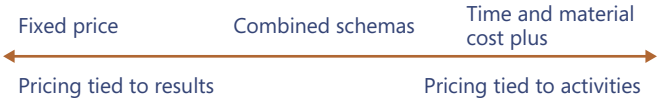


Figure 2 Three integration layers

opment teams (more than 20 people working on one independent project), systems where predictability is paramount (life-critical applications), and bureaucratic environments. We won't look at such scenarios here, and furthermore we assume a company has a corporate culture favoring agile development and intends to apply the ideas presented here to software teams of fewer than 20 people (that is, 20 people on a particular team or project, not on the entire development team). We'll instead look at the application of agile methods to distributed development in general and offshore outsourcing in particular.

Combining the Trends

Offshore software development deals represent a whole spectrum of different engagements, from hiring one offshore developer from rentacoder.com on one side, to billion-dollar deals with U.S. companies who own overseas subsidiaries on the other. Some of these deals are arranged in a way that prevents the companies from using an agile software development process, even if one of the sides wants to.

To implement an agile process, the selected outsourcing model should encourage communications and collaboration, assume flexibility, and justify releasing often. Though dozens of criteria may be applied to outsourcing deals, not many of them are as important for our further discussions as the pricing model. See Figure 1 for the mapping of the most common pricing schemas.

Predictable results imply predictable processes. In Figure 2 you can see groups of the development processes aligned on the predictive-adaptive scale. If we use a predictive-adaptive criterion, predictable schemas, which are closer to the left end of the scale (see Figure 1), require more predictability from the software development process; while agile development processes lie on the opposite side of the software development processes continuum.

The predictability concern is especially related to *fixed scope-fixed price* engagements. When this type of contract is applied to an outsourcing deal, the client and provider are naturally tied to predictive software development processes and their consequences. So this contract is not a good fit for agile software development. When designing an outsourcing engagement, remember that responsiveness to changes encourages using pricing models like time and material, which gives flexibility both to client and provider and is a better fit for agile development.

When structuring the deal, consider how the selected schema will affect communications and collaboration. An agile development process requires an open environment, tight in-team integration, shared common goals, understood business value, and frequent communication. The more barriers we have among engineers, customers, users, managers, and other stakeholders, the harder it is to provide a base for agile software development, which means reducing middlemen to allow maximum transparency and integration between teams.

Big players work this out by establishing their own branches in other countries, which makes economical sense if a company wants to have more than 100 engineers in its overseas development center. The exact number depends heavily on the other country and such factors as how easily the company may recruit talented people there.

When considering this alternative, do not repeat the mistakes of some small and midsize businesses, who underestimated the hidden expenses like top management time, travel budgets, local attorney's fees, and other costs. Also, the rapid growth of economies in developing countries causes shortages of talent, which

> "CHOOSING THE RIGHT MODEL IS ALSO VERY IMPORTANT, BUT IT DOESN'T GUARANTEE SUCCESS"

doesn't ease the life of small foreign newcomers. While local players are better networked in their local community, branches of big companies may set this off by well-known brands, bigger salaries, and better social packages.

Such luxuries are often out of reach of small companies, which may want to have only 15 developers overseas. Successful small and midsize players overcome the costs of handling remote office setup by using dedicated development teams, offshore development centers (ODCs), build-operate-transfer (BOT) models, virtual offices, virtual teams, and so forth. Regardless of the name and details, one thing is common in these models: the offshore provider is more focused on providing physical, legal, IT, and HR infrastructure to the client than on participating in the development process. The responsibility for project delivery in such deals is shared between client and provider.

To reap the full benefits of such deals, engineers must be assigned to one client, and team retention rates must be good in both companies. The provider must provide transparent communications to the client on every level, including developer-to-developer communications. People must speak one language without a translator.

In successful engagements we sometimes see that although people in remote offices get paid through the provider, they associate themselves more with the client and share the values and corporate culture of both companies.

Using the Right Practices and Tools

There are well-known practices used by successful software development teams: common coding standards; a source-control server; one-click, build-and-deploy scripts; continuous integration; unit

testing; bug tracking; design patterns; and application blocks. These practices must be applied to distributed teams more strictly than to local teams.

