Aprenda a disciplina, pratique a arte e contribua com idéias no:

www.ArchitectureJournal.net

Recursos nos quais você pode confiar.

THE ARCHITECTURE JOURNAL

Idéias para melhores resultados

Journal 8

Estratégias by Design

Microsoft

Sumário

THE ARCHITECTURE JOURNAL
Idéias para melhores resultados

Journal 6

Apresentação

Caro Arquiteto,

Bem-vindo à Edição 8 do *The Architecture Journal*, cujo tema é Dados By Design. Como arquiteto, percebo que, freqüentemente, não damos a devida importância à onipresença dos dados na arquitetura, principalmente quando se leva em conta o modo de uso dos dados em aplicativos e sistemas que abarcam várias geografias, fusos horários e organizações.

Uma analogia que eu costumo usar compara a disponibilidade dos dados dentro de um sistema com a água que corre nos canos de uma casa. Quando abrimos uma torneira, esperamos que uma água limpa e filtrada corra instantaneamente, geralmente com uma pressão determinada e constante. Nessa analogia, os canos são a infra-estrutura e os dados são a água. Ao considerar os dados e sua relação com a arquitetura, gosto de pensar na mesma abordagem — os dados que oferecemos aos usuários devem estar limpos e filtrados e devem ser entregues sem atraso, da forma esperada — não importando se os dados são um único e-mail, um registro de cliente ou um conjunto grande de dados financeiros mensais.

Apesar de não abordarmos várias técnicas de encanamento nessa edição, temos um grande número de ótimos artigos de um seleto grupo de autores que enfocam a importância dos dados.

Começamos com Roger Wolter, arquiteto de soluções da Microsoft e autor de vários documentos técnicos e livros sobre SQL Server e SQL SSB (Server Service Broker). Roger fala da importância da confiabilidade dos dados, particularmente no contexto do Projeto de Sistemas Conectados.

Tim Ewald e Kimberly Wolk escreveram o artigo seguinte, que explica o modelo que eles criaram para a integração dos dados do MSDN TechNet Publishing System, o sistema da próxima geração, baseado em XML, que constitui a base do MSDN2. Em seguida, Udi Dahan nos conduz em uma jornada sobre o uso da agregação de entidades para obter visualizações em 360 graus das entidades de dados e enfocar modos concretos de atender às necessidades de negócio imediatas.

Depois, outra equipe de autores, Tom Fuller e Shawn Morgan, compartilha algumas experiências, mostrando que a replicação de dados pode ser um antipadrão para a SOA (Arquitetura Orientada a Serviços), principalmente no contexto de aplicativos e serviços autônomos. Em seguida, Dion Hinchcliffe, CTO da Sphere of Influence, apresenta alguns padrões de consumo e composição de dados, mais especificamente nas áreas de mashups e aplicativos Web 2.0.

Paul Nielsen termina a nossa série de documentos sobre dados apresentando uma visão geral do Nórdico, um modelo híbrido relacional/de objeto que tem maior flexibilidade e melhor desempenho que os modelos de dados relacionais convencionais. Concluindo esta edição do Journal, Andrew Filev compartilha algumas de suas experiências, combinando uma metodologia ágil com um modelo de terceirização e, em seguida, Arvindra Sehmi e Beat Schwegler voltam com a Parte 2 da série Modelagem para Sistemas Conectados. Se você perdeu a Parte 1, baixe a Edição 7 do *The Architecture Journal* no *site www.architecturejournal.net*.

Isso conclui o nosso tema, os dados. Acredito que os artigos apresentados nesta edição o ajudem a projetar sistemas com a mesma disponibilidade da água que corre nas torneiras da sua casa — logicamente, não posso garantir que você não vai molhar as mãos!

Simon Guest

Confiabilidade em Sistemas Conectados

por Roger Wolter

Resumo

Os aplicativos de sistemas conectados são compostos por vários serviços fracamente acoplados que geralmente estão espalhados em uma rede; portanto, a obtenção de altos níveis de confiabilidade e disponibilidade em aplicativos de sistemas conectados representa um conjunto único de desafios arquitetônicos. Por exemplo: se uma aplicação pára de funcionar porque um dos dez serviços executados em dez servidores diferentes não está disponível, a taxa de falha do aplicativo é cerca de dez vezes maior que a taxa de falha dos serviços individuais. A falha de dados torna esse problema muito mais crítico, já que uma fonte de dados pode ser usada por dezenas de serviços. Esse artigo trata dos problemas de confiabilidade que devem ser considerados ao arquitetar um aplicativo de serviços conectados e mostra como alguns dos novos recursos do SQL Server 2005 e dos produtos de troca de mensagens da Microsoft abordam esses problemas.

Sob o ponto de vista do hardware, do software e da manutenção, a confiabilidade é cara – portanto, é muito importante entender os requisitos de confiabilidade de aplicativo. Embora um aplicativo sem requisitos de confiabilidade seja raro, a implementação de um aplicativo com confiabilidade maior do que a necessária pode ser um desperdício de tempo e recursos. Por isso, é importante entender os problemas de confiabilidade dos sistemas conectados, para poder arquitetar uma solução que proporcione o nível adequado de confiabilidade, mas sem desperdiçar recursos com uma confiabilidade além do necessário.

Na SOA (Arquitetura Orientada a Serviços), também conhecida como sistemas conectados, os serviços se comunicam entre si por meio de formatos de mensagens bem-definidos; isso significa que a confiabilidade do aplicativo de sistemas conectados sofrerá uma grande influência da confiabilidade da infra-estrutura de mensagens que ele usa para fazer a comunicação entre os serviços. Usaremos como exemplo os serviços de um aplicativo de caixa eletrônico para ilustrar os diversos graus de confiabilidade do sistema de mensagens e como eles são obtidos. Geralmente, o processamento de mensagens entre serviços é mais complexo que o das mensagens entre cliente e servidor, porque as mensagens cliente/servidor podem contar com o usuário para tomar decisões sobre o modo de lidar com várias situações de erro e limites de tempo, ao passo que o servidor que inicia a troca nas mensagens de servidor a servidor precisa tomar todas as decisões que normalmente seriam tomadas pelo usuário final em uma interação cliente/servidor.

Um Peso para a Infra-estrutura

Muitas aplicações orientadas a serviços atingem uma taxa de transferência melhor usando um sistema de mensagens assíncrono. Com esse sistema, um serviço envia uma mensagem a outro serviço sem esperar a resposta da mensagem para continuar. O serviço processa muito mais solicitações porque não perde tempo esperando respostas às solicitações que ele faz a outros serviços; em vez disso, deixa o encargo de garantir que a mensagem seja entregue e processada à infra-estrutura de mensagens. A opção pelo processamento síncrono ou assíncrono de mensagens geralmente é determinada pelos requisitos de negócio do aplicativo.

Por exemplo: se um cliente está tentando sacar dinheiro de um ATM (Caixa Eletrônico), o ato de fazer uma solicitação síncrona para verificar o saldo do cliente e liberar o dinheiro sem esperar a resposta do saldo bancário geralmente não é uma decisão de negócios correta. Entretanto, esse cenário não descarta solicitações assíncronas. O caixa eletrônico pode despachar uma solicitação de saldo assim que o cliente seleciona a opção de saque e, em seguida, permitir que o cliente digite a quantia enquanto a verificação do saldo é realizada de forma assíncrona. Assim que a quantidade do saque é digitada e o saldo é recebido, o aplicativo de caixa eletrônico pode tomar a decisão de liberar ou não o dinheiro.

Vamos ver como o caixa eletrônico processa a mensagem de solicitação de saldo. O serviço de caixa eletrônico enviará a mensagem de solicitação de saldo ao serviço de contas para obter o saldo da contado cliente. Nesse ponto, uma dessas três coisas irá acontecer: O saldo será retornado com êxito, um erro será retornado ou o tempo limite da solicitação irá expirar porque o resultado não foi retornado no tempo certo. Se o saldo é retornado, o serviço de caixa continua com a transação. Se um erro é retornado, o caixa eletrônico usa a sua lógica de negócio para trabalhar com ele — talvez, usando uma cópia em cache do saldo ou liberando ou não o dinheiro dependendo da quantia solicitada. A situação mais difícil de trabalhar é o limite de tempo. O estouro do limite de tempo pode significar que a mensagem se perdeu em trânsito, a mensagem chegou ao serviço de contas, mas a resposta atrasou por alguma razão ou a resposta foi enviada e se perdeu no caminho de volta.

Na maioria dos casos, a melhor resposta é tentar novamente e esperar que funcione melhor da próxima vez. Se a mensagem tiver se perdido na volta da primeira vez, pode chegar dessa vez. Se a resposta estava lenta, a resposta original pode chegar antes do limite de tempo da segunda solicitação. A não ser que o sistema de contas esteja inacessível ou completamente inoperante, as novas tentativas acabam dando certo. Dependendo do problema, a solicitação de saldo pode ter sido processada várias vezes, ou vários resultados podem ser recebidos, mas, desde que o serviço de caixa eletrônico esteja preparado para trabalhar com eles, esses problemas não são graves.

Se a mensagem tiver sido enviada de forma assíncrona, o serviço de caixa eletrônico poderá não ter as informações à disposição para reenviar a mensagem quando se solicita uma nova tentativa – portanto, a infra-estrutura de mensagens precisará manter uma cópia das mensagens

enviadas e reenviá-las se necessário. Para esse tipo de mensagem de solicitação simples, manter uma cópia da mensagem na memória provavelmente é adequado porque, se o caixa eletrônico ficar sem energia, o cliente terá que começar novamente de qualquer forma. Se o serviço de caixa eletrônico precisar de uma resposta até mesmo depois de uma queda de energia, a mensagem terá que ser colocada em algum armazenamento persistente, e o sistema de mensagens terá que reenviá-la quando for reiniciado. A Figura 1 mostra três possíveis resultados de uma solicitação de saldo.

Entrega de Mensagens

Para a versão síncrona dessa solicitação, um serviço da Web simples em SOAP proporciona o grau necessário de confiabilidade. O processamento de erros e do limite de tempo é de responsabilidade do serviço de caixa eletrônico e, portanto, ele determinará o nível de confiabilidade da solicitação. Para a versão assíncrona da solicitação de saldo, ou o canal WS-RM do WCF (Windows Communication Foundation) ou a entrega expressa do MSMQ proporcionará a confiabilidade necessária se a solicitação não precisar "sobreviver" a falhas do sistema. Se a mensagem precisar sobreviver ao reinício do sistema, a entrega recuperável do MSMQ é a opção adequada. Esses sistemas de mensagem processarão os limites de tempo e as novas tentativas que se fizerem necessários para entregar a mensagem e a resposta, para que o serviço de ATM não tenha que incluir essa lógica.

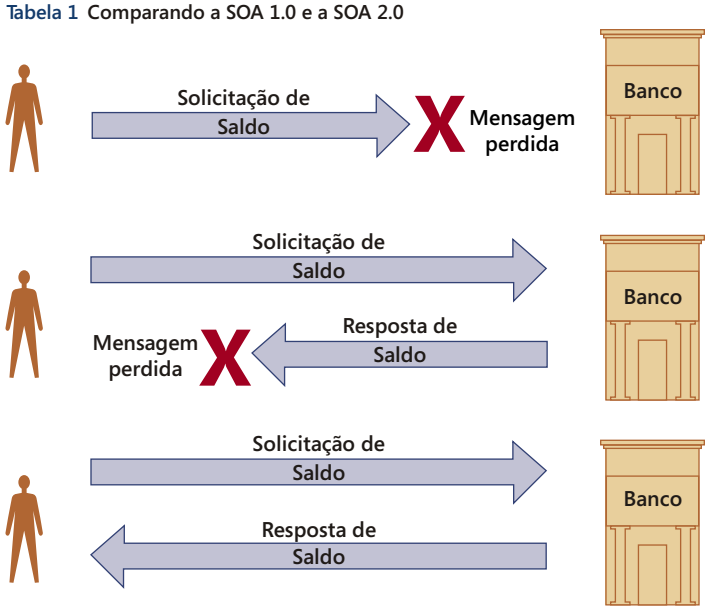
Agora que entendemos os problemas de confiabilidade em uma solicitação de saldo, vamos passar à solicitação de alteração de saldo que acontece depois da liberação do dinheiro. Os requisitos de confiabilidade dessa mensagem são muito mais altos porque, se ela não for entregue com sucesso, o banco dará dinheiro ao cliente sem reduzir o saldo da conta dele. Embora o cliente provavelmente não vá reclamar, o banco não ficará satisfeito se essa situação acontecer regularmente.

A mensagem de alteração de saldo tem que ser entregue e processada, independentemente de qualquer falha de rede ou de sistema. O mecanismo de nova tentativa baseado na memória obviamente não é adequado para essa tarefa, porque, em uma falha do sistema, a cópia da mensagem mantida para novas tentativas é perdida. A manutenção de uma cópia persistente da mensagem para novas tentativas também pode ser um problema, porque é possível que várias cópias da mensagem sejam enviadas por causa das tentativas. Se a mensagem descontar $200.00 do saldo da conta do cliente, por exemplo, a entrega repetida da mensagem poderá causar insatisfação do cliente.

As tentativas necessárias para garantir a entrega confiável da mensagem só funcionam se as mensagens forem idempotentes. Uma mensagem idempotente pode ser entregue várias vezes sem violar as restrições de negócio do aplicativo. Algumas mensagens são naturalmente idempotentes. Por exemplo: uma mensagem que altera o endereço do cliente pode ser executada várias vezes sem efeitos prejudiciais. Outras mensagens não são naturalmente idempotentes então, o sistema de mensagens precisa tornálas idempotentes para evitar o dano que pode resultar do processamento repetido de uma mensagem (consulte as Figuras 2 e 3). Normalmente, essa transação é feita mantendo-se uma lista das mensagens que já foram processadas; se a mesma mensagem for recebida mais de uma vez, ela não será processada mais de uma vez.

A maioria dos sistemas de mensagem não mantém uma cópia das mensagens que entram, mas o remetente atribui um identificador a cada mensagem e o destinatário faz o acompanhamento dos identificadores que já viu. Esses identificadores devem ser mantidos até que o destino da mensagem tenha certeza de que a origem descartou a mensagem porque, se a origem falhar, a mensagem será enviada novamente dias depois, quando a origem for reiniciada. A lista de mensagens processadas também deve ser persistente porque o serviço de destino pode falhar entre as tentativas. O destino também precisa rastrear a mensagem que foi enviada em resposta à mensagem de entrada, porque a origem ficará enviando a mensagem original até receber a resposta; o motivo da nova tentativa pode ser a perda da mensagem de resposta original.

Tabela 1 Comparando a SOA 1.0 e a SOA 2.0



Escolher uma infra-estrutura de mensagem

A não ser que você queira persistir e rastrear as mensagens na lógica do seu serviço, precisará de uma infra-estrutura de mensagem confiável para proporcionar esse nível de confiabilidade. A Microsoft oferece três opções para essa infra-estrutura: mensagem recuperável do MSMQ, SQL Server Service Broker e BizTalk. A opção escolhida depende dos requisitos de confiabilidade e do projeto do aplicativo.

O Service Broker e o BizTalk oferecem um armazenamento de mensagens mais confiável que o do MSMQ, porque ele armazena mensagens em um banco de dados SQL Server, ao passo que o MSMQ as armazena em um arquivo. Se o seu aplicativo puder funcionar com a perda ocasional de mensagens quando um arquivo é perdido ou corrompido, o MSMQ é adequado às suas necessidades. Alguns aplicativos MSMQ protegem contra a possível perda de mensagens, armazenando uma cópia da mensagem em um banco de dados. Se você for armazenar mensagens, o uso do Service Broker, que já armazena mensagens no banco de dados normalmente, é muito mais eficiente.

O Service Broker e o BizTalk proporcionam praticamente o mesmo grau de confiabilidade e defesa contra a perda de mensagens porque ambos as armazenam no banco de dados. O processamento de mensagens do Service Broker está incorporado à lógica de processamento do banco de dados e, portanto, o Service Broker conversa diretamente do processo do SQL Server com os soquetes de TCP/IP algo muito mais eficiente que a abordagem do BizTalk, na qual um processo externo chama o banco de dados para armazenar as mensagens em uma tabela. Embora o Service Broker consiga entregar muito mais mensagens por segundo que o BizTalk, o BizTalk oferece uma grande quantidade de recursos transformação de mensagens, roteamento de mensagens de acordo com os dados, vários transportes de mensagens, orquestração etc .— que o Service Broker não oferece.

Em geral, se o seu aplicativo só precisa de uma transferência confiável de mensagens entre os bancos de dados, o Service Broker é a melhor opção, por ser mais leve e eficiente na transferência de mensagens que o BizTalk. Por outro lado, se o seu aplicativo requer os recursos de manipulação de mensagens e as integrações de dados que o BizTalk oferece, o Service Broker provavelmente não é a solução correta.

Embora as mensagens recuperáveis do MSMQ não proporcionem os níveis de confiabilidade que os recursos baseados no SQL Server oferecem, elas têm a vantagem de não precisar do SQL Server nas duas extremidades das mensagens. Se o suporte a um banco de dados nas

Figura 2 Nova tentativa de solicitação de saque sem problemas

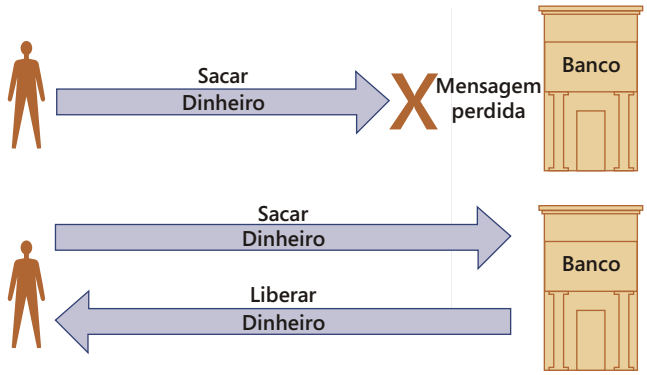
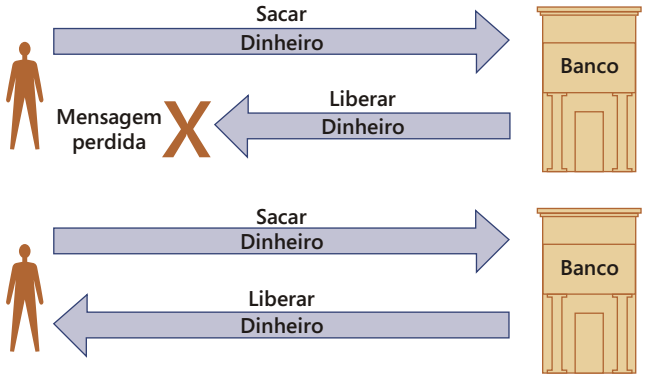


Figura 3 Nova tentativa de solicitação de saque na qual o cliente perde dinheiro



duas extremidades está fora de cogitação e os requisitos de confiabilidade da aplicação podem ser cumpridos pelo MSMQ, o MSMQ é provavelmente a opção correta para a infra-estrutura de mensagens. Entretanto, se os dois sistemas de comunicação exigirem armazenamento de dados, a confiabilidade extra obtida com o armazenamento de mensagens no banco de dados vale à pena.