For example, consider continuous integration. It might be extremely frustrating to come to work and get a broken build from the source-control server, while the person responsible for it is several thousand miles away and might be seeing dreams at the moment. This issue might not be big if done by the guy in the next office, but it might become a major problem in a distributed scenario hurting productivity and communications. You can minimize such risks by sticking to continuous integration practices teamwide and installing the corresponding server (such as Microsoft Team Foundation Server, CruiseControl.NET, and CruiseControl).

Teams working on the Microsoft .NET platform are in a great position with the features provided by Microsoft Visual Studio Team System right out of the box. You get prescriptive Microsoft Solutions Framework for Agile Development and supporting tools. This product is extremely helpful for teams who need more guidance with agile development in distributed environments. For experienced teams, it's an integrated solution that provides great ROI.

Another Microsoft product, which provides great value for distributed teams, is Windows SharePoint Services (WSS). Wikis naturally fit and help agile development in distributed teams, and the next version of WSS is planned to have wiki among its enhancements. WSS is also tightly integrated with Visual Studio Team System, which makes it the best choice for the team's Web portal.

From an IT infrastructure point of view, I recommend using a virtual private network (VPN), giving the teams equal access to shared resources. The VPN environment, being less strict than a public network, allows using such features as Windows Live Messenger's application sharing, video and voice calls, remote assistance, and whiteboard.

Communication, Communication, and Communication

Working remotely, small misunderstandings quickly grow into bigger problems. In distributed development teams managers must pay attention to communication practices, which they sometimes omit without negative consequences in local development. This attention includes regular (daily/weekly) reports and status update meetings, which allow the team members to synchronize, discuss achievements, and reveal problems. Managers should also try to build personal human relations in teams through introductory meetings, on-site visits, team-building activities, and other methods.

In offshore outsourcing deals, development managers should be aware of language, cultural, and time zone barriers and must find ways to surmount these obstacles. Globalization slowly but constantly erases the cultural distinctions in the professional environment, but there are still cases when cultural differences bring confusion. There are many country-specific issues in this topic, and they are out of the scope of this discussion. Language issues are much easier to detect, though it doesn't mean that they're easier to overcome. Where companies face a language barrier, it is common and highly desirable to have company-sponsored language training for employees. In most of the offshore development countries professionals are motivated to learn English, so it is usually people in these locations who get language training.

> "AGILE DEVELOPMENT PROCESS REQUIRES AN OPEN ENVIRONMENT, TIGHT IN-TEAM INTEGRATION, SHARED COMMON GOALS, UNDERSTOOD BUSINESS VALUE, AND FREQUENT COMMUNICATION"

Variations in time zones specifically make the process more difficult. But it turns out that in countries with developed outsourcing industries, software engineers are usually ready to adapt their working schedule to work with overseas counterparts. There are two strategies to handle time zone differences. The first is to separate teams by activity; for example, have quality assurance and product managers on site and developers overseas. This arrangement allows implementing a cycle, where developers implement fixes and new requirements while their counterparts are sleeping and vice versa. Of course there should be intersections in working schedules (in the beginning/end of a working day). The second approach is to divide projects into blocks, and try to assign each block to one location, delegating as many functions as possible to this location. The second approach forces better communication and thus better serves agile development, but both work, and sometimes there is no choice.

Choosing the right model is also very important, but it doesn't guarantee success. It's highly recommended that at least one party has experience in agile development, preferably in a distributed environment. The lack of face-to-face communication, along with time, cultural, and language differences, requires attention and investing additional efforts to get desired results. The benefits of having a good offshore partner—cost-savings, on-demand staff augmentation, and outsourcing infrastructure-related tasks—(which might be summarized as "getting more for less") far outweigh investment in building the productive relations. This positive balance would be impossible without modern tools empowered by the great communications infrastructure now available globally. •

Resources

"Exploding the Myths of Offshoring," Martin N. Baily and Diana Farrell, *The McKinsey Quarterly* (McKinsey & Company, 2004) www.mckinseyquarterly.com (Note: registration required.)

Manifesto for Agile Development
http://agilemanifesto.org

About the Author

Andrew Filev (MCA, MVP) is vice president responsible for offshore operations at Murano Software. He establishes offshore development centers and leads and motivates teams. An excellent communicator, Andrew fills the gap between different cultures and builds lasting partnerships with clients.

Service-Oriented Modeling for Connected Systems – Part 2

by Arvindra Sehmi and Beat Schwegler

Summary

As architects, we can adopt a new kind of thinking to essentially *force* explicit consideration of service model artifacts into our design processes, which helps us identify the artifacts correctly and at the right level of abstraction to satisfy and align with the business needs of our organizations. In part 1 we offered a three-part approach for modeling connected, service-oriented systems in a way that promotes close alignment between the IT solution and the needs of the business. We examined the current perspective of service-orientated thinking and explained how the current thinking and poor conceptualization of service orientation has resulted in many failures and generally poor levels of return on investment (ROI). We also examined the benefits of inserting a service model in between the conventional business and technology models that are familiar to most architects, and discussed the Microsoft Motion methodology and capability mapping to identify business capabilities that can be mapped to services. In this second of two parts we'll show you how to implement these mapped services.

In part 1 we ended our discussion with a pragmatic Service Oriented Analysis and Design (SOAD) process that is used to extract all of the necessary pieces required to build your service model. This extraction includes service contracts, service-level agreements (SLAs) derived from the service-level expectation (SLE) defined for each business capability, and the service orchestration requirements. With a detailed service model closely aligned with and derived from the business model, you should now be well placed to map the service model to a technology model that identifies how each service will be implemented, hosted, and deployed.

By using the preceding approach and by creating the service model, you can hand over to your IT department data schemas, service contracts, and SLA requirements. Before the service is built, however, consideration should be given to supporting service autonomy at the technology level by clearly separating interfaces from implementation and the underlying transport mechanisms. Developing service implementation strategies that decouple the service endpoint from the service implementation helps you to plan for change. Selecting appropriate hosts and service management options to fulfill the stated SLAs also requires careful consideration. Your choices in these areas are captured and defined by the technology model.

Creating a Technology Model

The technology model consists of several artifacts: service interface, service implementation, service host, service management, and orchestration engine. we will look at the details of each.

The *service interface* specifies how a document or message can be received. The interface allows you to specify which transports are going to be provided, regardless of the implementation. More than one service interface can implement a service contract, but every service interface implements one specific binding such as SOAP over HTTP.

If required you can provide multiple service interfaces using different transports. For example, you might bind a single interface to a Web services transport and also to a Windows message-queuing transport. Each transport provides different capabilities such as interoperability and transactional support and different restrictions. For example, message queuing does not directly support the request/response pattern. At least one interface should be provided for interoperability by ensuring that the interface conforms to the Web Services Interoperability Basic Profile (WS-I BP) version 1.x.

Service implementation is the implementation of a business capability independent of the underlying host. It should also have no dependency on its interfaces. The service implementation can call other services by using binding-dependent proxies, and to achieve binding-independent proxies they should be created by using factories.

The *service host* provides an endpoint for the service interfaces. The choice of host should be based on specified SLA requirements. For example, if the business expects 24/7 operations, this characteristic has a huge impact on your choice of host, and in these scenarios queuing technologies such as Microsoft Message Queue or SQL Server Service Broker are often required. In the technology model hosts represent cost options. In the business model the SLE represents the value. By being able to map the choice of host back to an SLA captured by the service model, and ultimately to a business-driven SLE, the cost of a given host can be justified and quantified.

An added benefit of moving to services and away from silo-based applications is that if you have one specific business capability that requires 24/7 operations, you can move this service to a host that provides the necessary performance and redundancy. You might be able to use less-costly hosts for other capabilities. With the silo-based approach, you are forced to select one host for the entire application.