No nosso exemplo de caixa eletrônico, o serviço requer armazenamento local de dados para auditoria, armazenamento local para operação offline e armazenamento de dados de referência; portanto, provavelmente haverá um banco de dados no caixa eletrônico de qualquer forma, e uma das opções baseadas no SQL Server é adequada. A escolha entre o Service Broker e o BizTalk depende dos requisitos do aplicativo que não estão relacionados a mensagens, dos recursos disponíveis no caixa eletrônico e dos requisitos de volume de mensagens.

Confiabilidade na Execução

Abordamos anteriormente a confiabilidade na entrega de mensagens de um serviço a outro. Não é de se surpreender que tenhamos constatado que o nível necessário de confiabilidade depende do que o aplicativoestá fazendo e da importância dos dados da mensagem para o aplicativo. Neste texto, iremos pressupor que as mensagens sejam transferidas com o grau necessário de confiabilidade a um serviço e analisar os requisitos de confiabilidade para o serviço que processa as mensagens.

Um dos requisitos específicos de um serviço que processa mensagens assíncronas é que o recebimento de uma mensagem da fila é uma operação de "recepção". Em outras palavras, quando a mensagem chega a uma fila, ela fica lá até que o aplicativo execute uma operação de "recepção" para recuperar e processar a mensagem. Esse requisito significa que um serviço síncrono deve garantir a execução quando há mensagens na fila para serem processadas. A forma mais comum de cumprir esse requisito é transformar o serviço em um serviço Windows gerenciado pelo Windows SCM (Service Control Manager). O SCM irá garantir que o serviço seja iniciado quando o Windows for iniciado e pode ser configurado para reiniciar o serviço caso ele falhe por alguma razão.

Embora essa configuração geralmente proporcione o nível de confiabilidade necessário e, em geral, seja a solução preferida quando as mensagens chegam em um ritmo constante, ela pode causar problemas se a carga de mensagens variar de forma significativa. Se o serviço for configurado com recursos suficientes para processar picos de carga, isso será um desperdício de recursos quando a carga estiver baixa; se estiver configurado para processar a carga média, não conseguirá dar conta dos picos de carga. As mensagens do BizTalk operam como um serviço Windows e, portanto, um aplicativo com o BizTalk pode contar com o serviço Windows para processar as mensagens de entrada.

O MSMQ trata do problema da carga de mensagens por meio de disparadores que iniciam um serviço de processamento de mensagens sempre que uma mensagem chega à fila. Embora esse processamento funcione bem quando as mensagens chegam com pouca freqüência, quando a carga de mensagens é alta, a sobrecarga de iniciar mil cópias dos serviços pode ser maior que o da própria lógica do serviço.

Para resolver esse problema, o Service Broker oferece um recurso chamado *ativação*. Quando uma mensagem chega a uma fila vazia, o Service Broker inicia um procedimento armazenado para processar a mensagem. Esse procedimento armazenado aguarda a chegada de mais mensagens em um loop e continua nesse loop até que a fila esteja vazia. Se o Service Broker determinar que o procedimento armazenado não está dando conta das mensagens que entram, ele iniciará cópias adicionais do procedimento armazenado até que haja cópias suficientes em execução. Quando a taxa de chegada de mensagens diminuir, a fila ficará vazia e as cópias extras sairão. Portanto, sempre haverá aproximadamente o número adequado de recursos disponíveis para trabalhar com as mensagens que entram. Como o Service Broker inicia esses procedimentos, ele será notificado se um deles falhar e substituirá a cópia com defeito. Se o serviço for um aplicativo externo em vez de um procedimento armazenado, o Service Broker oferecerá eventos, nos quais o aplicativo externo pode se inscrever, que notificarão o serviço quando houver necessidade de mais recursos para processar as mensagens de uma fila.

Mensagem Perdida

O outro problema de confiabilidade com o qual a execução do serviço tem que trabalhar é uma falha de serviço durante o processamento de uma mensagem. Se o serviço excluir uma mensagem da fila assim que a receber e, em seguida, falhar antes de processá-la completamente, essa mensagem estará efetivamente perdida. Da mesma forma, se o serviço aguardar o processamento completo da mensagem para removê-la da fila, uma falha entre a etapa do processamento e a remoção da mensagem fará com que a mensagem ainda esteja na fila, quando o serviço for reiniciado, e seja processada novamente.

Como já se mencionou anteriormente, o processamento da mesma mensagem várias vezes não é um problema se ela for uma consulta de saldo, mas o processamento de um saque duas vezes pode irritar o cliente. A única forma real de garantir que cada mensagem seja processada "exatamente uma vez" é fazer com que o processamento da mensagem e a exclusão da fila façam parte da mesma transação. Se houver uma falha no processamento, as alterações no processamento e a exclusão da mensagem serão retrocedidas, de forma que tudo volte a ser como era antes da recepção da mensagem.

Uma operação única de confirmação confirma as ações de recebimento e processamento da mensagem. Da mesma forma, se o processamento da mensagem gerar uma mensagem de saída, o "envio" da mensagem de saída deve fazer parte da transação, para evitar a situação em que o serviço é retrocedido, mas a mensagem é enviada assim mesmo. Esse tipo de processamento de mensagem é chamado de *mensagem transacional*. A maioria das infra-estruturas de mensagens confiáveis oferece suporte a mensagens transacionais.

Como as mensagens são armazenadas em um arquivo que não é o banco de dados, as mensagens transacionais do MSMQ requerem uma confirmação em duas fases para garantir que as duas partes da transação sejam confirmadas. Já que os comandos SEND e RECEIVE do Service Broker são comandos TSQL, as operações de mensagens e atualizações de dados em um serviço do Service Broker podem ser executadas na mesma conexão do SQL Server e fazer parte da mesma transação do SQL Server. Portanto, não há necessidade da confirmação em duas fases, e isso torna a implementação de mensagens transacionais do Service Broker muito mais eficiente do que a implementação do MSMQ.

Também nesse caso, as mensagens transacionais são necessárias para fazer um serviço não idempotente comportar-se como um serviço idempotente para eliminar os problemas causados pelas tentativas repetidas das mensagens. Se o serviço é idempotente por natureza, não há necessidade de mensagens transacionais. Se as mensagens transacionais não forem usadas, a pessoa que solicitou o serviço precisa estar preparada para receber a resposta várias vezes e, ocasionalmente, ao longo de um grande intervalo de tempo.

Confiabilidade dos Dados

Agora abordaremos o impacto dos dados sobre a confiabilidade do serviço. A maioria dos serviços acessa dados enquanto processa as mensagens de serviço portanto, a confiabilidade dos dados está estreitamente ligada à confiabilidade dos serviços.

Um dos aspectos diferenciados da execução assíncrona do serviço é que, em muitos casos, as mensagens de serviço são importantes objetos de negócio. No nosso serviço de caixa eletrônico, por exemplo, se as mensagens de alteração do saldo forem perdidas por causa de falhas, o saldo da conta não será alterado e o banco não perderá dinheiro. Por isso, armazenar mensagens no banco de dados para que tenham as mesmas proteções à confiabilidade, redundância e disponibilidade que o restante dos dados armazenados no mesmo faz muito sentido. Se as mensagens de alteração do saldo, no nosso exemplo, forem armazenadas no mesmo banco de dados das contas, elas só serão perdidas se as contas também forem. Os backups, backups de registro e os recursos de SAN (Rede de Armazenamento) usados para garantir que as informações das contas bancárias não se percam também se aplicam às mensagens de alteração de saldo, tornando a confiabilidade do serviço extremamente alta. Se os seus requisitos de confiabilidade das mensagens forem altos, o Service Broker e o BizTalk têm vantagens significativas quanto à confiabilidade, já que armazenam as mensagens no banco de dados.

O DBM (Espelhamento de Banco de Dados) é um dos novos recursos do SQL Server 2005 que pode aumentar a confiabilidade do serviço. O DBM proporciona confiabilidade mantendo uma cópia secundária do banco de dados que se mantém consistente quanto às transações do banco de dados primário, aplicando à cópia secundária todas as transações confirmadas no primário, antes de devolver o controle do serviço. Se o primário falhar, a cópia secundária do banco de dados poderá se tornar a primária em poucos segundos.

O Service Broker usa o espelhamento de banco de dados para melhorar a confiabilidade das mensagens. Se o banco de dados de contas for um par de bancos de dados em DBM, o banco de dados do Service Broker no caixa eletrônico abrirá conexões de rede para os bancos de dados primário e secundário e enviará mensagens ao primário. Se o banco de dados secundário se tornar o primário, o Service Broker será notificado imediatamente e as mensagens serão roteadas para o novo primário, sem interrupção nem intervenção do usuário.

Os serviços com intensa utilização de dados podem aproveitar outro recurso do SQL Server 2005 para melhorar a confiabilidade. A integração do CLR (Common Language Runtime) ao mecanismo do servidor do SQL Server significa que a lógica do serviço pode ser executada também no banco de dados. Portanto, para um serviço do Service Broker, lógica, mensagens, ambiente de execução, contexto de segurança e dados de serviço podem ficar no mesmo banco de dados.

Esse armazenamento em um único local apresenta várias vantagens em um sistema que tem altos requisitos de confiabilidade. Geralmente, os servidores de bancos de dados têm os recursos de hardware e software para manter a alta confiabilidade exigida por um banco de dados. Agora essa confiabilidade pode ser aplicada a todos os aspectos da implementação do serviço. No caso improvável de uma falha de sistema, é possível restaurar todo o ambiente do serviço para um estado consistente em termos de transações por meio dos recursos de recuperação de banco de dados. Os dados são salvos e, além disso, todas as operações em andamento são retrocedidas e reiniciadas no mesmo ambiente de segurança e execução em que estavam no momento da falha.

A natureza assíncrona e livremente acoplada das aplicações voltadas a serviços apresenta requisitos específicos de confiabilidade. Ao arquitetar serviços, o nível de confiabilidade de que o serviço precisa deve ser compreendido e levado em conta. A Microsoft oferece várias infra-estruturas para implementação de serviços, que oferecem diversos níveis de confiabilidade. A escolha da infra-estrutura correta envolve a correspondência entre o nível de confiabilidade necessário e os recursos da infra-estrutura. Os novos recursos do SQL Server 2005 proporcionam uma infra-estrutura de hospedagem de serviços que oferece níveis inéditos de confiabilidade a serviços que requerem níveis muito altos de confiabilidade.

Recursos

Arquitetura Orientada a Serviços
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

Revista MSDN
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/

Sobre o autor

Roger Wolter é arquiteto de soluções da equipe da Microsoft Architecture Strategy. Roger tem 30 anos de experiência em diversos aspectos do setor de informática, como trabalhos na Unisys, Infospan, Fourth Shift, e os últimos sete anos como gerente de programa da Microsoft. Seus projetos na Microsoft englobam o SQLXML, o SOAP Toolkit, o SQL Server Service Broker e o SQL Server Express. Seu interesse pelo Service Broker surgiu por causa de um sistema de fabricação baseado em mensagens no qual ele trabalhou. Ele também escreveu *The Rational Guide to SQL Server 2005 Service Broker Beta Preview* (Rational Press, 2005).

Um Modelo Flexível para a Integração de Dados

por Tim Ewald e Kimberly Wolk

Resumo

Na integração de sistemas, há vários desafios para arquitetos e desenvolvedores, e o setor vem se concentrando em XML, serviços da Web e SOA para resolver o problema da integração e em protocolos de comunicação particularmente no que diz respeito ao acréscimo de recursos avançados que oferecem suporte ao fluxo de mensagens em topologias de rede complexas. Entretanto, essa concentração em protocolos de comunicação desviou o foco do problema da integração de dados. Modelos flexíveis para combinar dados em sistemas diferentes são essenciais para uma integração bem-sucedida. Esses modelos são expressos em esquemas XML (XSD) nos sistemas baseados em serviços da Web, e instâncias do modelo são representadas como XML transmitido em mensagens no protocolo SOAP. No nosso trabalho sobre a arquitetura do MTPS (MSDN TechNet Publishing System), abordamos três armadilhas. Iremos ver o que são essas armadilhas e nossas soluções para elas, no contexto de um problema mais geral: a integração das informações do cliente.

A integração de sistemas apresenta uma ampla gama de desafios aos arquitetos e desenvolvedores. Nos últimos anos, o setor enfocou o uso de XML, serviços da Web e SOA (Arquitetura Orientada a Serviços) para resolver problemas de integração. Boa parte do trabalho feito nesse espaço se concentrou nos protocolos de comunicação, particularmente no acréscimo de recursos avançados projetados para dar suporte às mensagens que fluem em topologias complexas de rede. Embora essa abordagem tenha, sem dúvida, algum valor, todo esse trabalho com os protocolos de comunicação desviou o foco do problema da integração dos dados.

Dispor de modelos flexíveis para combinar dados em sistemas diferentes é essencial para um trabalho de integração bem-sucedido. Em sistemas baseados em serviços da Web, esses modelos são expressos em XSD. Instâncias do modelo são representadas como XML que é transmitido de um sistema para outro em mensagens SOAP. Alguns sistemas mapeiam os dados XML em bancos de dados relacionais; alguns não fazem isso. Do ponto de vista da integração, a estrutura desses modelos relacionais de bancos de dados não é importante. O importante é o formato do modelo de dados XML definido no XSD.

Existem três armadilhas nas quais os projetos de integração de dados baseados em serviços da Web costumam cair. Todas as três estão relacionadas ao modo de definição dos esquemas XML. Enfrentamos as três armadilhas em nosso trabalho na arquitetura do MSDN TechNet Publishing System, o sistema baseado em XML da próxima geração, que é a base do MSDN2. Analisaremos nossas soluções no contexto da integração de informações do cliente.

O Problema Essencial da Integração de Dados

Imagine que você trabalha em uma empresa grande. Sua empresa tem vários sistemas voltados para fora que são usados pelos clientes para realizar diversas tarefas. Por exemplo: um sistema oferece informações personalizadas sobre os produtos para usuários registrados que declararam interesses específicos. Outros sistemas oferecem ferramentas de gerenciamento de associação para clientes participantes do programa de parceiros. Um terceiro sistema rastreia clientes que se registraram para comparecer em futuros eventos. Infelizmente, todos os sistemas foram desenvolvidos separadamente um deles foi desenvolvido por uma empresa separada que a sua companhia adquiriu há um ano. Cada um desses sistemas armazena informações sobre clientes em formatos e locais diferentes. Essa configuração é um problema crítico para os negócios: não se tem uma visão unificada de um determinado cliente. Esse problema tem dois efeitos: primeiro, a experiência dos clientes é prejudicada, porque a única companhia com que eles fazem negócios os trata como pessoas diferentes quando usam sistemas diferentes. Por exemplo: um cliente que declarou o desejo de receber informações por email sobre um determinado produto sempre que ele fica disponível tem que declarar interesse nesse produto pela segunda vez quando se registra para determinados debates em um futuro evento patrocinado pela empresa.

> **"DISPOR DE MODELOS FLEXÍVEIS PARA COMBINAR DADOS EM SISTEMAS DIFERENTES É ESSENCIAL PARA UM TRABALHO DE INTEGRAÇÃO BEM-SUCEDIDO"**

A experiência dela seria mais conveniente se o sistema que a registrou para um evento futuro já tivesse conhecimento do seu interesse em um determinado produto.

Em segundo lugar, a empresa é prejudicada porque não tem uma compreensão integrada em relação aos clientes. Quantos clientes que são membros de um programa de parceiros também estão recebendo informações por email sobre os produtos? Nos dois casos, as divisões entre os sistemas com que o cliente trabalha estão limitando o nível de qualidade da resposta da empresa às necessidades dos clientes.

Não importa se essa situação ocorreu porque os sistemas foram projetados e desenvolvidos individualmente, sem considerar o contexto mais amplo onde operam ou porque grupos diferentes de sistemas integrados foram coletados por meio de fusões e aquisições. O problema continua: a empresa tem que integrar os sistemas para melhorar a experiência dos clientes e também o conhecimento a respeito de quem é o cliente e qual é a melhor forma de atendê-lo.

A abordagem mais comum para resolver esse problema é exigir que todos os sistemas adotem um único modelo canônico para um cliente. Um grupo de arquitetos se reúne e projeta o formato único da empresa para representar os dados do cliente em XML. O formato é definido usando um esquema escrito em XSD. Para permitir que os sistemas compartilhem dados no novo formato, uma equipe central cria um novo armazenamento que dá suporte a eles. A equipe do modelo de dados XSD e a equipe de armazenamento entregam a sua solução para todas as equipes responsáveis por sistemas que, de alguma forma, interagem com os clientes e exige que elas o adotem. A mudança essencial é ilustrada nas Figuras 1 e 2.

Figura 1 Três armazenamentos de dados separados, um por sistema.

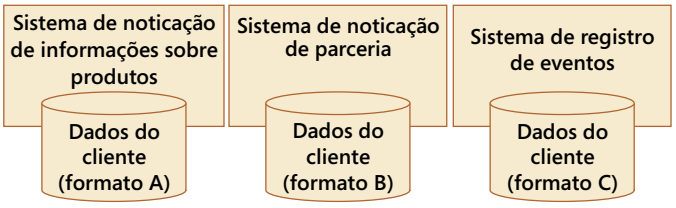
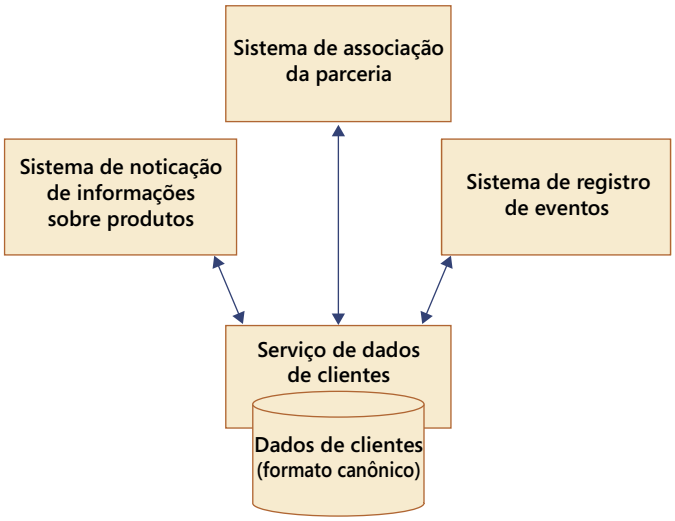


Figura 2 Um único formato e armazenamento de dados.



Cada sistema é modificado para usar o armazenamento de dados de base por meio de sua interface de serviços da Web. Os sistemas armazenam e recuperam informações do cliente como um XML que se enquadra no esquema do cliente. Todos os sistemas compartilham a mesma instância de serviço, o mesmo modelo de dados XSD e as mesmas informações XML.

Essa solução parece simples, elegante e boa, mas uma implementação ingênua normalmente falha por uma destas três razões: excesso na exigência de informações, falta de uma estratégia eficiente de versões e falta de suporte para uma extensão no nível do sistema.