For the *service management* artifact appropriate action must be taken if an SLA cannot be fulfilled, and to support this action you must be able to monitor and manage the service. Service interfaces, implementations, and hosts should be instrumented, for example, by using Windows Management Instrumentation (WMI). Microsoft Operations Manager (MOM) can then be used to monitor and manage the service, while Microsoft Systems Management Server (SMS) is used to support change and configuration management in the service's "edge" ecosystem. Your chosen orchestration engine should also provide monitoring support such as the Business Activity Monitoring (BAM) features provided by BizTalk Server. In the future, WS-Management will play a central role in managing resources independently from the hosting and management platform.

For the *orchestration engine* artifact orchestration requirements should be defined in a platform-independent way by the service model, but ultimately you need to use a specific orchestration engine. The technology model identifies the target platform such as Microsoft BizTalk Server.

By creating the technology model you can explicitly capture the foregoing artifacts, but how should you approach service development? How do you map these artifacts to a service implementation, and how do you implement the contract specified by the service model? Shortly, we will discuss an approach that helps you to build services in a way that satisfies the service tenets and modeling principles discussed earlier.

Creating Services Today

Given a set of conceptual service artifacts, how can you define these in code, and how should you organize the code within Visual Studio 2005? The key thing to bear in mind is the need for separation among

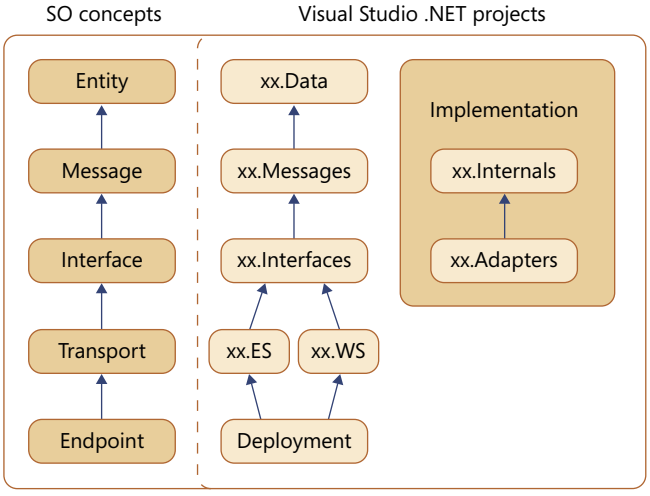
> "EACH TRANSPORT PROVIDES DIFFERENT CAPABILITIES SUCH AS INTEROPERABILITY AND TRANSACTIONAL SUPPORT AND DIFFERENT RESTRICTIONS"

data, messages, interfaces, implementation internals, and transport bindings. Figure 1 shows an approach for mapping service artifacts to Visual Studio 2005 solutions and projects.

The "xx" placeholders shown in Figure 1 represent an appropriate namespace based on the service name—for example, OrderService. Using a single Visual Studio 2005 solution file for a given service and as a container for the multiple projects you see in Figure 1 is a recommended practice. We will look at these projects.

- The *xx.Data* project is used to hold the on-the-wire data schema definitions that are obtained from the service model entity and also contains the common language runtime (CLR) representation of that schema.
- The *xx.Messages* project is used to hold definitions of the messages required to communicate with the service, which includes inbound and outbound messages. A message is represented as a schema element containing elements of the data contract.
- The *xx.Interfaces* project contains the interface definitions.
- The *xx.ES* and *xx.WS* projects contain transport endpoints for interfaces. In the example shown in Figure 1, xx.ES provides an enterprise services transport, and xx.WS provides a Web services transport.
- The *xx.Adapters* and *xx.Internals* projects contain the implementation of the service. An adapter contained in the xx.Adapters project provides a level of indirection between the interface and the internal implementation. The adapter parses the inbound message and passes the relevant data entities to the internal implementation code. The adapter also wraps data output from the implementation within the response message. Note that with this approach, the implementation knows nothing about the messages passed through the interface, which allows you to change your messaging infrastructure without impacting your internal service implementation.

Figure 1 Mapping service artifacts to Visual Studio 2005 projects



The endpoint shown in Figure 1 is purely a deployment artifact and is dependent on your choice of transport binding. For example, with a Web services transport, you would deploy your service to an Internet Information Services (IIS) Web server running ASP.NET.