Excesso na Exigência de Informações

A primeira possível causa de falha é o uso de um esquema e armazenamento que exigem informações demais. Quando se cria um serviço da Web simples para a integração ponto a ponto, tende-se a pensar nos dados de que o serviço em particular precisa. Define-se um contrato que exige o fornecimento de dados específicos. Quando um contrato é gerado a partir do código fonte, esses dados podem ocorrer de forma implícita. A maioria das ferramentas que mapeia a partir do código fonte para um contrato de serviço da Web trata os campos do tipo "valor simples" como elementos de dados obrigatórios, insistindo que o cliente os envie. Mesmo quando o contrato é criado à mão, ainda existe a tendência de tratar todos os dados como obrigatórios. Assim que o serviço determina (por meio de validação do esquema ou do código) que alguns dados obrigatórios não estão presentes, ele rejeita a solicitação. O cliente obtém uma falha de serviço.

Essa abordagem à definição de contratos de serviços Web é rígida demais e leva a sistemas que são fortemente acoplados. Qualquer alteração nos requisitos do serviço força uma alteração no contrato e nos clientes que o consomem. Para tornar esse acoplamento mais livre, é necessário fazer uma distinção entre a definição do formato dos dados que um serviço espera e os requisitos atuais de processamento de um sistema. Mais concretamente, os formatos de dados definidos pelo contrato devem tratar tudo o que não seja dados de identidade como opcional. A implementação do seu serviço deve impor requisitos de ocorrência internamente no tempo de execução (usando um esquema de validação dedicado ou código). Deve ser o mais "condescendente" possível quando os dados não estiverem presentes em uma solicitação do cliente e "degradar com classe" (ou seja, não deixar de exibir as informações quando há elementos faltando).

No exemplo das informações do cliente, é fácil pensar em casos nos quais alguns sistemas querem trabalhar com os clientes mas não têm informações completas do cliente à disposição. Por exemplo: o sistema que registra o interesse de um cliente em um determinado produto pode coletar apenas o nome e o endereço de email preferencial do cliente. O sistema de registro de eventos, por sua vez, pode captar também as informações de endereço e cartão de crédito. Se um modelo comum de dados de cliente exigir que cada registro válido do cliente inclua nome, endereço de email e informações de cartão de crédito, nenhum sistema poderá adotá-lo sem coletar mais dados que o necessário ou fornecer dados falsos. Ao fazer com que todos os dados que não sejam a identidade (número da cédula de identidade, endereço de email etc.) sejam opcionais, a adoção de um modelo de dados se torna mais fácil, porque os sistemas podem simplesmente fornecer as informações que têm.

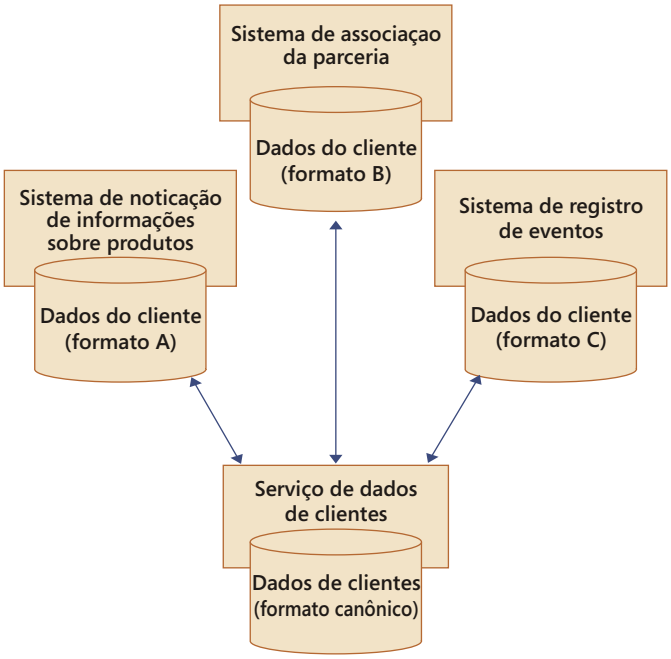
Ao fazer a separação entre o formato de dados e os requisitos de ocorrência, você facilita o gerenciamento da mudança na implementação de um serviço único. Isso também é critico quando você define um esquema XML comum para ser usado por vários serviços e clientes. Se houver um excesso de informações obrigatórias, cada sistema que quiser usar o modelo de dados pode estar sem algumas informações obrigatórias. Isso dá a cada sistema a opção de não adotar o modelo compartilhado e armazenar ou fornecer dados falsos (freqüentemente, o valor padrão de um tipo simples de linguagem de programação). Qualquer uma dessas opções pode ser considerada uma falha.

Você ganha bastante flexibilidade para que os sistemas adotem o modelo flexibilizando quase que completamente os requisitos de ocorrência do esquema. Cada sistema pode contribuir com todos os dados que tiver à sua disposição, o que facilita muito a adoção de um esquema XML comum. O preço a pagar é que os sistemas que recebem os dados devem ter o cuidado de se certificar de que os dados realmente necessários estejam presentes. Se não estiverem presentes, eles devem reagir adequadamente, obtendo mais dados do usuário ou outro armazenamento, simplificando o comportamento ou — somente na pior hipótese — gerar uma falha. O que você está fazendo, na verdade, é desviar de algumas das restrições que você normalmente colocaria em um esquema XML do seu código, onde seriam verificadas no tempo de execução. Esse desvio permite alterar essas restrições sem revisar o esquema compartilhado.

Falta de uma Estratégia Eficiente de Versões

A segunda possível causa de falha é a falta de uma estratégia de versões. Independentemente do tempo e do esforço despendidos na definição de

Figura 3 Uma combinação de armazenamentos (consulte as Figuras 1 e 2).



um esquema XML antecipadamente, o esquema terá que ser alterado com o passar do tempo. Se os esquemas, o armazenamento compartilhado que dá suporte a eles e todos os sistemas que os usam tiverem que mudar de uma vez para outra versão, não haverá possibilidade de sucesso. Alguns sistemas terão que aguardar as alterações necessárias, porque os outros sistemas não estão em condições de adotar uma revisão. Conseqüentemente, alguns sistemas serão forçados a fazer um trabalho extra e inesperado, já que os outros sistemas precisam adotar uma nova revisão. Essa abordagem é injustificável.

Para resolver esse problema, é necessário adotar uma estratégia de versão que permita que o esquema e ao armazenamento continuem funcionando, independentemente da velocidade com que os outros sistemas adotam suas revisões. Essa solução parece simples e é desde que você tenha uma concepção correta em relação aos esquemas XML.

Os sistemas que se integram usando um esquema XML comum o consideram como um contrato. O fato de diminuir a quantidade de dados obrigatórios, tornando os elementos opcionais, facilita a aceitação do contrato porque os sistemas estão se comprometendo a realizar menos. No caso da versão, também é necessário permitir que os sistemas façam mais, mas sem mudar o espaço de nome do esquema. Na prática, isso significa que um sistema sempre deve produzir dados XML com base na versão do esquema com o qual ele foi desenvolvido. Devem sempre consumir dados baseados na mesma versão com informações adicionais. Essa definição é uma variação da lei de Postel: "Seja liberal naquilo que aceita, e conservador naquilo que envia". Pode-se dizer que essa idéia está por trás de todas as tecnologias bem-sucedidas de sistemas distribuídos e certamente, de todas as tecnologias de acoplamento livre. Se adotar essa abordagem, você poderá ampliar um esquema sem atualizar os clientes.

No exemplo do cliente, uma atualização do esquema e do armazenamento pode adicionar suporte a um elemento adicional opcional que capta o nome de solteira da mãe do usuário, para fins de segurança. Se os sistemas que funcionam com a versão antiga gerarem registros de clientes sem essa informação, não haverá problema, já que o elemento é opcional . Se eles enviarem esses registros a outros sistemas que exigem essas informações, a solicitação poderá falhar, mas não há problema nisso também. Se os novos sistemas enviarem dados do cliente aos sistemas antigos incluindo o nome de solteira da mãe, também não haverá problema, porque eles estão projetados para ignorar isso.

Felizmente, muitos kits de ferramentas para serviços da Web fornecem suporte a esse recurso diretamente nos encaixes do marshalling dirigidos a esquemas. Certamente, os mapeadores de objetos XML do .NET (tanto o confiável XmlSerializer quanto o novo XmlFormatter/ DataContract) processam dados extras sem problemas. Alguns kits de ferramentas para Java também fazem isso, assim como os frameworks que fornecem suporte às novas especificações de JAX-WS 2.0 e JAXB 2.0. Tendo isso em mente, é muito fácil adotar essa abordagem.

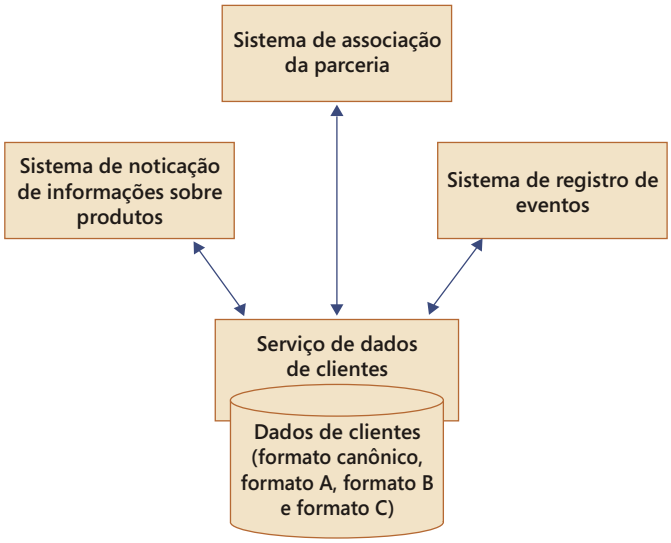
O único verdadeiro problema desse modelo é que ele introduz definições múltiplas de um determinado esquema, cada uma representando uma versão diferente. Considerando-se um dado — por exemplo: um fragmento de XML captado de uma mensagem enviada pelo meio físico — é impossível responder a pergunta: "Esse dado é válido?" Só é possível responder a pergunta da validade em relação a uma versão específica do esquema. A incapacidade de determinar com certeza se um determinado dado é válido é um problema para a depuração e, talvez, também para a segurança. No caso dos modelos de dados descritos com um esquema XML, é possível responder uma pergunta diferente e mais interessante: "Esse dado é suficientemente válido?"

Essa pergunta é o que realmente importa para os sistemas, e você pode respondê-la usando as informações de validade que a maioria dos validadores fornece, que é um caminho razoável a seguir quando a validação do esquema é obrigatória, e isso pode ser implementado com os atuais frameworks de validação do esquema.

Falta de Suporte para uma Extensão no Nível do Sistema

A terceira possível causa de falha é a falta de suporte para extensões do esquema específicas para o sistema. A estratégia de versão baseada na idéia de que a definição de um esquema muda com o tempo é necessária para promover a adoção, mas não é suficiente. Embora libere os sistemas da necessidade de adotar imediatamente a versão mais recente do esquema, não faz nada para ajudar os sistemas que estão aguardando atualizações específicas do esquema. Os atrasos nas revisões também podem deixar um esquema caro demais para o sistema. A solução desse problema é permitir que os sistemas que adotam o mesmo esquema o estendam com suas próprias informações adicionais. As informações de extensão podem ser armazenadas localmente em um armazenamento no nível do sistema (consulte a Figura 3).

Figura 4 O mesmo armazenamento (consulte a Figura 3) com formatos de dados em um armazenamento compartilhado



Nesse caso, cada sistema é modificado para gravar os dados no cliente no armazenamento dedicado, usando o seu próprio modelo de dados, e no armazenamento compartilhado, usando o esquema canônico. O sistema também é modificado para ler dados do cliente a partir do armazenamento dedicado e também do compartilhado. Dependendo do que ele encontra, ele sabe se a empresa e o sistema já conhecem o cliente. A Tabela 1 resume as três possibilidades.

O sistema pode usar essas informações para decidir quantas informações do cliente ele precisa armazenar. Se o cliente é novo para a empresa, o sistema adiciona ao sistema canônico a maior quantidade possível de informações. Essas informações ficam disponíveis para outros sistemas que trabalham com clientes. O sistema também pode armazenar dados no armazenamento dedicado, para suprir as suas próprias necessidades. Esse modelo pode ser expandido ainda mais, para que os dados específicos do sistema sejam armazenados também no armazenamento compartilhado (consulte a Figura 4).

Essa solução permite que os sistemas se integrem usando dados de extensão que estão além do escopo do esquema canônico. Para funcionar corretamente, é necessário que o armazenamento e os outros sistemas tenham visibilidade em relação aos dados de extensão. Em outras palavras, ele não pode ser opaco. A forma mais fácil de resolver esse problema é fazer com que os próprios dados de extensão sejam XML. O sistema que fornece os dados define um esquema para os dados de extensão, para que os outros sistemas possam processá-los de forma confiável. O armazenamento compartilhado faz um acompanhamento dos esquemas, para poder garantir que os dados de extensão sejam válidos, mesmo que não precise saber explicitamente o que a extensão contém. No caso mais extremo, um determinado sistema pode optar por armazenar um registro

"A INCAPACIDADE DE DETERMINAR COM CERTEZA SE UM DETERMINADO DADO É VÁLIDO É UM PROBLEMA PARA A DEPURAÇÃO E, TALVEZ, TAMBÉM PARA A SEGURANÇA"

de cliente inteiro em um formato específico do sistema, como dados de extensão XML. Outros sistemas que entendem esse formato podem usá-lo. Os sistemas que não o entendem dependem da representação canônica.

Quando os sistemas são independentes, cada um deles controla o seu próprio destino. Eles podem captar e armazenar quaisquer informações necessárias, no formato e local que preferirem. A passagem para um único esquema e armazenamento em comum muda isso. Se a adoção de dados XML em comum restringir a liberdade do sistema para proporcionar a funcionalidade necessária, você estará condenado ao fracasso.

O uso de uma combinação de dados de extensão tipo XML em um armazenamento compartilhado ou armazenamento no nível do sistema aumenta a complexidade, porque você tem que manter a sincronia dos dados. Porém, também proporciona uma grande flexibilidade. É possível alinhar sistemas em torno de qualquer combinação de esquemas canônicos e esquemas alternativos que você quiser. Você pode ir mudando para o formato XML com o passar do tempo, mas terá sempre a flexibilidade de desviar desse formato para cumprir novos requisitos. Essa liberdade extra vale muito no incerto mundo empresarial.

Outro benefício sutil desse modelo é o fato de permitir que a equipe que define o esquema em comum diminua a velocidade ao trabalhar nas revisões. Os sistemas podem usar os dados de extensão para atender

Tabela 1 Os três casos possíveis de dados de clientes

Registro no amazenamento compartilhado	Registro no armazenamento do sistema	Significado
Não	Não	O cliente é novo para a empresa. Coleta todos os dados comuns e específicos para o sistema.
Sim	Não	O cliente já é conhecido pela empresa, mas é novo para o sistema. Coleta dados específicos para o sistema.
Sim	Sim	O cliente já é conhecido pela empresa e pelo sistema.

novos requisitos entre as revisões. A equipe que trabalha no modelo canônico pode buscar essas informações para contribuir no processo de revisão. Esse círculo de feedback ajuda a garantir que as mudanças de modelo sejam motivadas por requisitos reais do sistema.

Diminuição dos Riscos

Diversas organizações estão trabalhando na integração de sistemas usando os dados XML que são descritos por meio de esquemas XML e trocados por meio de serviços da Web. Nessa explanação, apresentamos três causas comuns de falha nesses projetos de integração centrados nos dados: excesso na exigência de informações, falta de uma estratégia eficiente de versões e falta de suporte para extensões no nível do sistema. Para diminuir esses riscos:

- Faça com que os elementos do esquema sejam opcionais e codifique requisitos de ocorrência específicos do sistema como parte da implementação do sistema.
- Crie sistemas que produzam dados de acordo com a sua versão do esquema compartilhado, mas consumam de forma que os sistemas possam adotar revisões de esquema a velocidades diferentes, sem alterar o espaço de nome do esquema.
- Permita que os sistemas estendam esquemas compartilhados com seus próprios dados para cumprir novos requisitos, sem depender das revisões de modelo de dados.

Todas essas soluções se baseiam em uma idéia central: integrar de forma bem-sucedida e sem sacrificar a agilidade de que os sistemas precisam para concordar o menos possível e, ainda assim, cumprir a sua função. Tudo isso funciona? A resposta é sim. Essas técnicas são fundamentais para o projeto do MSDN/TechNet Publishing System, que é a base do MSDN2.

Recursos

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

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

Sobre os Autores

Tim Ewald é uma das principais arquitetas da Foliage Software Systems, que ajuda os clientes a projetar e criar aplicativos que vão desde TI empresarial até dispositivos médicos. Antes de entrar para a Foliage, Tim trabalhou na Mindreef, um dos maiores fornecedores de ferramentas de diagnóstico para serviços da Web; antes disso, Tim foi líder dos gerentes de programa no MSDN, onde trabalhou com Kim Wolk como co-arquiteta do MTPS, o mecanismo de publicação baseado em XML e serviços da Web que é a base do MSDN2. Tim é palestrante e escritora de renome internacional.

Kim Wolk é gerente de desenvolvimento do MSDN e é a força que move o MTPS, o mecanismo de publicação baseado em XML e serviços da Web que é a base do MSDN2. Antes disso, ela atuou como co-arquiteta e principal desenvolvedora do MTPS. Antes de entrar para a MSDN, Kim atuou por muitos anos como consultora, tanto independente quanto ligada à Microsoft Consulting Services, onde trabalhou em uma ampla gama de sistemas empresariais de missão crítica.

Serviços Autônomos e Agregação de Entidades Empresariais

por Udi Dahan

Resumo

As empresas de hoje dependem de sistemas e aplicativos heterogêneos para funcionar. Cada um desses sistemas gerencia seus próprios dados e, com freqüência, não os expõe explicitamente para consumo externo. Muitos desses sistemas dependem dos mesmos conceitos básicos como cliente e funcionário e, conseqüentemente, essas entidades foram definidas em vários lugares de formas ligeiramente diferentes. A agregação de entidades incorpora a necessidade de negócio de se ter, em um só lugar, uma visão em 360 graus dessas entidades. Entretanto, essa necessidade de negócio é somente um sintoma de um problema maior: o alinhamento entre o negócio e a TI. As SOAs (Arquiteturas Orientadas a Serviços) foram consideradas como a "cola" que aproximaria a TI dos negócios, mas esse modismo já está passando. Abordaremos formas concretas de usar serviços autônomos para transformar a forma de desenvolvimento dos sistemas, a fim de se assemelhar mais aos processos de negócio e suprir necessidades imediatas de agregação de entidades.

O termo SOA ganhou popularidade no ano passado e se tornou o jargão do momento. Hoje em dia tudo é orientado aos serviços, habilitado para SOA ou "a chave do seu sucesso na SOA." O setor segue tendo dificuldades em relação à definição de serviço; entretanto, várias propriedades dos serviços parecem ser bem aceitas. Os princípios da Microsoft para a orientação aos serviços definem quatro dessas propriedades: os serviços são autônomos; têm limites explícitos ; compartilham o contrato e o esquema, e não a classe ou o tipo; a compatibilidade do serviço é baseada em diretivas.

Embora esses princípios atuem como diretrizes arquitetônicas importantes para o desenvolvimento de softwares complexos, é obvio que há mais a ser dito sobre isso. Muitos aspectos do tempo de execução dos serviços, como escalabilidade, disponibilidade e força, não são abordados pela orientação a serviços. Os serviços autônomos enfocam exatamente esses aspectos do tempo de execução.