Adapters. The adapter is a key artifact that enables you to decouple your implementation from the messaging. It is also a good place to perform message transformations if they are required. For example, within the adapter you could transform an external representation of a customer ID (for example, 10 characters) into your internal representation (for example, 12 digits). By creating a new adapter per version of your service, the adapter also provides a way to version services without affecting the implementation or your previously published service interfaces.

Versioning. When considering versioning, you need to distinguish between versioning the abstract or concrete contract and versioning the internal implementation. A key architectural goal is to provide independent versioning between the two. The contract itself consists of the data, message, interface, and endpoints. You can version each of these artifacts independently if you architect the service so that each composing artifact refers to a single version of the composed artifact. For example, the message contract that is composed of the data contract should refer to precisely one version of the data contract. This reference must be true for the XSD/WSDL and the CLR representation of that contract.

Data and message contracts should be versioned according to the XML versioning and extensibility policy (see Resources). You can version service interfaces by providing a new interface that inherits from the previous (CLR), one that is mapped into a new WSDL port type.

Six Steps for Building a Service

To arrive at these Visual Studio 2005 projects and build a service today using this approach, adopt this six-step process:

1. Design the data and message contract.
2. Design the service contract.
3. Create the adapters.
4. Implement the service internals.
5. Connect the internals to the adapters.
6. Create the transport interfaces.

Let's look at the details of each step. For the first step—design the data and message contract—take the canonical data schema that defines the wire representation of the data (an output of your service-oriented analysis and design) and define the data classes and message classes. There are two approaches to creating your data schemas and data classes. By using a schema-first approach, you can create an XSD schema and then use a tool such as Xsd.exe or XsdObjectGen.exe to generate the data classes automatically. Alternatively, if you prefer a code-first approach, you can define your data classes in C# or Visual Basic and then use Xsd.exe to create an equivalent XSD schema. Here is an example of a data contract:

```
namespace DataContracts
{
   [Serializable]
   [XmlType("Order", Namespace=
      "urn.contoso.data/order")]
   public partial class Order
   {
      [XMlElement("Customer")]
      public Customer customerField;
      [XMlElement("Items")]
      public OrderItemsList
         ordersItemsField;
      ...
   }
}
```

Once you have defined your data classes, you can define your inbound and outbound messages, which contain the data classes as their payload. For example, this code snippet shows an input message called OrderMessage for a hypothetical order service that contains an Order as its payload:

```
namespace MessageContracts
{
   using System.Xml;
   using System.Xml.Serialization;
   using DataContracts;

   [Serializable]
   [XmlType(Namespace=
      "urn.contoso.msgs/orderservice"
      )]
   [XmlRoot(Namespace=
      "urn.contoso.msgs/
      orderservice"]
   public class OrderMessage
   {
      [XmlElement("Order")]
      public Order order;
   }
}
```

In step 2—design the service contract—define the abstract service contract either by using a WSDL-first approach or by defining your interfaces using C# or Visual Basic. Your interfaces define which messages your service receives and which messages (if any) it returns. To generate the interface from the WSDL, you need to use the Wsdl.exe /si switch:

```
Wsdl.exe xx.wsdl /si
```

Note that this switch works only with Microsoft .NET Framework version 2.0. This next example defines an interface to the order service that defines a single PlaceOrder() method that accepts an OrderMessage message and returns an OrderTrackingMessage message:

```
namespace Interfaces
{
   using MessageContracts;

   public interface IOrderService
   {
      OrderTrackingMessage
         PlaceOrder(OrderMessage
         placeOrderMsg);
   }
}
```

Note that in this case a request/response message pattern is used, and therefore you need to ensure that the message processing time (the time between the request and response) must be sub-second. If this processing time cannot be guaranteed, use another message-exchange pattern such as a duplex message exchange, which correlates two one-way messages to a logical request/response pattern.

To generate the WSDL from the preceding interface, you can create an ASMX Web service class that implements the interface and then calls the Web service passing ?WSDL—for example, http://localhost/OrderService/OrderService.asmx?wsdl. This Web service causes the WSDL to be generated and returned from the Web service.