Autonomia do Serviço

A expressão "serviços autônomos" parece ser uma paráfrase simples do primeiro princípio, mas essas duas expressões têm significados muito diferentes. Dizer que os "serviços são autônomos" significa que as equipes que desenvolvem serviços cooperativos podem operar de forma independente — até certo ponto, é claro. Ao considerá-lo juntamente com o princípio sobre contrato e esquema, fica claro que não há necessidade de dependências binárias entre essas equipes. O desenvolvimento de cada serviço pode ser feito em uma plataforma diferente, com linguagens e ferramentas diferentes. Por outro lado, um serviço autônomo é um serviço cuja capacidade de funcionar não é controlada nem inibida por outros serviços.

A palavra *"autônomo"* tem várias definições, como: que governa a si mesmo, que controla a si mesmo independentemente, auto-suficiente e livre de controle e restrição externa. Com base nessas definições de autonomia, analisaremos dois tipos de interação de serviços: *comunicação síncrona e assíncrona*.

Na Figura 1, pode-se ver que o Serviço A precisa manter ativamente os recursos de tempo de execução (o segmento que chama) até que o Serviço B responda. O tempo que o Serviço A leva para responder uma única solicitação depende de sua interação com o Serviço B. O Serviço A é afetado se a rede está lenta ou se o Serviço B está indisponível ou lento. Portanto, nesse caso, o Serviço A não parece "livre de restrição interna".

Outro problema a se considerar é o acoplamento. Embora os Serviços A e B possam estar acoplados livremente, já que foram desenvolvidos separadamente por equipes diferentes em plataformas diferentes, que compartilhavam apenas um arquivo WSDL, pode-se ver que estão rigidamente acoplados quanto ao tempo. Pode-se notar esse acoplamento temporal porque o tempo que o Serviço A leva para responder a uma solicitação inclui o tempo de processamento do Serviço B. É exatamente esse acoplamento que faz com que as falhas indesejáveis do Serviço B influenciem o Serviço A.

"MUITOS ASPECTOS DO TEMPO DE EXECUÇÃO DOS SERVIÇOS, COMO ESCALABILIDADE, DISPONIBILIDADE E FORÇA, NÃO SÃO ABORDADOS PELA ORIENTAÇÃO A SERVIÇOS"

Existem duas formas de quebrar esse acoplamento temporal. Uma delas é fazer o Serviço A consultar o Serviço B quanto ao resultado. Infelizmente, a consulta causa uma carga indevida no Serviço B (de acordo com o número de clientes e o intervalo de consulta), e os consumidores obtêm essa resposta além do tempo disponível. A Figura 2 mostra a complexidade dessa solução.

Se continuarmos a análise, veremos que a geração de um novo segmento, ou até mesmo o uso do pool de segmentos para processar a consulta de cada solicitação enviada irá esgotar rapidamente os recursos do Serviço A. Uma solução com melhor desempenho (e até mesmo mais complexa) envolveria um único segmento que gerencia a consulta para todas as solicitações enviadas. Esse segmento transmitirá os resultados a outro segmento à medida eles se tornam disponíveis, e esse segmento finalizaria o processamento da solicitação original. Seria bom termos em mente a navalha de Occam antes de seguir por esse caminho.

Figura 1 Comunicação síncrona

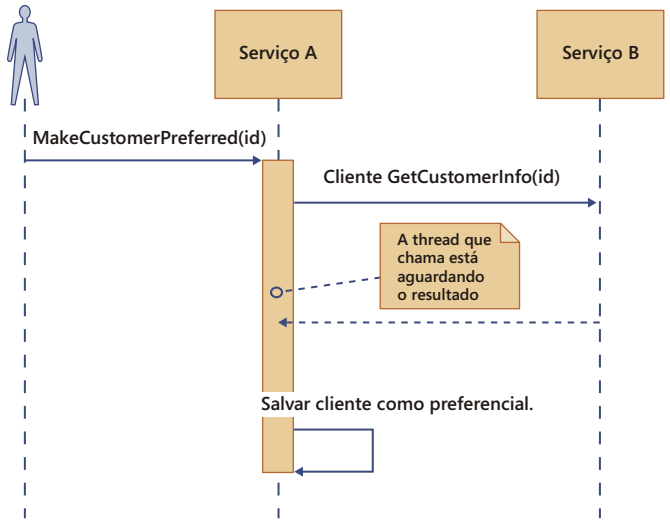


Figura 2 Comunicação síncrona com consulta

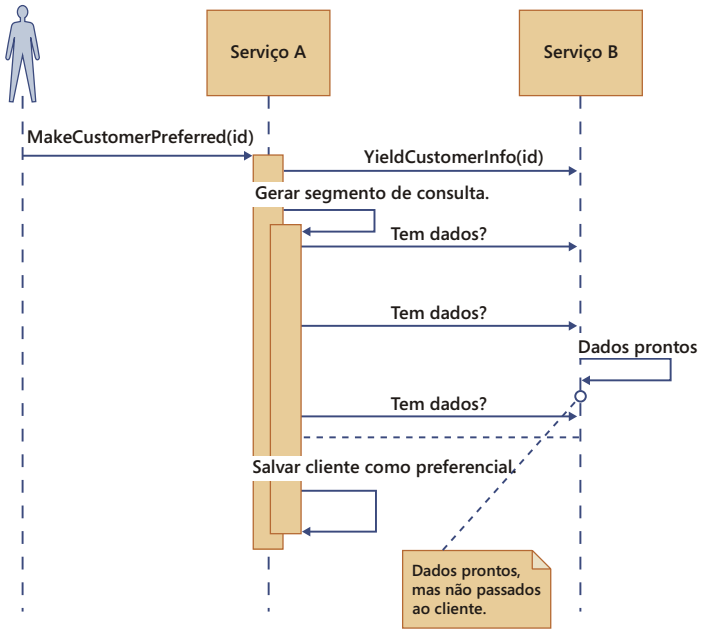
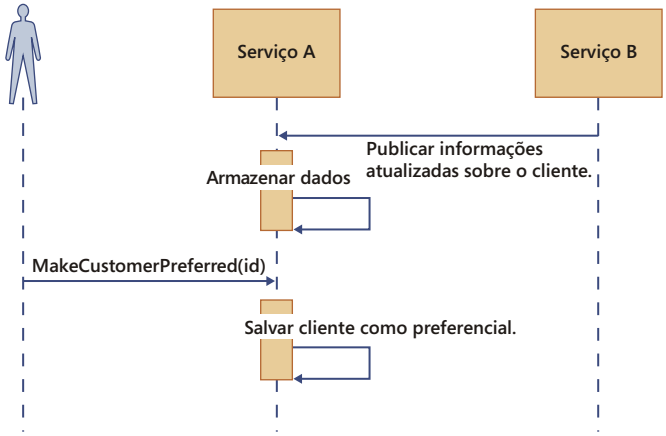


Figura 3 Comunicação assíncrona



Outra solução para o problema do acoplamento temporal que evita os problemas citados anteriormente é usar a comunicação assíncrona entre esses serviços (consulte a Figura 3). Nesse caso, o Serviço A se inscreve nos eventos publicados pelo Serviço B sobre alterações no estado interno. Quando esses eventos ocorrem, o Serviço A armazena os dados que ele considera relevantes. Dessa forma, quando o Serviço A recebe uma solicitação, ele não depende mais de partes externas para o processamento, não afetando a sua disponibilidade. Observe que a carga do Serviço B é até mesmo mais baixa do que no exemplo original de comunicação síncrona, já que ele não recebe mais essas solicitações. A infra-estrutura moderna de publicação/inscrição e mensagens pode manter a carga quase constante, independentemente da quantidade de consumidores. Com a comunicação síncrona, os dados se tornam mais atualizados, já que o Serviço A recebe os dados em um tempo muito mais próximo do momento em que o Serviço B os disponibilizou. Não é necessário abrir mão da atualização dos dados (em conseqüência do intervalo de consultas) para ter uma carga baixa.

Limites Técnicos e Replicação de Dados

Embora o segundo princípio pareça óbvio à primeira vista, a natureza de um limite não é nada óbvia. Os limites dos Serviços A e B dos exemplos anteriores apresenta alguma diferença nos casos síncronos e assíncronos? Parece que não, mas há uma grande diferença técnica: as transações.

Para processar uma solicitação única adequadamente, nos casos em que a solicitação causa alteração dos dados, o processamento da solicitação deve ser feito dentro do contexto da transação, para que o estado do serviço permaneça constante. Se esse serviço tiver que interagir com outros serviços para processar a solicitação e, conseqüentemente, esses serviços também alterarem os seus dados, essas alterações devem ocorrer dentro do contexto da transação original?

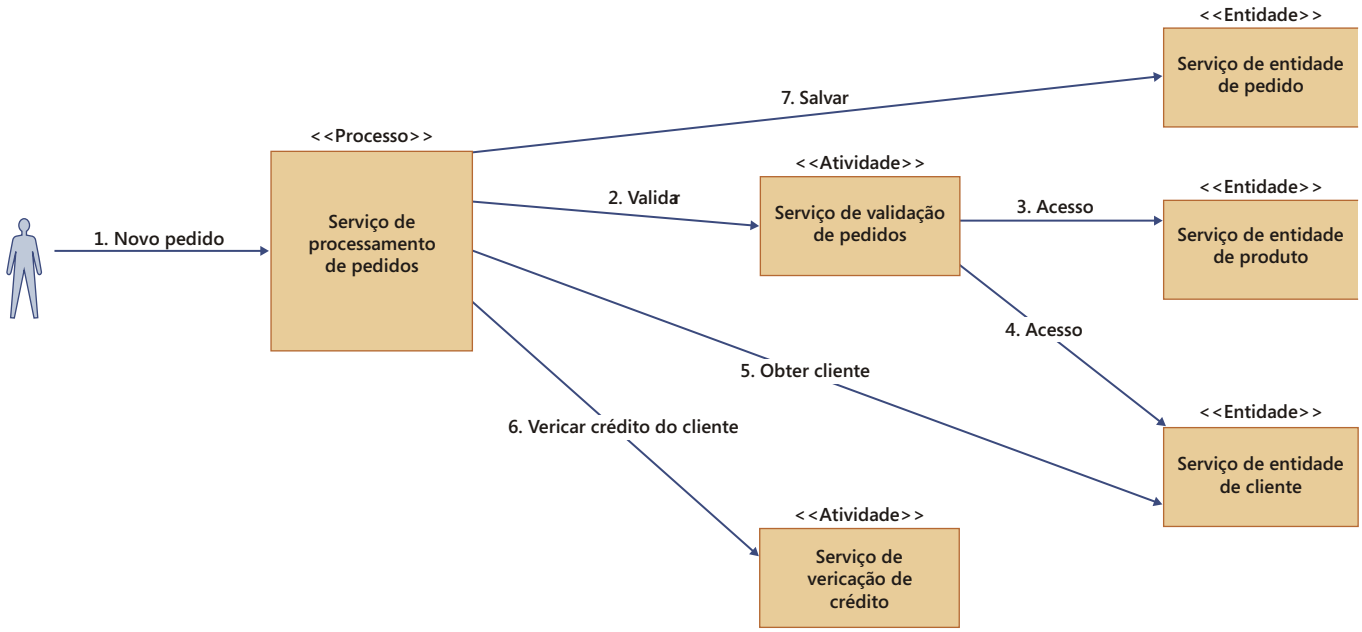
Quando a interação entre serviços é síncrona, a divisão da responsabilidade entre os mesmos pode, freqüentemente, requerer que alterações em todo o serviço sejam realizadas dentro de uma única transação. Quando a interação entre os serviços é assíncrona (ou síncrona com consulta), evitamos totalmente a alteração de dados em outros serviços portanto, não é necessário fazer com que as transações ultrapassem as fronteiras do serviço. Obviamente, se a transação que inicia o serviço travasse os recursos de outros serviços, seriam necessários altos níveis de confiança entre todos os serviços envolvidos, e isso dificultaria a distinção de onde terminou um serviço e começou outro.

Vamos definir os serviços autônomos. É evidente que os serviços autônomos vão muito além das comunicações de baixo nível, englobando vários aspectos, como confiança e confiabilidade. Entretanto, vimos que a restrição das interações entre os serviços e as mensagens assíncronas levou a arquitetura por um caminho no qual os serviços autônomos e livremente acoplados ficaram cristalizados. Na verdade, os serviços autônomos parecem expandir a orientação a serviços (ou restringir seus graus de liberdade) acrescentando um novo princípio: *um serviço interage com outros serviços usando padrões de comunicação assíncrona.*

Observe que, embora um serviço possa consumir outros serviços de forma síncrona, esse consumo não significa necessariamente que ele não possa expor uma interface síncrona. O Google e a Amazon fazem exatamente isso. Os serviços da Web que eles expõem são síncronos, mas isso não influencia sua autonomia.

À primeira vista, parece que o uso de comunicações do tipo publicação/inscrição leva à duplicação de dados entre os serviços, algo semelhante ao que acontece ao fazer a replicação de dados. As técnicas de replicação de dados, como ETL (“Extrair, Transformar e Carregar”) ou a transferência

Figura 4 Interações entre serviços de entidades, processos e atividades



de arquivos, processam a transferência de dados entre fontes de dados de baixo nível, como bancos de dados e diretórios. Essa transferência ignora construtos lógicos de nível mais alto para gerenciar esses dados com coerência, e isso freqüentemente leva à duplicação desses mesmos construtos lógicos na extremidade do consumidor da transferência.

"A AGREGAÇÃO DE ENTIDADES REPRESENTA A NECESSIDADE DE NEGÓCIO DE TER UMA VISÃO GLOBAL DOS DADOS DE TODA A EMPRESA"

Os dados que fluem por uma interação de publicação/inscrição se comportam de forma diferente. Quando um serviço publica uma mensagem, essa mensagem deve fazer parte do contrato do serviço — esse contrato é independente do esquema de armazenamento de dados de base. O processo de construção da mensagem — recuperar os dados adequados do armazenamento de dados e transformá-los para o esquema da mensagem — passa por todas as camadas do serviço. Os serviços que consomem essas mensagens não precisam implementar a mesma lógica. Além disso, a decisão sobre o tempo da publicação da mensagem é uma decisão lógica, para a qual foi escrito um código; não é um detalhe de baixo nível relacionado ao tempo em que o script de ETL foi programado para ser executado.

A diferença mais importante entre a replicação simples de dados e a interação de serviços autônomos é que o serviço consumidor decide salvar somente os dados de que ele precisa. Não há mais um "back door" no banco de dados do serviço. Quando o serviço consumidor recebe a mensagem que foi publicada, os dados dentro da mensagem não ignoram nenhuma camada do serviço até chegarem ao banco de dados (consulte os Recursos). Ao usar esses tipos de interação assíncrona entre serviços, freqüentemente constatamos que os nossos serviços tendem a ser maiores e ter maior granularidade, e muitas vezes contêm seus próprios bancos de dados, hospedados em seus próprios servidores ou data centers. Ao manter os contextos de transação limitados ao escopo de um único serviço, as responsabilidades desse serviço tendem a se expandir e chegar ao nível de uma função ou departamento de negócio o limite natural encontrado na área dos negócios. Esse efeito é bastante compreensível quando consideramos os níveis de acoplamento nos diversos níveis do negócio. Os departamentos são livremente acoplados entre si, colaborando sem ter um conhecimento profundo do funcionamento interno de cada um. Os grupos que são internos ao departamento freqüentemente precisam ter um conhecimento muito mais profundo sobre o funcionamento de grupos paralelos para cumprir a sua função.

Pode-se ver que esse estilo arquitetônico não contradiz, de forma alguma, nenhum dos quatro princípios embora os tipos de serviço mais difundidos que encontramos ao usar a orientação a serviços não sejam mais adequados.

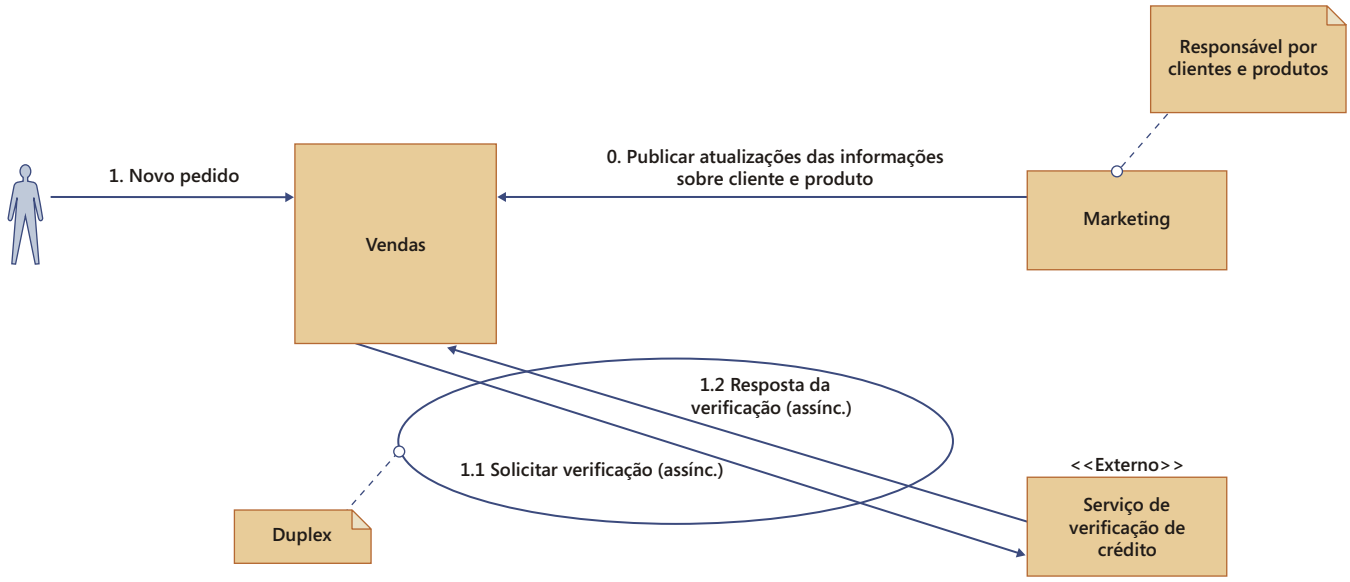
Armadilhas Comuns dos Limites

Serviços de processos e serviços de atividades já foram propostos como forma de realizar a orientação a serviços. Os serviços de processo gerenciam processos de negócio de longo prazo. Os serviços de atividade gerenciam operações atômicas que encapsulam a interação com mais de um serviço. Os serviços de entidade gerenciam a interação com uma única entidade de negócio. Nesse modelo, a agregação de entidades ocorre no nível do serviço de entidade. O problema dessa opção de limites de serviço se manifesta nos fluxos assíncronos entre os serviços (consulte a Figura 4).

Embora seja perfeitamente possível modelar a mesma interação com mensagens assíncronas (especificamente em relação a um serviço externo de verificação de crédito), a importância da separação do processamento de pedidos em três serviços não fica evidente. É improvável que equipes diferentes trabalhem em cada um dos serviços de pedido ou que sejam executados em plataformas diferentes. O acoplamento rígido entre esses serviços é inerente. Todos eles trabalham com o processamento de pedido; o fato de não ter uma classe de pedido em comum provavelmente causaria duplicação do código, mas essa duplicação vai contra o terceiro princípio da orientação a serviços. Há apenas uma conclusão: não é possível separar nossos serviços por meio dos limites de processo/atividade/entidade. Considere o resultado da modelagem desse processo de negócio usando serviços autônomos (consulte a Figura 5).

Os serviços autônomos presentes na Figura 5 representam uma divisão possível, correlacionada a uma determinada organização; empresas

Figura 5 Interações entre serviços autônomos



diferentes teriam grupos diferentes que seriam responsáveis por processos de negócio distintos. Nesse ponto, há duas diferenças importantes a serem observadas. A primeira é que o serviço de vendas armazena os dados do cliente e do produto dos quais ele precisa internamente, para que ele já tenha todos os dados necessários quando precisar processar um pedido. A segunda diferença é que a divisão do processamento de pedidos entre serviços separados não ocorre nesse caso. Embora seja provável que o serviço de vendas tenha camadas e componentes diferentes para processar as várias etapas do processamento de pedidos, e talvez algum tipo de mecanismo de workflow para gerenciar essas etapas, esses aspectos são detalhes da implementação do serviço.

A duplicação do código, que ocorre quando se segue o princípio de "esquema/classe" no caso anterior, não ocorre dessa vez simplesmente porque o princípio não se aplica a um limite de serviço. O problema, nesse caso, é que todos eles são a mesma entidade (pedido) provavelmente com os mesmos campos. Antes que o serviço de entidade passe a salvar o pedido, ele também precisa realizar a validação: um código que já foi escrito para validar o serviço de atividade. Se não podemos mais usar serviços de entidade, como e onde a agregação de entidades ocorre ao usar serviços autônomos?

Requisitos da Agregação de Entidade em Nível de Negócio

Antes de entrar nos detalhes técnicos da solução, é preciso melhor definir o problema. Como já foi dito anteriormente, a agregação de entidades representa a necessidade dos negócios de ter uma visão global dos dados de toda a empresa. Diferentes departamentos de negócio requerem elementos de dados diferentes de uma determinada entidade, mas raramente requerem todos os dados dessa entidade. Esse caso é um daqueles em que um departamento de negócio requer dados que pertencem a outro departamento. Por exemplo: o departamento de marketing precisa saber o valor total dos pedidos por cliente, por trimestre — dados que são gerenciados pelo departamento de vendas. Nesse caso, estamos agregando dados de dois sistemas em um desses sistemas — um estilo de agregação de entidades diferente daquilo que normalmente é considerado como agregação de entidades; porém, é o caso mais comum que se vê nessa área.

Integração OLTP Intersistemas. Começaremos analisando um exemplo concreto. Em um cenário típico de processamento de pedidos, temos dois sistemas: um aceita pedidos provenientes do nosso site; o outro é um sistema de CRM (Gestão de Relacionamento com o Cliente) criado na própria empresa. O marketing tem um novo requisito, segundo o qual os "clientes preferenciais" devem ter um desconto de 10% em todos os pedidos. O cliente preferencial é definido como um cliente que vive nos Estados Unidos e fez negócios de pelo menos US$ 50.000 com a empresa no último trimestre, mas espera-se que essa definição mude.

Após analisar esse requisito, vemos que ele tem duas partes: quem é o cliente preferencial, e o que fazemos com essa informação? A primeira decisão a ser tomada está relacionada a limites e responsabilidades: qual sistema será responsável pelos clientes preferenciais e qual sistema será responsável pelo que vamos fazer com os clientes preferenciais? Nesse caso, não há motivos para o sistema de CRM não assumir a primeira responsabilidade e o sistema de processamento de pedidos assumir a segunda. A próxima decisão a ser tomada refere-se ao modo de interação entre esses sistemas: síncrono ou assíncrono.

Integração OLTP Síncrona. A primeira forma de lidar com esse requisito é adicionar ao sistema de processamento de pedidos um método de consulta que recupere os clientes cujo total de pedidos atingiu pelo menos X ao longo de um período Y. Esse método permite trabalhar com várias quantias e períodos de tempo, à medida que o departamento de marketing muda suas regras. O sistema de CRM atenderia as solicitações sobre a identificação dos clientes preferenciais consultando o sistema de processamento de pedidos (consulte a Figura 6).

Integração OLTP Autônoma. Na segunda abordagem, consideramos cada sistema como um serviço autônomo que notifica os clientes externos sobre eventos importantes. Nesse caso, as interações entre

"O GERENCIAMENTO DE UM ÚNICO SERVIÇO QUE TRABALHA COM AS NECESSIDADES DE AGREGAÇÃO DE ENTIDADES DE TODA A EMPRESA TEM UM CUSTO/BENEFÍCIO MUITO MELHOR QUE O GERENCIAMENTO DE VÁRIOS SERVIÇOS DE ENTIDADES"

os serviços são mais controladas pelos eventos. Na Figura 7, quando o sistema de processamento de pedidos precisa saber se um determinado

Figura 6 Agregando dados de sistemas diferentes sem serviços autônomos

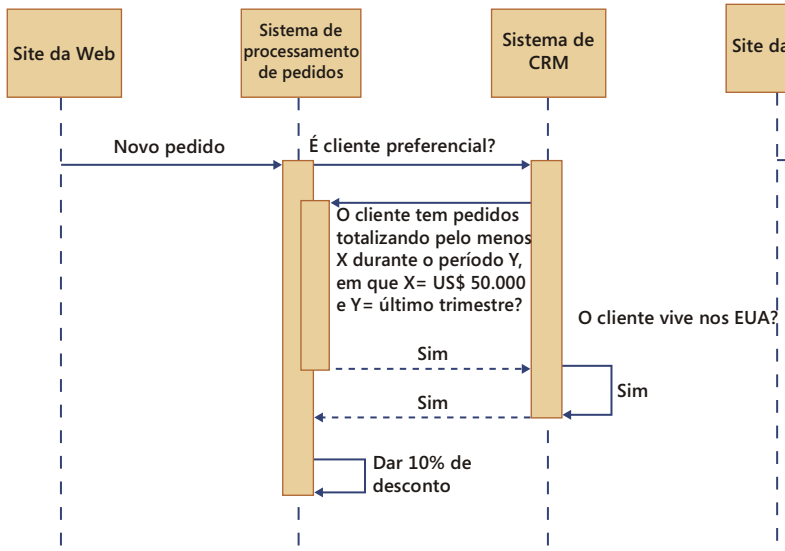


Figura 7 Agregando dados de sistemas diferentes com serviços autônomos

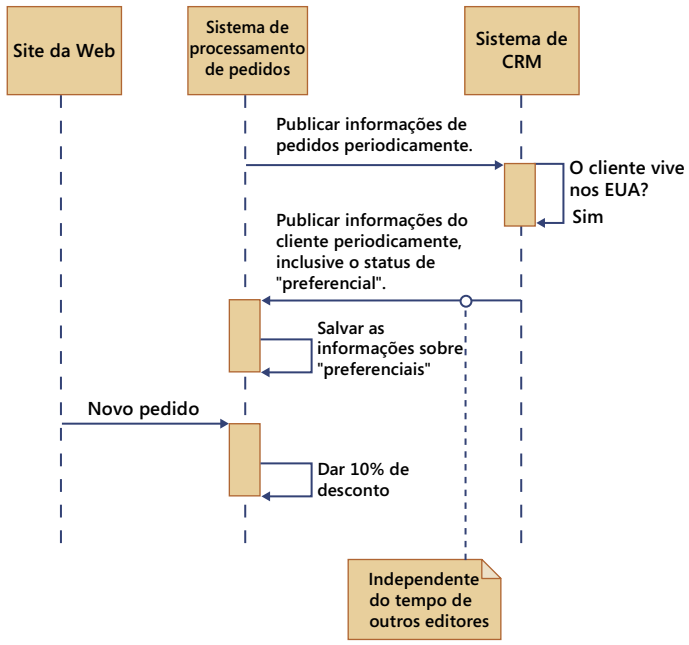


Figura 8 Alterações feitas quando os serviços autônomos não são usados

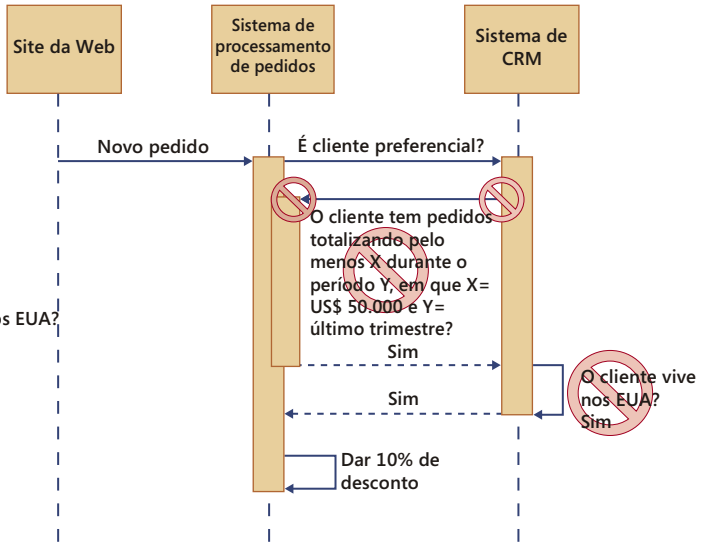
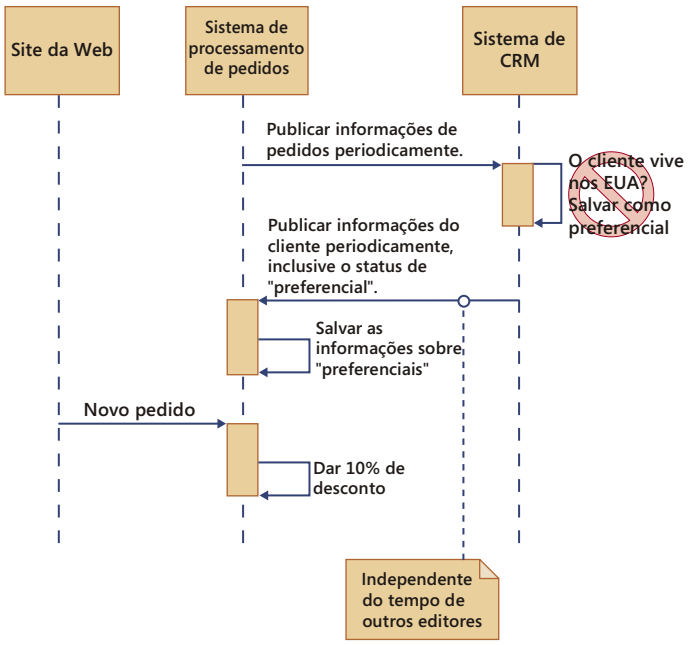


Figura 9 Alterações feitas quando os serviços autônomos são usados



cliente é preferencial, ele não precisa se comunicar com o sistema de CRM. Isso também vale para os dados do pedido e o sistema de CRM. Os dados desses dois sistemas foram agregados propositalmente. Vamos explorar esse exemplo um pouco mais, à medida que o departamento de marketing muda as regras de definição de clientes preferenciais. Agora, clientes preferenciais são os que vivem em toda a América do Norte e fizeram pelo menos três pedidos no último trimestre, cada um totalizando US$ 15.000 ou mais.

Na nossa primeira abordagem, a consulta original que adicionamos já não é relevante. Agora precisamos dar suporte a outro tipo de pergunta (identificar os clientes com pelo menos X pedidos ao longo do período Y, cada pedido totalizando Z ou mais, em que X = 3, Y = último trimestre e Z = US$ 15.000), alterando a interface do sistema de processamento de pedidos, parte de sua implementação (para dar suporte à nova interface) e o código que a ativa no sistema de CRM (consulte a Figura 8). Na segunda abordagem, as únicas alterações que precisamos fazer são internas ao sistema de CRM. Não é necessário modificar nem a interface nem a implementação do sistema de processamento de pedidos (consulte a Figura 9).

O termo agregação de entidades dá idéia de um processo ativo de coleta de dados em fontes diferentes e fundi-los para formar um conjunto coeso. A dinâmica da primeira abordagem se ajusta muito bem a esse processo, mas a segunda abordagem não se ajusta. Entretanto, é evidente que a segunda abordagem mantém um nível mais baixo de acoplamento entre esses dois serviços, à medida que os requisitos mudam. Leve essa questão em consideração ao projetar serviços; é necessário que ocorra um acoplamento livre entre os serviços, tanto nos dados quanto na lógica.

Agregação de Entidades para Inteligência de Negócios

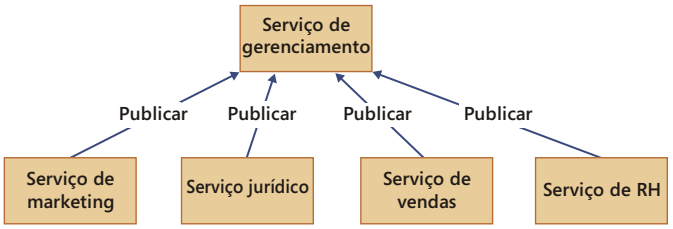
No caso anterior, os dados agregados eram críticos para o funcionamento dos departamentos de negócios envolvidos, que participam do processamento de transações no *dia-a-dia*. Entretanto, a agregação de entidades é abordada com maior freqüência no contexto do gerenciamento — a empresa quer ter uma visão dos dados empresariais em 360 graus. Esse contexto é muito diferente do anterior, já que os dados agregados são usados principalmente para suporte a decisões e inteligência de negócios — em outras palavras, principalmente em cenários "somente leitura".

A instituição de um serviço de gerenciamento (consulte a Figura 10) é uma forma simples e eficiente de modelar esse contexto. Esse serviço não realiza o RPM (Gerenciamento de Provisionamento de Recursos) nem as operações para outros serviços; em vez disso, reúne os dados publicados por todos os outros serviços e os armazena em um formato otimizado para os seus cenários de utilização.

A análise histórica é outro requisito de negócio comum que ocorre no contexto da agregação de entidades. Embora a manutenção de dados históricos sobre produtos do passado que já não são oferecidos talvez não faça sentido para o serviço de marketing, os usuários do serviço de gerenciamento podem considerar importante a comparação dos valores de vendas e lucros de produtos atuais e do passado. O gerenciamento dessas tendências históricas seria responsabilidade do serviço de gerenciamento.

Um dos benefícios do uso de um serviço de gerenciamento em vez de serviços de entidades está exatamente na área de gerenciamento de operações. O gerenciamento de um único serviço que trabalha com as necessidades de agregação de entidades de toda a empresa tem um custo/benefício muito melhor que o gerenciamento de vários serviços de entidades um para cada entidade que precisa ser agregada. Alterações internas em qualquer um desses serviços, que não afetem o contrato, podem até mesmo não influir no serviço de gerenciamento. As alterações no contrato que podem afetar várias entidades provocam alterações correspondentes somente no serviço de gerenciamento, e não em cada um dos serviços de entidades que anteriormente agregavam essas entidades. Ao que parece, os serviços podem ser a nova unidade de reutilização na empresa, mas isso não está de acordo com o princípio da autonomia; além disso, as restrições na forma de interação com um serviço o tornam desagradável. Por exemplo: apesar da impressão de que o serviço de gerenciamento poderia proporcionar facilmente o rastreamento das etapas de auditoria e armazenamento para todos os serviços, essa opção pode acabar com a autonomia desses serviços. Se o serviço de gerenciamento cair por algum motivo, os outros serviços não devem ter permissão para continuar a processar sem auditoria.

Figura 10 Relações entre o serviço de gerenciamento e outros serviços autônomos



Além disso, um serviço autônomo não poderia confiar em outro serviço para fornecer essa capacidade central; isso não quer dizer que você não pode encapsular o rastreamento das etapas de auditoria (ou outras funcionalidades específicas) em um componente que seja reutilizado por todos os serviços. As questões de cumprimento de regulamentos devem ser resolvidas dentro de cada serviço.

Ao reconhecer que os requisitos da agregação de entidades OLTP e OLAP são diferentes, tivemos condições de identificar duas soluções separadas, porém simples, usando um único paradigma de comunicação. Os padrões de mensagens assíncronas permitem a criação de serviços autônomos, com acoplamento livre, que se assemelham mais aos processos de negócio que eles modelam. Conseqüentemente, muitas vezes, basta uma alteração local em um único serviço para responder a mudanças nos requisitos de negócio. Essas alterações em pequena escala não afetam os contatos entre serviços e podem ser realizadas com a certeza de que outros sistemas não serão afetados e, portanto, podem ser feitas em menos tempo. O alinhamento da TI com os negócios está mais ligado ao entendimento e às comunicações interpessoais do que à tecnologia, mas isso não significa que a tecnologia não pode ajudar.

Recursos

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/dng rfsoachallenges-entityaggregation.asp*

Sobre o autor

Udi Dahan é arquiteto de soluções, ganhador do prêmio Microsoft MVP, renomado especialista em desenvolvimento em .NET e também arquiteto chefe de TI e gerente da linha de produtos C4ISR na KorenTec. Udi é conhecido como uma das principais autoridades em SOA de Israel e presta consultoria sobre arquitetura e projeto de sistemas de missão crítica, desenvolvidos em larga escala, em todo o país. Sua experiência engloba tecnologias relacionadas a sistemas de comando e controle, aplicativos em tempo real e serviços de Internet de alta disponibilidade. Para obter mais informações, acesse *www.UdiDahan.com*.

Replicação de Dados como um Antipadrão de SOA Corporativo

por Tom Fuller e Shawn Morgan

Resumo

À medida que os aplicativos são idealizados e entregues completamente em qualquer organização, o desejo de usar a replicação dos dados como uma estratégia de integração de "rápida correção" geralmente ganha terreno. Esse caminho de mínima resistência gera redundância e inconsistência. Aplicar regularmente esse antipadrão dentro de uma empresa dissolve os objetivos originais de longo prazo de uma SOA (Arquitetura Orientada a Serviços). No entanto, a replicação de dados pode ser vista de forma positiva ou negativa, dependendo do contexto no qual aparece. Para oferecer com êxito uma estratégia orientada a serviços, os arquitetos empresariais precisam se espelhar nos sucessos e nas falhas de outros arquitetos. Os padrões de arquitetura, quando descobertos e documentados, podem fornecer a melhor técnica para gerenciar os trabalhos onerosos de influentes mudanças em sua empresa. Usaremos os dois, um padrão e um antipadrão, para descrever como a replicação dos dados pode influenciar na sua arquitetura empresarial.

O caminho em direção à orientação de serviços ainda está na fase da infância. Para que a SOA alcance os majestosos objetivos visualizados pela indústria, os arquitetos precisarão contar com as práticas recomendadas documentadas. É aí que os padrões e os antipadrões ajudarão a criar uma ponte na lacuna entre a conceituação e a realidade. Os padrões representam uma estratégia comprovada que pode ser repetida para alcançar os resultados esperados. Os padrões e os antipadrões da arquitetura de software fornecem exemplos documentados de sucessos e falhas na tentativa de aplicar essas estratégias na TI. Por meio da identificação bem-sucedida das abstrações arquitetônicas, a SOA começará a localizar estratégias de implementação que assegurem a entrega bem-sucedida.