In step 3—create the adapters—create the adapter class that provides the indirection between the interface and the internals. This class ensures that the internals knows nothing about the message contract. The adapter unwraps the inbound message and passes the payload data to the internal implementation performing any required data

```
namespace Endpoints.WS
{
  using System;
  using System.Diagnostics;
  using System.Web.Services;
  using System.ComponentModel;
  using System.Web.Services.Protocols;
  using System.Web.Services.Description;
  using System.Xml.Serialization;
  using MessageContracts;
  using ServiceInterfaces;
  using Adapters;

  [WebService(Namespace=
    "urn.contoso.interfaces/orderservice")]
  public class OrderService : System.Web.Services.
    WebService, IOrderService
  {
    [WebMethod]
    [SoapDocumentMethod(ParameterStyle=
      SoapParameterStyle.Bare)]
    public OrderTrackingMessage PlaceOrder(
      OrderMessage placeOrderMsg)
    {
      IOrderService adapter = new OSA();
      return adapter.PlaceOrder(PlaceOrderMsg);
    }
  }
}
```

Listing 1 The IOrderService interface

format transformations from message to internal format. Similarly, on the way back the adapter performs any necessary format transformations on the data returned from the service implementation and wraps it inside an outbound service message if there is one. Here's a sample adapter:

```
namespace Adapters
{
   using MessageContracts;
   using ServiceInterfaces;

   public class OSA : IOrderService
   {
      public virtual
         OrderTrackingMessage
         PlaceOrder(
         OrderMessage placeOrderMsg)
      {
         // Call internals here
      ...
      }
   }
}
```

Note that the adapter is always in process with the caller, and the process identity depends on your choice of host. If the internal implementation is hosted within IIS, then the ASMX ("edge") interface instantiates the service internal. If the internal is hosted within enterprise services, but the calls arrive through an ASMX Web service, the ASMX interface delegates the call to the enterprise services interface, which then instantiates the adapter and passes the message to it. Because all transports implement the same interface you can chain calls together.

In step 4—implement the service internals—create your service internal implementation. A single implementation exists regardless of your chosen transport or transports. This code snippet shows the skeleton code required to provide an implementation of the order processing service:

```
namespace BusinessLogic
{
   public class OSI
   {
      public static string
         AcceptOrder(
         DataContracts.Order order)
      {
         // Process the order
         ...
         return "XYZ";
      }
   }
}
```

Notice how the internal implementation has no knowledge of the message. It knows about only the data contract. In this case it is passed a single Order object. If you want to be completely independent of the wire format, the internal would not even have access to the data contract. In this scenario, the adapter would map the external data contract to the internal data types.

In step 5—connect the internals to the adapters—call your internal implementation from your adapter code. Note that the message is not passed to the implementation, which decouples the internal implementation from the message contract and enables you to change one without impacting the other. This code snippet shows the adapter code again, this time with a call to the internal service implementation:

```
namespace Adapters
{
   using MessageContracts;
   using ServiceInterfaces;

   public class OSA : IOrderService
   {
      public virtual
         OrderTrackingMessage
         PlaceOrder(
         OrderMessage placeOrderMsg)
      {
         OrderTrackingMessage otm =
            new OrderTrackingMessage();
```

```
            otm.TrackingId =
               BusinessLogic.OSI.
               AcceptOrder(
               placeOrderMsg.Order);
            return otm;
         }
      }
   }
```

In step 6—create the transport interfaces—bind your abstract service interfaces defined in step 2 to a specific transport. Listing 1 shows the IOrderService interface defined in step 2 bound to a Web services transport.

Using Process-Automated Guidance

By following the six-step process just discussed, you can build services that conform to all tenets and principles covered here. However, because all of the items described in the six-step process can be described by metadata, it is possible to automate large sections of the service generation. You can use tools such as the Guidance Automation Toolkit (GAT) to help automate the guidance (see Resources). Process-automated guidance is particularly helpful to transform between semantically identical XML and CLR artifacts such as XSD and CLR classes, transform among the WSDL port type and CLR interfaces, generate adapters based on the interface description, and generate different endpoint technologies based on the interface description.

The old way of thinking about service orientation is not working, and a new way of thinking is required. By adopting this new kind of thinking, as architects we can *force* explicit consideration of service model artifacts in your design process, which helps you to identify the artifacts correctly and at the right level of abstraction to satisfy and align with business needs.