Descreveremos o motivo pelo qual a replicação dos dados, quando utilizada como um padrão fundamental na SOA de sua empresa, pode conduzir a uma arquitetura complicada e cara, e proporcionar uma estratégia de "refatoração" que se move da replicação dos dados (antipadrão) para os serviços diretos (padrão). Em seguida, faremos uma breve explicação de onde a replicação de dados pode ser utilizada para resolver com êxito alguns problemas específicos nos domínios (replicação como um padrão) sem comprometer outras iniciativas da arquitetura empresarial. Os arquitetos vão extrair desta discussão uma estratégia para facilitar a mudança longe da malevolência na arquitetura.

Para promover consistência, a indústria de software estabeleceu um modelo para documentar os padrões de design. Esse modelo, no entanto, parecia insuficiente para descrever os antipadrões de arquitetura e suas eventuais soluções "refatoradas'. Uma coleção híbrida das seções de padrão de diversos recursos foi reunida para ajudar a definir uma estrutura para o padrão e o antipadrão descritos aqui (consulte os Recursos). A Tabela 1 mostra o superconjunto de seções e se elas se aplicam ou não a um padrão, um antipadrão ou a ambos.

Replicação de Dados: Um Antipadrão de Arquitetura SOA

Comecemos com uma descrição detalhada de como a replicação dos dados é um antipadrão quando aplicada em uma SOA, utilizando o modelo de antipadrão para explicar as particularidades do contexto e as forças que validam essa arquitetura. Um exemplo ilustra os problemas ao aplicar esse antipadrão. Para finalizar, você verá a solução "refatorada" e os benefícios que podem ser vistos quando o padrão for aplicado no lugar do antipadrão.

Contexto. Esse antipadrão descreve um cenário comum encontrado pelos arquitetos ao tentarem criar soluções empresariais de SOA em um ambiente distribuído. Na transição do ambiente de mainframe para o ambiente distribuído comum, ocorreu uma mudança no gerenciamento de nossos principais dados importantes (os dados que conduzem nossa empresa diariamente). Decidimos criar nossos novos aplicativos distribuídos utilizando um modelo de silo, no qual um aplicativo é definido como um conjunto de funcionalidade de negócios, exposto pela interface de usuário, e essa funcionalidade de negócios é criada dentro de seu próprio contexto. Essa decisão encoraja as arquiteturas que favorecem o isolamento extremo e a agregação desnecessária da entidade.

"PARA PROMOVER CONSISTÊNCIA, A INDÚSTRIA DE SOFTWARE ESTABELECEU UM MODELO PARA DOCUMENTAR OS PADRÕES DE DESIGN. ESSE MODELO, NO ENTANTO, PARECIA INSUFICIENTE PARA DESCREVER OS ANTIPADRÕES DE ARQUITETURA E SUAS EVENTUAIS SOLUÇÕES REFATORADAS".

Forças. Esse antipadrão ocorre em uma SOA devido ao conforto que os arquitetos e os projetistas têm com esse modelo. Mesmo quando explicado como um antipadrão dentro do contexto de SOA, os arquitetos e os projetistas continuam usando esse antipadrão pelos seguintes motivos:

- **O arquiteto está familiarizado com a estratégia de replicação de dados**. As técnicas utilizadas para replicar os dados passaram por ajustes refinados desde os primeiros dias do desenvolvimento orientado por lote. Esse ajuste refinado cria um nível de conforto para a arquitetura que vem projetando sistemas com replicação de dados por anos, usando as transferências baseadas em arquivo ou a extração, a transformação e o carregamento (ferramentas ETL -

Tabela 1 Correlacionando as seções aos padrões e antipadrões

Seção	Descrição	Padrão	Antipadrão
Nome	Um nome abreviado útil que pode ajudar a descrever rapidamente o padrão/antipadrão.	x	x
Contexto	As informações de apoio interessantes onde o padrão/antipadrão será aplicado.	x	x
Forças	Existem inúmeros motivos pelos quais essa solução está sendo usada. Com muita freqüência em um antipadrão, esses motivos são constituídos de concepções equivocadas ou falta de conhecimento.	x	x
Solução	Esta seção descreve a solução proposta para o problema com base no contexto e nas forças documentados e representa a estratégia ou a inteligência do padrão.	x	
Solução ruim (Antipadrão)	Esta seção descreverá o antipadrão ou a solução para um problema que parece trazer benefícios, mas que, na realidade, gerará conseqüências negativas quando empregado		x
Conseqüências	As conseqüências são os efeitos colaterais da solução implementada. No caso de um antipadrão, as conseqüências sempre devem ser constituídas de impactos negativos que pesam sobre o benefício do próprio padrão.	x	x
Solução refatorada	Todo antipadrão deve ter um oposto que, na verdade, seria uma solução ou padrão, que pode ser documentado como padrões.		x
Benefícios	Os benefícios serão os mesmos para um padrão documentado e a solução refatorada de um antipadrão.	x	x
Exemplo	Um exemplo do mundo real de onde o padrão/antipadrão é aplicado para destacar/resolver um problema.	x	x

Extrair, Transformar e Carregar). Parece simples o suficiente para continuar utilizando essa estratégia sem parar.

- **O arquiteto está preocupado com o desempenho de um serviço exposto pelo sistema de dados principais**. Por exemplo, acessar o armazenamento de dados remotamente por meio dos serviços pode diminuir o andamento do novo sistema sobre uma solução que usa um banco de dados local, que é o cenário de um para vários.
- **Cada serviço separado pode ter visões ligeiramente diferentes da entidade**. Combinar as diferentes visões em uma entidade agregada e manter a autonomia de serviço são tarefas tidas como muito difíceis; assim, os dados dos serviços são replicados em um armazenamento, que é o cenário de dados de vários para um. O arquiteto está preocupado que a durabilidade do novo sistema fique comprometida. Por exemplo, se o novo sistema acessar os dados principais por meio de um serviço existente na rede, as falhas na rede ou as falhas no sistema de dados principais poderiam provocar inatividade no novo sistema. Ter os dados disponíveis localmente parece atenuar essa preocupação.
- **Os aplicativos são construídos utilizando um modelo de silo**. Um aplicativo é definido como um conjunto de funcionalidades de negócios, exposto por uma interface de usuário, e essa funcionalidade de negócios é criada dentro do seu próprio contexto sem que um arquiteto empresarial ajude a orientar os desenvolvedores de aplicativos na reutilização dos serviços disponíveis.
- **Os recursos para fornecer uma solução usando a replicação de dados são abundantes**. As habilidades necessárias para implementar uma solução que use a replicação de dados normalmente são fáceis de serem localizadas. Criar aplicativos que aproveitem os serviços novos ou existentes demanda um conjunto de habilidades mais raro, exigindo alguém com conhecimento profundo nos sistemas distribuídos, serviços da Web, orientação a objetos e outros modernos mecanismos para fornecer soluções à sua empresa.

Encobrindo as Linhas de Controle

Solução ruim (antipadrão). Para realizar esse antipadrão, as informações de dados principais são replicadas da fonte de dados principais para um novo banco de dados criado para expor a funcionalidade do novo aplicativo (consulte a Figura 1). Essa exposição pode ser realizada por meio de vários mecanismos diferentes, como o uso de uma ferramenta ETL ou por meio de mecanismos mais básicos como uma transferência de arquivo dos dados de um sistema a outro. As informações de dados principais também podem ser aumentadas pelos novos sistemas, adicionando mais atributos de entidade à entidade original.

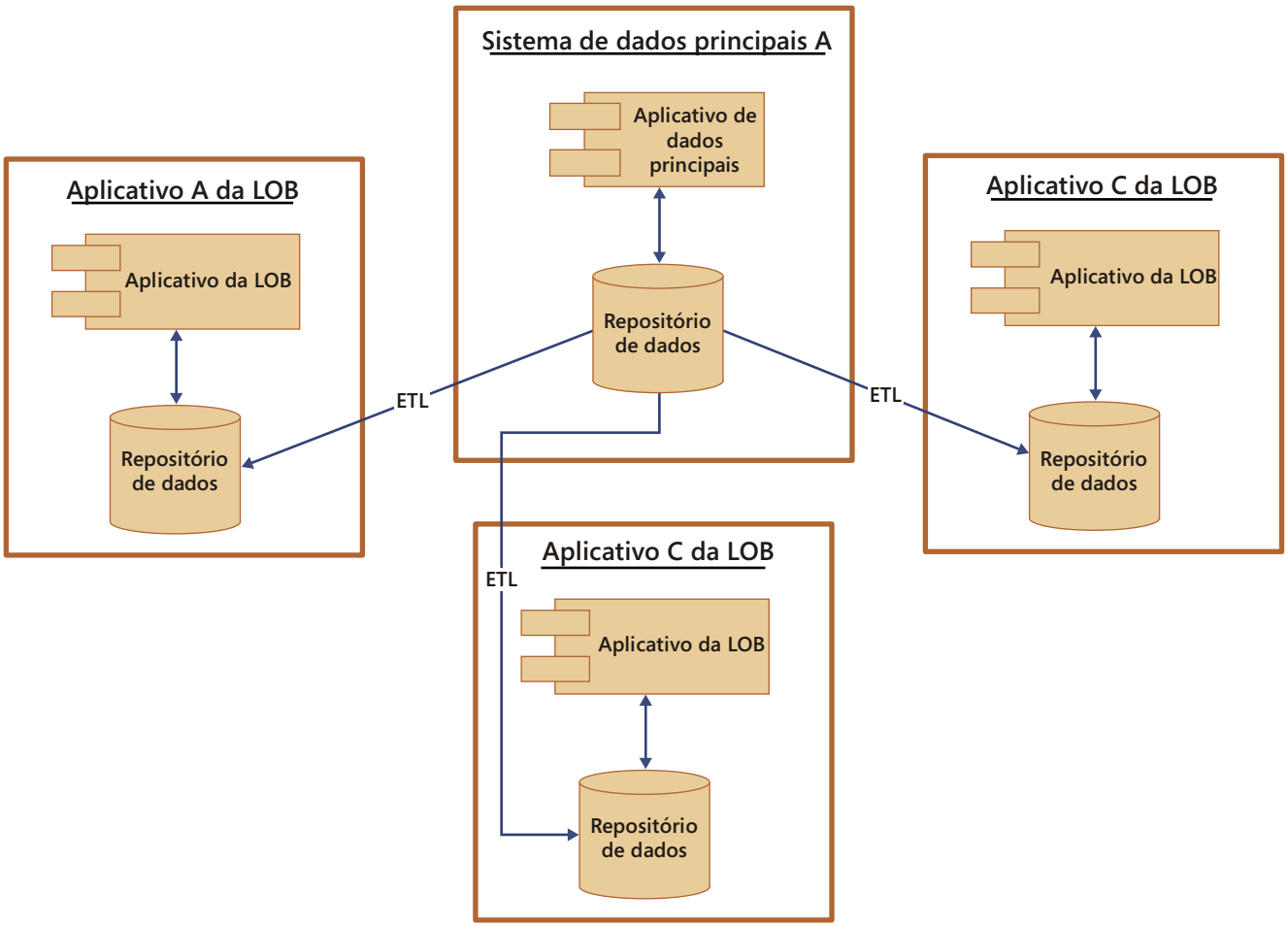
Conseqüências. As conseqüências desse padrão, aplicadas incorretamente em uma SOA, não são imediatamente visíveis. Quando os dados são replicados do sistema de dados principais proprietários, de uma maneira fractal, para vários bancos de dados de aplicativos diferentes, a orientação de seu serviço fica muito difícil de ser gerenciada. Os dados replicados se tornam penetrantes na empresa. A responsabilidade obscura para os dados encobre as linhas de controle. Existem vários serviços para manipular os mesmos dados (ou ligeiramente aumentados). A "visão comercial" dos dados se torna discrepante.

Essa responsabilidade obscura para os dados e a perda de controle sobre os dados principais incomodam qualquer arquiteto. Conforme os dados são proliferados e os aplicativos começam a adicionar seus próprios rodízios aos dados, a complexidade começa a ser montada. Os novos campos adicionados aos dados, que também podem ser importantes para a empresa, talvez não sejam adicionados de volta ao sistema de dados principais. Evidentemente, com as linhas de tempo e os orçamentos reduzidos da maioria dos projetos empresariais, uma equipe de projeto raramente dedicará o esforço extra na criação de novos elementos de dados dentro do sistema de dados principais, expondo esses novos elementos por meio de serviços estendidos fora do sistema de dados principais ou criando uma camada de serviços de agregação (substituindo assim o serviço inicial de dados principais).

Na realidade, eles podem estender os serviços para fora de seus sistemas, o que vai expor esses novos dados principais. Além disso, eles podem criar novos serviços que imitam algumas das funcionalidades dos sistemas de dados principais para expor os dados replicados para seus novos sistemas. Essa exposição cria uma grande dor de cabeça para os arquitetos empresariais que a gerenciam. O arquiteto agora precisa impedir que os novos sistemas consumam os serviços expostos usando os dados replicados, o que gera confusão dentro da arquitetura empresarial da propriedade verdadeira do objeto de negócios.

A replicação também leva consigo uma enorme responsabilidade no sistema de replicação para duplicar a lógica comercial do sistema de dados principais. Várias vezes os dados são manipulados pelo sistema de dados principais antes de serem impostos como um serviço a outros

Figura 1 Proliferação dos dados principais conduz à propriedade distribuída de diversas permutações.



aplicativos. Nesses casos, o método rápido e certeiro da replicação dos dados para o novo sistema leva consigo o fardo de também replicar a lógica comercial usada para manipular os dados. A menos que seja feita uma boa análise do sistema de dados principais, essa lógica pode não ser implementada nunca ou pode ser implementada incorretamente. Um cenário ainda mais perigoso é aquele no qual não existe inicialmente nenhuma lógica comercial cercando a recuperação dos dados principais.

"OS BENEFÍCIOS DE UMA SOA SÓ SERÃO TOTALMENTE PERCEBIDOS POR UMA ORGANIZAÇÃO QUE SEJA CONHECEDORA DA MUDANÇA DE TENDÊNCIAS NECESSÁRIA PARA FORNECER SOLUÇÕES QUE USEM NOVOS PADRÕES DE ARQUITETURA"

Em vez disso, a lógica é adicionada posteriormente a um projeto que precisa atualizar o sistema de dados principais. Esse cenário pode ser penoso para uma empresa que tenha replicado os dados várias vezes. Agora, cada sistema deve ser analisado para que as mudanças requeridas no sistema atendam às novas demandas dos dados empresariais.

Redundâncias Difundidas

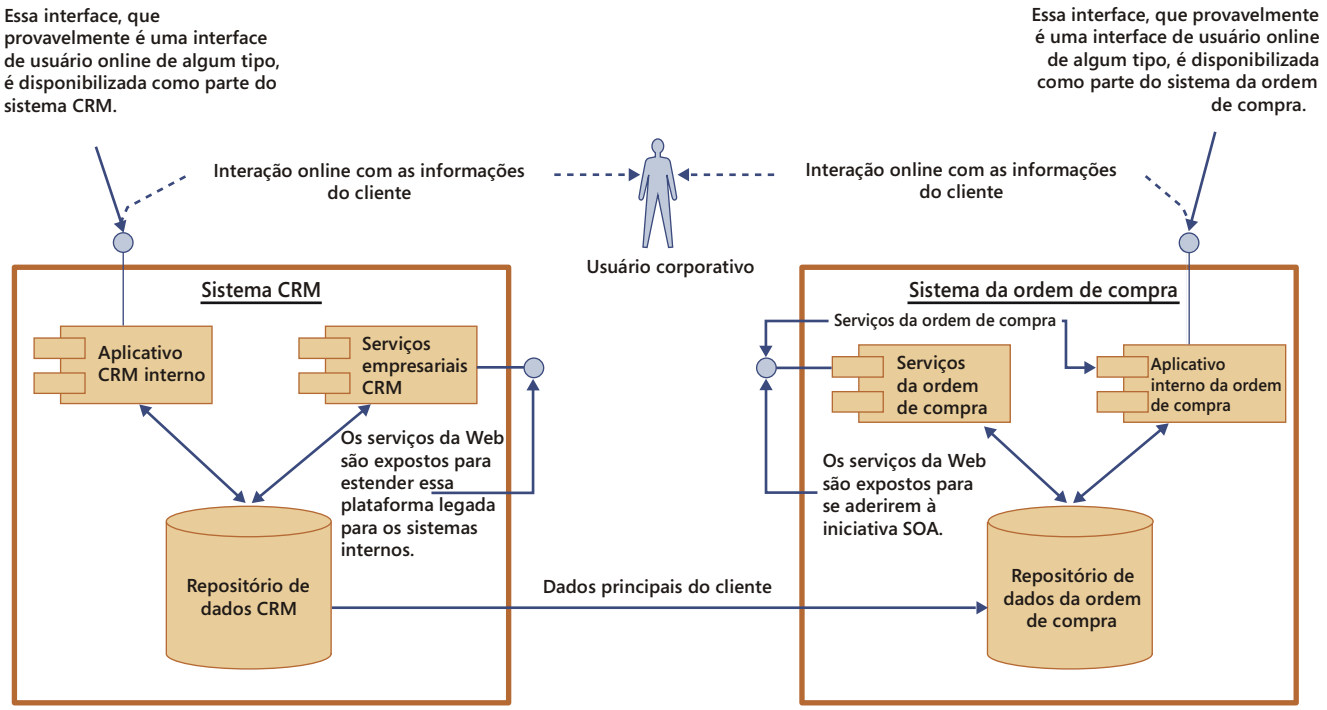
Geralmente é fácil rejeitar as conseqüências do uso da replicação dos dados, pois externamente elas parecem superficiais. Os problemas como espaço em disco, ativos reutilizáveis e redução de trabalho parecem ser gerenciáveis com ou sem se mover rumo à orientação de serviço. A conseqüência que assombra a todos é a capacidade de minimizar a complexidade e fornecer soluções que possam se adaptar rapidamente para serem alteradas. A replicação de dados adiciona complexidade, fragilidade e inflexibilidade devido às redundâncias difusas que cria em sua arquitetura empresarial.

Vejamos um exemplo. Um novo aplicativo está sendo criado para cuidar do gerenciamento de uma ordem de compra. A arquitetura exige que os serviços de negócios sejam criados para o novo sistema e que a interface de usuário, seja uma implementação de portal ou baseada na Web, consuma esses serviços de negócios.

Os requisitos de negócios ditam que o sistema deve executar as funções de criação, modificação e visualização das ordens de compra. Parte da ordem de compra é de informações do cliente, que ficam armazenadas em um sistema CRM (Gestão de Relacionamento com o Cliente). O sistema CRM expõe os serviços para recuperar as informações do cliente, mas a decisão é tomada pelo arquiteto para replicar os dados principais do cliente para os sistemas do pedido de compra com base nas forças discutidas anteriormente.

Um requisito dos novos sistemas de ordem de compra é criar um número de conta para cada cliente. Esses novos dados são anexados aos dados do cliente replicados no banco de dados local do sistema da OC. A interface de usuário exibe as informações do cliente por meio do uso de um novo serviço da Web criado pelo sistema de ordem de compra vertical. Outros serviços de negócios são desenvolvidos

Figura 2 Redundância de dados no antipadrão



para criação, modificação e visualização das ordens de compra que são então consumidas pela interface de usuário do sistema da OC. Qualquer manipulação de dados ou manuseio que o sistema de dados principais execute quando um usuário acessar seus serviços de dados deve ser replicada no novo sistema. Todas as mudanças nesses dados em andamento também devem se replicadas para o novo sistema, bem como todas as alterações em qualquer lógica comercial que acessar esses dados.

O diagrama mostrado na Figura 2 deixa claro que a replicação de dados resulta em serviços duplicados. Cada aplicativo agora possui sua própria visão da entidade do cliente, com sua maneira própria de acessar os dados por meio de um serviço que foi criado para esse propósito. O sistema da ordem de compra copia e estende os dados do cliente por meio de seu próprio conjunto de serviços, que gera efetivamente redundância e confusão para quaisquer sistemas futuros que pretendam consumir os serviços do cliente.

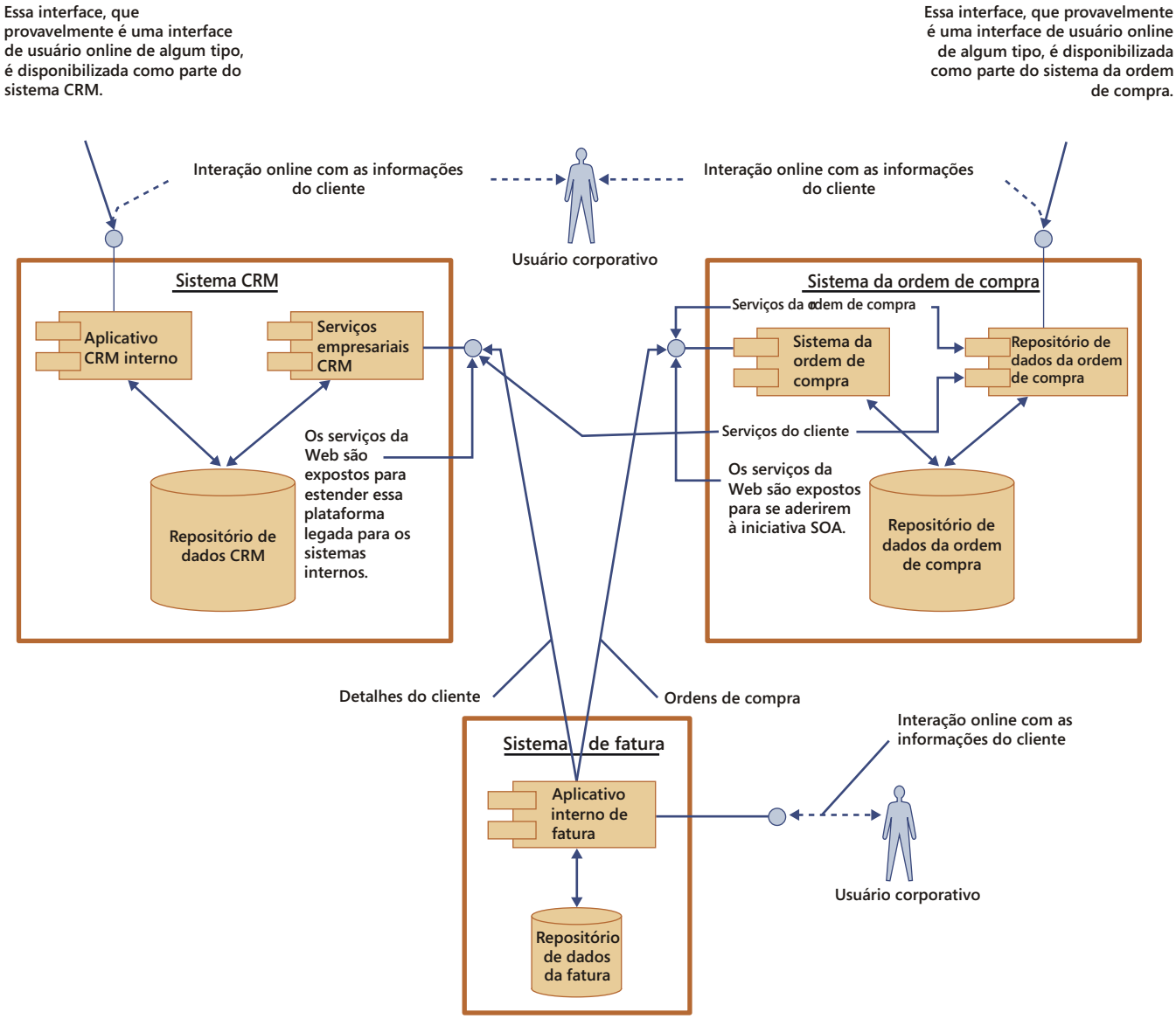
Agora chegamos ao próximo aplicativo, um sistema que gerencia faturamento/contabilidade. Esse sistema também precisa dos dados do cliente, incluindo o número da conta. Agora existem várias escolhas que os projetistas do sistema terão que fazer em relação aos dados do cliente. Eles vão atrás das principais informações de dados do cliente a partir do sistema CRM e do número de conta do cliente a partir do sistema da OC, ou eles apenas acessam o sistema da OC para obter todas as informações do cliente, uma vez que todos os dados existem no sistema da OC? Na realidade, como a durabilidade é uma grande preocupação (uma força nesse antipadrão), é tomada uma decisão para replicar os dados do cliente a partir dos sistemas da OC para o sistema de faturamento. Agora, três sistemas possuem uma visão de dados do cliente em determinados graus de integridade. Três sistemas são forçados a criar alguma lógica comercial em torno do acesso a esses dados. Cada um deles expõe os dados do cliente por meio dos serviços (porque estamos criando uma "arquitetura orientada a serviços") e a capacidade para a empresa gerenciar os dados do cliente de uma maneira ágil continua entrando em colapso.

Conhecedor de Mudanças

Solução refatorada. "Os benefícios de uma SOA só serão totalmente percebidos por uma organização que seja conhecedora da mudança de mentalidade necessária para fornecer soluções que usem novos padrões de arquitetura" A SOA não oferecerá os tipos de benefícios que forem possíveis até que os arquitetos percebam que apenas fornecendo um único ponto de controle sobre os serviços de negócios e os dados é que a organização poderá se tornar realmente a empresa ágil, flexível e escalável prometida pela SOA. Para refatorar a solução com êxito, você deve endereçar as forças que criam o antipadrão:

- **O arquiteto está familiarizado com a estratégia de replicação de dados**. Use KPI (Indicadores Principais de Desempenho) que conduzam uma implementação SOA para criar um caso que não replique os dados desnecessariamente. O treinamento de implementações SOA que reduz a replicação de dados também precisa atenuar essa força.
- **O arquiteto está preocupado com o desempenho de um serviço exposto pelo sistema de dados principais**. Teste para validar o desempenho de um serviço exposto. Por meio do teste e de um entendimento do SLA (Contrato de Nível de Serviço) para o novo aplicativo, muitas pessoas descobrirão que o uso de um serviço novo ou existente que exponha os dados principais será executado dentro das demandas do SLA.
- **Cada serviço separado pode ter visões ligeiramente diferentes da entidade**. As habilidades e as tecnologias para combinar as diferentes visões em uma entidade agregada e manter a autonomia de serviço já estão bem estabelecidas e geralmente podem ser feitas no tempo de execução.
- **O arquiteto está preocupado que a durabilidade do novo sistema possa ser comprometida**. Cuide da durabilidade dos serviços de negócios expostos nos mecanismos de infra-estrutura disponíveis como o agrupamento.

Figura 3 O sistema refatorado



- **Os aplicativos são construídos utilizando um modelo de silo**. Vários projetos precisam do envolvimento de um arquiteto com um olho na direção de serviços novos e existentes e na funcionalidade que eles proporcionam. Na verdade, os aplicativos como conhecemos atualmente devem passar por uma mutação, de uma simples caixa em um diagrama a uma montagem de serviços novos e existentes.
- **Os recursos para fornecer uma solução usando a replicação de dados são abundantes**. Os padrões, os modelos, os arquétipos e as ferramentas atualmente disponíveis permitem a verdadeira SOA com projetistas e desenvolvedores de "nível intermediário".

Seguindo uma análise detalhada dessas questões, você pode decidir sua estratégia de arquitetura. A mudança rumo à orientação de serviço requer que pensemos de forma diferente sobre quais são as perguntas essenciais ao tomar esses tipos de decisões. Em vez de concentrar tanto tempo na arquitetura de meu "aplicativo" os arquitetos devem se concentrar na arquitetura dos serviços de negócios. Os serviços devem ser criados no sistema que mantém e gerencia os dados ou os serviços de agregação devem ser criados para fornecer os dados, eliminando a necessidade de replicar os dados para que outro aplicativo forneça um serviço.

A visão empresarial de um aplicativo precisa ser fundamentalmente alterada. Para que a SOA funcione, devemos mudar da criação de aplicativos (um padrão independente que fornece funcionalidade de usuário) para a criação de produtos: uma série de recursos a serem entregues ao cliente. Devemos estar criando serviços que forneçam essa funcionalidade de negócios. O produto não é nada mais do que uma composição ou orquestração de um ou mais serviços e fornece uma ou mais partes da funcionalidade de negócios. É a agregação de casos de uso solucionados.

Eliminar Ambigüidade

Assim que essa tendência ficar clara e essas forças tiverem sido atenuadas, o acesso direto aos serviços de LOB (Linha de Negócios) se tornará uma realidade. As interfaces de usuário dos novos sistemas podem chamar serviços existentes conforme necessário, sem qualquer via indireta de dados.

Seguindo uma "refatoração" da arquitetura para direcionar os serviços, a ambigüidade da propriedade de dados é eliminada. Os novos aplicativos, como o sistema de Fatura (consulte a Figura 3), estão disponíveis para recuperar os dados dos sistemas principais de registro.

Benefícios. Os arquitetos são constantemente chamados para avaliar a relação custo-benefício de um padrão dentro de determinada solução. Quando o ambiente de arquitetura empresarial tenta mudar para uma SOA, por todos os motivos que fazem da SOA uma arquitetura importante, as trocas da replicação de dados para melhoria da velocidade ou da durabilidade do aplicativo devem ser avaliadas em relação aos benefícios de clareza e simplicidade adquiridos de um ponto único de administração e propriedade que uma arquitetura de serviço pode trazer para a sua organização.

Os benefícios adicionais para o uso do padrão de serviços diretos podem ser localizados no KPI que impulsiona para uma iniciativa SOA originalmente. A orientação a serviços é a mais nova tentativa de criar artefatos reutilizáveis para um aplicativo. Uma extrema economia de recursos pode ser obtida pela reutilização de serviços, com a diminuição dos custos de trabalho e espaço de armazenamento. Embora a replicação de dados possa atender aparentemente os requisitos funcionais, os objetivos da visão de arquitetura empresarial não estão sendo alcançados.

Outro benefício pode ser conseguido na eficiência obtida por recuperar apenas os dados de que você precisa. As arquiteturas que aproveitam a recuperação de dados controlada por evento certamente são mais eficientes do que os processos orientados por lote. Replicar todos os dados durante uma janela em lote, restrita pelo tempo, cria uma sobrecarga dispendiosa. Sem um mecanismo para recuperar os dados certos no tempo certo, utilizando os serviços, você utiliza ciclos de processamento adicionais para mover os dados que podem ter pouco ou nenhum uso dependendo das atividades que dependem desses dados. O padrão de serviços diretos assegurará que você não execute nenhum processamento sem necessidade.

"OUTRO BENEFÍCIO PODE SER ENCONTRADO NA EFICIÊNCIA OBTIDA AO RECUPERAR APENAS OS DADOS DE QUE VOCÊ PRECISA. AS ARQUITETURAS QUE APROVEITAM A RECUPERAÇÃO DE DADOS CONTROLADA POR EVENTO CERTAMENTE SÃO MAIS EFICIENTES DO QUE OS PROCESSOS ORIENTADOS POR LOTE"

Por fim, o objetivo de qualquer serviço de dados corporativo é fornecer uma visão mais completa possível de uma entidade sem sacrificar os benefícios das iniciativas da arquitetura corporativa. Nos casos em que as partes principais dessa entidade são fragmentadas em vários sistemas, o design do serviço pode precisar de algum nível de agregação. No entanto, isso não elimina a opção de usar o padrão de serviços diretos para recuperar esses subcomponentes de nossa entidade.

Uma arquitetura que implementa e consome efetivamente os serviços diretos minimizarão a fragilidade do portifólio da solução corporativa eliminando a redundância. Pela centralização e reutilização, os serviços criados utilizando esse padrão fornecerão a melhor abordagem possível para permanecerem ágeis em um clima de mudança.

Replicação de Dados Como um Padrão

Todas as empresas possuem sistemas complicados e requisitos que requerem que os arquitetos permaneçam pragmáticos quanto aos padrões que endossam. Nesses cenários um arquiteto deve controlar a adoção difundida dos padrões de replicação por meio da boa administração corporativa. Se isso não acontecer, a empresa correrá o risco de usar exageradamente esse padrão. Devido a sua baixa restrição de entrada percebida pelos desenvolvedores, projetistas e arquitetos, a estratégia de replicação de dados deve ser considerada um padrão em raras situações.

Figura 4 Bloco de construção de movimento dos dados

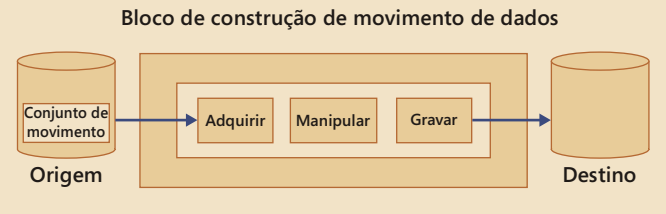
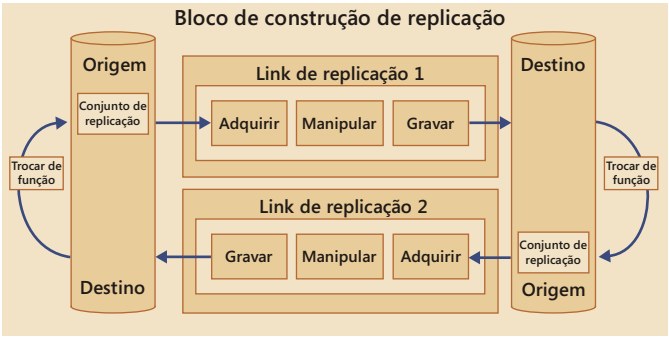


Figura 5 Um cenário de replicação principal-principal



Contexto. O contexto no qual a replicação de dados é um padrão é ligeiramente diferente do contexto no qual ele é um antipadrão. Ele ainda é derivado de um modelo distribuído no qual você deseja ter uma cópia dos dados principais mantidos em um sistema externo do sistema de dados principais. Em algumas situações, você talvez não consiga evitar a replicação. Alguns exemplos incluem: onde a latência da rede é um problema, como em uma WAN; onde o sistema de dados principais é inconstante ou possui janelas de manutenção incompatíveis; quando os recursos offline são um requisito principal do aplicativo; e quando os riscos de não ter os serviços são atenuados pelo uso esperado do aplicativo.

Forças. As forças que influenciam a decisão de fazer uma cópia dos dados principais e que podem justificar a complexidade envolvida na duplicação dos dados são:

- **A disponibilidade de dados sobre o sistema de dados principais não corresponde aos requisitos para o novo sistema**. Por exemplo, o sistema de dados principais deve ser desligado para a manutenção entre certas horas em que o novo sistema deve estar disponível (e os dados são críticos para os casos de uso entregues pelo novo sistema).
- **A rede está inconstante ou muito lenta**. Por exemplo, em uma WAN em que a rede regularmente esteja perdendo conexão ao sistema de dados principais, copiar os dados pode aliviar esse problema. Outro cenário é que a rede é muito lenta para suportar o consumo direto dos dados quando necessário.

- **Espera-se que o sistema tenha recursos offline**. Essa força normalmente é um requisito encontrado nos sistemas que terão pouca ou nenhuma conectividade para os serviços empresariais que estejam fornecendo os dados.
- **Os dados serão copiados sem qualquer lógica comercial reutilizável apenas para fins de análise**. Existem cenários nos quais os dados precisariam ser copiados para um armazém de dados para uso nos relatórios analíticos. Na maioria dos casos, esse tipo de transferência de dados é mais eficaz quando uma ferramenta ETL for utilizada simplesmente para copiar os dados assim que forem considerados prontos para arquivamento.

Solução. Uma solução para essas forças é a replicação de dados. A replicação de dados pode ser executada utilizando um bloco de construção de arquitetura conhecido como *bloco de construção de movimento de dados*. O bloco de construção de movimento de dados consiste em uma origem, um link de movimento e um destino.

O diagrama mostrado na Figura 4 exibe o bloco de construção básico de vários outros padrões de movimento de dados (consulte os Recursos). Dependendo da solução, você pode precisar de múltiplos blocos de construção de movimento de dados para tratar um cenário, por exemplo, quando os dados podem ser atualizados na origem e no destino (consulte a Figura 5).

Benefícios. Grande parte do trabalho descrito nos padrões anteriormente discutidos pode ser realizada com as ferramentas e recursos sem desenvolvimento. Essa capacidade é um dos principais benefícios dessa abordagem porque você geralmente verá um TCO (Custo Total de Propriedade) inicial mais baixo, embora esse benefício não conte a história toda. A manutenção de longo prazo e a versão desses tipos de ferramentas e estratégias são, geralmente, muito dispendiosas. Se o foco principal for o prazo de entrega e o ciclo de vida da solução for relativamente curto, esse pode ser um padrão viável com um alto grau de otimismo. Tenha em mente que você terá fragilidade e possivelmente reduzirá os custos de "refatoração" no trajeto. Às vezes, essa "refatoração" pode ser realizada agrupando-se um sistema legado que não atende os princípios da SOA com uma interface que seja compatível com a SOA. Dessa forma, a arquitetura empresarial pode mudar para a SOA como um todo, sem ter que refatorar cada aplicativo em seu domínio.

Documentar Práticas Recomendadas

Um aspecto inevitável da inovação na tecnologia é a mudança. Os arquitetos de aplicativo devem permanecer concentrados na minimização dos riscos de mudança. Usar os padrões e os antipadrões para documentar as práticas recomendadas tem mostrado êxito desde os padrões OO mais antigos. Quando aplicado de maneira oportuna, os padrões de arquitetura são as armas mais viáveis na batalha contra a mudança.

O padrão e o antipadrão discutidos aqui se concentraram na replicação de dados e sua função em uma empresa com uma iniciativa de SOA. Utilizando um modelo para a descrição do padrão, você pode ver o motivo pelo qual essa técnica pode ter um impacto positivo ou negativo dependendo do contexto no qual está aplicado. Usar os padrões para resolver essa incompatibilidade na arquitetura ilustra exatamente como pode ser a adoção bem-sucedida de uma metodologia orientada à arquitetura.

À medida que a indústria de software continua evoluindo, o papel da arquitetura e dos padrões se torna ainda mais essencial. Padrões, como a replicação de dados que são comuns dentro das atuais arquiteturas, agora podem ser prejudiciais ao progresso de arquitetura da sua empresa. As organizações que podem ver a mudança dos aplicativos para os serviços serão aquelas que farão a mudança de forma muito mais rápida e que verão o potencial completo da SOA.