From a modeling perspective, the gap between conventional business and technology models is too large, which is a key contributing factor to the failure of many service-orientation initiatives. We've presented a three-part model with the introduction of a service model in between the business and technology models to promote much closer alignment of your services with the needs of the business. With a detailed service model closely aligned with and derived from the business model, you are well placed to map the service model to a technology model that identifies how each service will be implemented, hosted, and deployed. Capability mapping and the Motion methodology provide an effective way to identify business capabilities and ultimately services. The decomposition of the business into capabilities provides the top-level decoupling for the underlying service contracts, and not the other way around as it usually is today.

Connected systems are instances of the entire three-part model, and they respect the four tenets of service orientation. They can be *more completely* implemented by using the five pillars of Microsoft's platform technologies. Recall that we asked upfront: How do we avoid making the same mistakes with SOAs that previous, hopeful initiatives have resulted in? How do we ensure that the chosen implementation architecture relates to the actual or desired state of the business? How do we ensure a sustainable solution that can react to the dynamically changing nature of the business—in other words, how can we enable and sustain an agile business? How can we migrate to this new model elegantly and at a pace that we can control? And, how can we make this change with good insight into where we can add the greatest value to the business from the outset?

Service orientation with Web services is only the implementation of a particular model. It is the quality and foundation of the model that determines the answers to these questions.

Acknowledgements

The authors would like to thank Ric Merrifield, director, Microsoft Business Architecture Consulting Group; David Ing, independent software architect; Christian Weyer, architect, thinktecture; Andreas Erlacher, architect, Microsoft Austria; and Sam Chenaur; architect, Microsoft Corporation for providing feedback on early drafts of this article. We would also like to show our appreciation to Alex Mackman, principal technologist, CM Group Ltd., an excellent researcher and writer who helped us enormously. •

Resources

"Designing Extensible, Versionable XML Formats," Dare Obasanjo (Microsoft Corporation, 2004) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml07212004.asp

"Modeling and Messaging for Connected Systems," Arvindra Sehmi and Beat Schwegler
A Webcast of a presentation at Enterprise Architect Summit–Barcelona (FTPOline.com, 2005) www.ftponline.com/channels/arch/reports/easbarc/2005/video/

MSDN – Guidance Automation Toolkit (GAT) http://msdn.microsoft.com/vstudio/teamsystem/Workshop/gat/default.aspx

Obtain a case study on Microsoft Motion methodology by sending a request to motion@microsoft.com.

"Service-Oriented Modeling for Connected Systems – Part 1," Arvindra Sehmi and Beat Schwegler, *The Architecture Journal*, Issue 7, March 2006. http://www.thearchitecturejournal.net

About the Authors

Arvindra Sehmi is head of enterprise architecture in the Microsoft EMEA Developer and Platform Evangelism Group. He focuses on enterprise software engineering best practices adoption throughout the EMEA developer and architect community and leads architecture evangelism in EMEA for the financial services industry. Arvindra is editor emeritus of the Microsoft *Architecture Journal*. He holds a Ph.D. in biomedical engineering and a master's degree in business.

Beat Schwegler is an architect in the Microsoft EMEA Developer and Platform Evangelism Group. He supports and consults to enterprise companies in software architecture and related topics and is a frequent speaker at international events and conferences. He has more than 13 years experience in professional software development and architecture and has been involved in a wide variety of projects, ranging from real-time building control systems and best-selling shrink-wrapped products to large-scale CRM and ERP systems. For the past four years, Beat's main focus has been in the area of service orientation and Web services.


ready_
ready_
ready_
Welcome to the people ready business.

Your potential. Our passion.™

Microsoft®

In a people-ready business, people make it happen. People, ready with software. When you give your people tools that connect, inform, and empower them, they're ready. Ready to make the most of their knowledge, skill, and ambition. Ready to develop new products, help customers, and solve problems. Ready to build a successful business: a people-ready business. Microsoft. Software for the people-ready business.℠ To learn more, visit **microsoft.com/peopleready**