"A MANUTENÇÃO DE LONGO PRAZO E A VERSÃO DESSES TIPOS DE FERRAMENTAS E ESTRATÉGIAS SÃO, GERALMENTE, MUITO DISPENDIOSAS. SE O FOCO PRINCIPAL FOR O PRAZO DE ENTREGA E O CICLO DE VIDA DA SOLUÇÃO FOR RELATIVAMENTE CURTO, ESSE PODE SER UM PADRÃO VIÁVEL COM UM ALTO GRAU DE MELHORIA."

As metas que pareciam inacessíveis da industrialização de software estão rapidamente se tornando uma realidade. Os padrões são as peças de consistência e agilidade na arquitetura empresarial. Tratá-los como artefatos de arquitetura de primeira classe ajudará a acelerar sua descoberta e adoção. No momento apropriado, uma empresa que cria uma biblioteca reutilizável desses padrões terá aumentado a receptividade dos patrocinadores de seus negócios e terá ganhado uma margem sobre qualquer concorrência.

Recursos

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 e Carsten Zeitz (Microsoft Corporation, 2003)

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

"SOA Antipatterns" (IBM, novembro de 2005)

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

Sobre os Autores

Tom Fuller é CTO e consultor SOA sênior na Blue Arch Solutions Inc. (www.BlueArchSolutions.com), um provedor de consultoria em arquitetura, de treinamento e de solução em Tampa, Flórida. Tom recebeu recentemente o prêmio MVP da Microsoft na categoria Visual Developer ñ Solution Architec. Ele também é o presidente atual da subsidiária Tampa Bay da IASA (International Association of Software Architects), é portador de uma certificação MCSD.NET e gerencia um site de comunidade dedicado à SOA, aos serviços da Web e à Windows Communication Foundation (antiga "Indigo"). Ele possui uma série de artigos publicados recentemente na Active Software Professional Alliance e na revista SQL Server Standard. Tom fala a vários grupos de usuários no sudeste dos EUA. Visite www. SOApitstop.com para obter mais informações e entre em contato com Tom em *tom.fuller@soapitstop.com*.

Shawn Morgan é CEO e arquiteto sênior na Blue Arch Solutions, Inc. (www.BlueArchSolutions.com). Shawn tem arquitetado soluções para várias empresas da Fortune 500 incluindo Federal Express, Beverly Enterprises e Publix Super Markets. Shawn se concentra em permitir que as empresas forneçam soluções de TI por meio da arquitetura. Entre em contato com Shawn em *shawn.morgan@bluearchsolutions.com*.

Padrões para Consumo e Composição de Dados de Alta Integridade

por Dion Hinchcliffe

Resumo

O desafio é consumir e gerenciar dados de várias fontes e em diferentes formatos enquanto participa do ecossistema federado de informações e mantém integridade e baixo acoplamento com bom desempenho. Aqui estão alguns padrões emergentes do mundo crescente de combinações e composição de aplicativos.

O Atualmente o campo de desenvolvimento de software refere-se tanto à montagem e composição de serviços e dados preexistentes quanto à criação de nova funcionalidade. O mundo de combinações na Web e de aplicativos compostos no mundo da SOA (Arquitetura Orientada a Serviços) exige que os dados, em formatos muito diferentes e de praticamente qualquer origem, sejam reunidos, interajam de forma limpa e sigam seus caminhos separados. E essas abordagens exigem que essa interação aconteça de maneira eficiente, sem perder a fidelidade ou a integridade.

A considerável variedade de dados atualmente inclui um conjunto extenso de formatos baseados em XML, além de formatos de dados mais leves e cada vez mais generalizados como JSON (JavaScript Object Notation) e microformatos. Os aplicativos também precisam trabalhar cada vez mais com formatos de dados densos, como imagens, áudio e vídeo, além dos formatos de trabalho cotidiano mais antigos, como texto, EDI e mesmo objetos nativos. Todos esses formatos estão sendo cada vez mais entrelaçados e mesclados, ao mesmo tempo em que os sistemas se tornam mais interconectados e integrados em vastas cadeias de suprimentos, ESBs (Enterprise Service Buses), SOAs e combinações da Web. Manter a ordem nesse caos de dados é mais importante que nunca. A boa notícia é que estão começando a surgir regras para tratar toda essa heterogeneidade de dados.

Os serviços da Web, a SOA e principalmente a própria Web são compostos de padrões abertos que possibilita a comunicação dos sistemas — sendo o onipresente e essencial HTTP o exemplo principal. No entanto, essa comunicação não permite pressupor que tipo de dados o seu software terá de consumir no futuro. Embora você possa apostar que provavelmente irá encontrar alguma forma de XML, é igualmente provável que irá encontrar formatos de dados leves que estão crescendo em popularidade, como JSON. No entanto, cada vez mais, poderíamos ter facilmente apenas texto simples; uma árvore de objetos nativos; ou mesmo uma série de serviços WS-* que será fornecida pelo futuro WCF (Windows Communication Foundation).

Os desenvolvedores, portanto, enfrentam sérios desafios em termos de criar boa modelagem de dados, arquitetura e técnicas de integração que possam abranger essa diversidade. A heterogeneidade das formas de representação de dados pode ser bastante desencorajadora em ambientes com altos níveis de integração de sistemas. Não esqueça que, em sistemas bastante flexíveis e altamente federados, como muitos aplicativos estão se tornando, a probabilidade de alterações freqüentes seja alta. Portanto, está nas mãos da comunidade de desenvolvimento de aplicativos criar um corpo de conhecimento de práticas recomendadas para o consumo, com poucas restrições, de dados de muitas origens e formatos diferentes, com técnicas compatíveis para integrar, unir e relacioná-los. A comunidade também precisa garantir que a integridade seja mantida enquanto permanece altamente permeável à mudança e mesmo à inevitável evolução dos formatos de dados utilizados.

Embora os padrões aqui apresentados não sejam impositivos e tenham a intenção principal de orientação, são baseados no conceito bastante aceito dos contratos de interface. Os contratos de interface agora são comuns no mundo dos serviços da Web com WSDL e também com muitas linguagens de programação, mas particularmente em design por contrato. Em um contrato de interface, um provedor e um fornecedor concordam com um conjunto de interações, geralmente descrito em termos de métodos ou serviços. Essas interações farão com que dados de interesse comum sejam transmitidos de um lado para outro dos limites de interface.

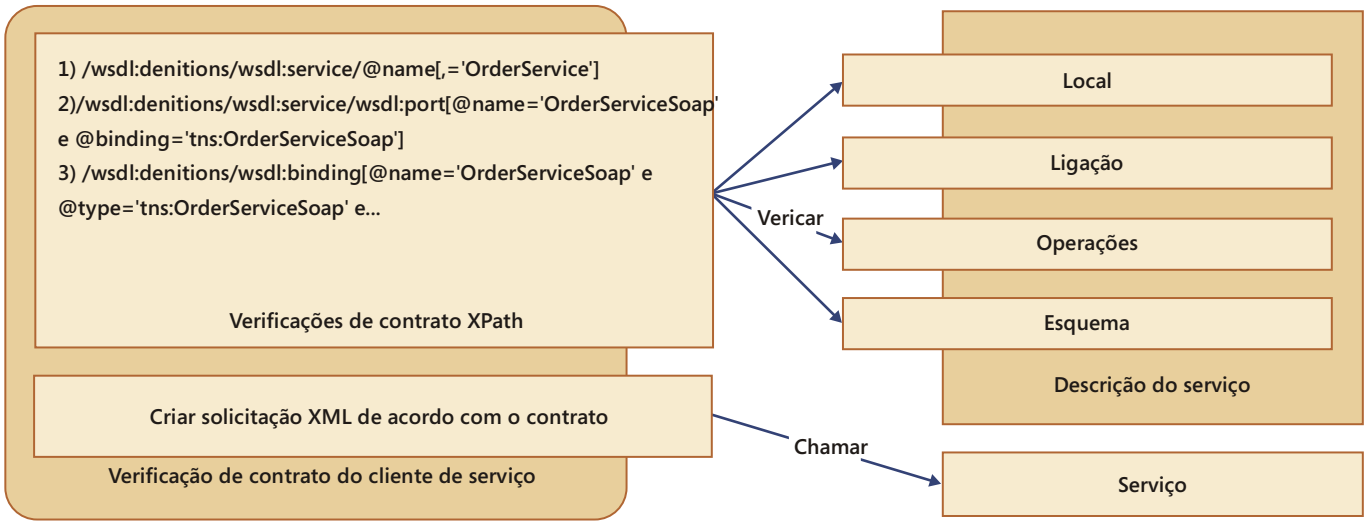
A definição exata das interações, incluindo suas localizações e seus protocolos, é determinada com precisão no contrato, que de preferência será legível por máquina. Principalmente as estruturas de dados transmitidas como parâmetros entre o fornecedor e o cliente durante cada interação. E essas estruturas de dados constituem na realidade o centro do modelo de combinação/composição de aplicativos. Porque é ali, quando essas estruturas de dados altamente diversas se encontram, que precisamos colocar princípios de integração de dados e recombinação no mais alto nível.

Forças e Restrições de Consumo e Composição

Nesse sentido, podemos obter uma noção geral das forças que estão definindo a forma da composição e do consumo de dados nos sistemas modernos de software distribuído. Em uma ordem preliminar, elas são:

O contrato de interface como condutor da composição e do consumo de dados. Ao utilizar estruturas de dados de qualquer origem externa ou serviços, como um serviço da Web, as estruturas de dados devem ser validadas em relação ao contrato de interface fornecido pelo serviço de origem. Essa validação geralmente pode ser feita em tempo de projeto para origens de dados que são conhecidas como altamente estáveis e confiáveis. Mas em sistemas federados, particularmente nos que não estão sob controle direto do consumidor, deve-se considerar cuidadosamente a verificação do contrato em tempo de execução. A verificação de contrato em tempo de execução em sistemas flexíveis é uma técnica emergente e algumas opções interessantes estão à disposição do consumidor de serviços para evitar problemas básicos.

Figura 1 Os esquemas incorporados em um contrato de interface são geralmente a maior dependência do cliente



O resultado é que o contrato de interface é o artefato principal que conduz a composição e o consumo de dados.

O obstáculo da abstração atrapalha a composição e o consumo de dados. O ponto físico em que ocorre a integração de dado está cada vez mais fora do controle clássico dos bancos de dados, e as bibliotecas de acesso a dados convertem tudo em uma abstração de dados unificada.

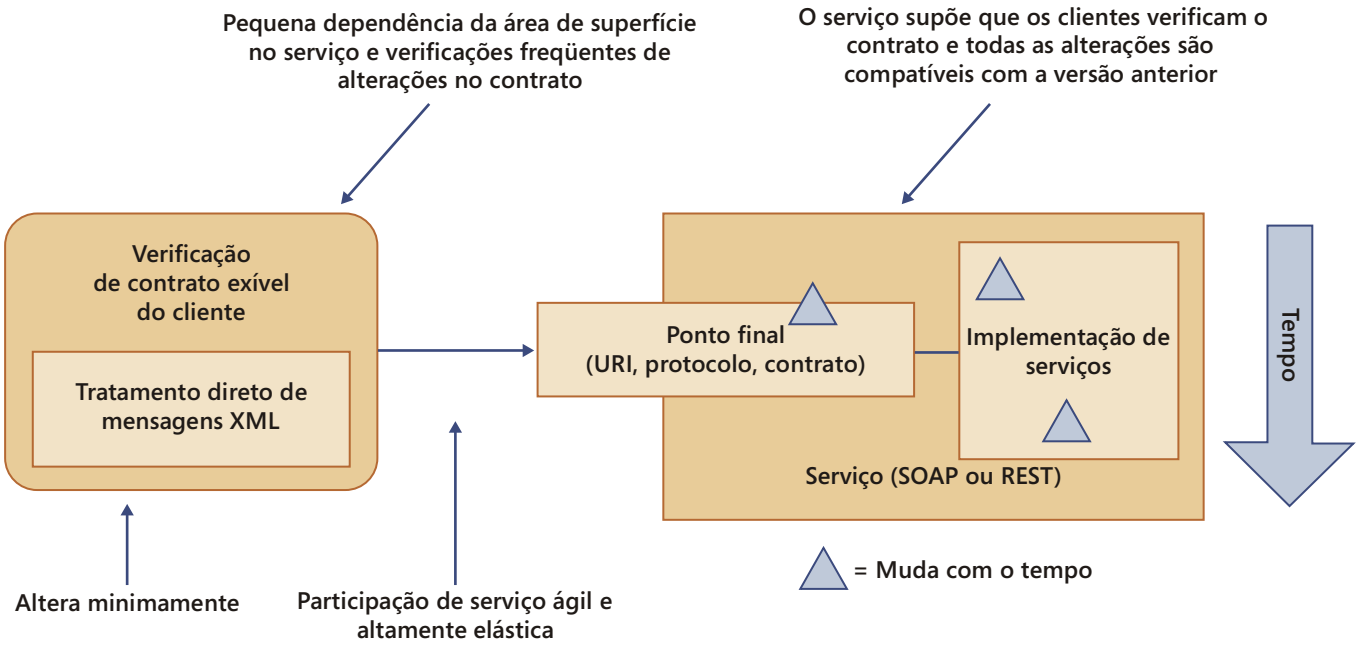
"AO UTILIZAR ESTRUTURAS DE DADOS DE QUALQUER ORIGEM EXTERNA OU SERVIÇOS, COMO UM SERVIÇO DA WEB, AS ESTRUTURAS DE DADOS DEVEM SER VALIDADAS EM RELAÇÃO AO CONTRATO DE INTERFACE FORNECIDO PELO SERVIÇO DE ORIGEM"

Os dados recuperados de vários serviços externos e em vários formatos podem colidir, e realmente colidem dentro do navegador, dentro de front-end no clientes ou no código interno no lado do servidor ou banco de dados. As estruturas de dados desses serviços, dependendo do aplicativo, variam de objetos nativos, XML, JSON e documentos/fragmentos SOAP a dados de texto, binários e de multimídia. Embora o obstáculo da abstração seja um problema bem conhecido em software há muito tempo, ele só foi exacerbado pela proliferação do número de modelos disponíveis para representar informações. Ao consumir e compor essas estruturas de dados em representações agregadas utilizáveis e, em seguida, retorná-las aos seus serviços de origem, surge uma série de grandes problemas.

- **Variabilidade dos formatos de dados**. Os formatos fundamentais das estruturas de dados de base geralmente são altamente incompatíveis, e bons recursos locais para processar todas as formas que podem ser encontradas geralmente não estão disponíveis.Particularmente problemáticos são os formatos extremamente complexos que necessitam de pilhas de protocolos sofisticados para serem manipulados, como serviços da Web com requisitos de WS-Policy.
- **Identidade de dados**. As chaves e os identificadores de objetos/dados exclusivos geralmente não estão em formatos comuns, e não existem os recursos disponíveis para verificá-los e aplicá-los de forma consistente nas abstrações.
- **Relacionamentos de dados.** Estabelecer relacionamentos e associações entre estruturas altamente diferenciadas pode ser problemático para a eficiência e o desempenho por uma série de razões. Converter todos os dados em um formato comum pode ser dispendioso, em termos de custo de manutenção e de tempo de execução. Isso também apresenta o problema de manter a procedência de dados misturados e extraí-los de volta após serem modificados. Manter as estruturas de dados nos formatos nativos presume bons mecanismos para sua composição, seu relacionamento e sua manipulação.
- **Procedência dos dados**. Manter o serviço de origem de um bloco de dados é essencial para manter um contexto válido para conversações contínuas com a origem e para alterações nos dados em particular, como atualizações, acréscimos e exclusões dos dados.
- **Integridade dos dados**. É importante modificar a origem de dados, garantindo que as alterações sejam válidas de acordo com o contrato de interface do serviço, e não violar as definições de esquema. Com documentos XML bem-definidos suportados por esquemas XML cheios de recursos, essas avaliações tornam-se um problema menor. No entanto, modelos de programação mais leves como JSON e outros semelhantes geralmente não têm nenhum esquema legível por máquina. Esses tipos de formatos de dados, embora geralmente disponíveis somente em um serviço federado, apresentam os maiores riscos de problemas de integridade uma vez que as regras para alterá-los ou manipulá-los de algum outro modo geralmente não são tão bem-definidas quanto para os outros formatos.
- **Conversão de dados**. Geralmente não existem conversões bem especificadas entre formatos de dados ou, pior ainda, várias opções se apresentam que alteram os dados de maneiras sutis. As áreas afetadas com mais freqüência são as representações numéricas, mas essas conversões também podem afetar uma origem de dados em vários conjuntos de caracteres, dados de áudio/vídeo e dados GIS quase com a mesma freqüência.

Grandes áreas de dependência dos contratos. Estruturas de interface e esquemas extremamente complexos estão se transformando em um dos principais problemas dos sistemas distribuídos. A experiência tem mostrado constantemente que "mais simples" é mais confiável e de

Figura 2 Verificação em tempo de execução do contrato de interface em serviços da Web



qualidade superior. No entanto, os contratos de interface de serviços distribuídos geralmente possuem definições WSDL e esquemas XML grandes e complexos. A probabilidade de que ocorrerão mudanças, inadvertidamente ou não, aumenta com a complexidade de um contrato de interface e seus esquemas incorporados. Embora as abordagens XML e REST para os serviços tendam a estimular a simplicidade desejada, às vezes determinados domínios de aplicativos não são "simples" e esses modelos de serviços estão sujeitos a problemas adicionais devido à falta de padrões de contratos de interface. O resultado é que a probabilidade de que ocorram alterações no contrato sem que um fornecedor remoto esteja ciente das alterações aumenta com o tamanho do contrato global, porque as alterações em esquemas grandes são mais prováveis de passarem despercebidas do que alterações em um esquema menor.

Alterações de versão implícitas. Mesmo em ambientes bem controlados, podem correr alterações em serviços e em seus contratos de interface sem que todas as partes dependentes sejam notificadas. Exceto se houver alterações na maneira como os serviços de dados funcionam atualmente, esse padrão se tornará cada vez mais comum nos ambientes altamente federados atuais e futuros. Se não houver nenhuma alteração, os serviços dos quais você depende serão modificados sem o seu conhecimento ou aviso prévio e você terá que aceitar esse fato como prático, inevitável e infeliz. Por conseqüência, desenvolver estratégias conscientes para detectar alterações de versão e tratá-las de maneira efetiva é essencial para manter a qualidade dos dados e da funcionalidade de um aplicativo composto.

Padrões para Composição e Consumo de Dados

Os padrões aqui descritos têm a intenção de apresentar uma postura leve em relação à modelagem de dados e à arquitetura de aplicativos. Essas observações são resultado da análise do mundo dinâmico dos aplicativos de composição e portais que proliferam na Web há vários anos. As abordagens ágeis e minimalistas usadas pelos portais em particular são uma inspiração, mas de modo geral não são suficientes para criar software de qualidade. Esses padrões têm a intenção de capturar o espírito dos portais, colocá-los no contexto com boas práticas de engenharia de software e iniciar discussões e debates na comunidade de software sobre esses métodos enxutos para conectar serviços e dados.

Padrão 1 - Mínima área de dependência dos contratos. Esse padrão é o aspecto front-end da conhecida máxima de projeto de software: "Seja liberal naquilo que recebe e conservador naquilo que envia". A maioria dos kits de ferramentas de serviços da Web, das estruturas de bancos de dados relacionais e das bibliotecas de multimídia estimula dependências de partes muito grandes do contrato de interface, muitas vezes do contrato inteiro, sem levar em consideração aquilo de que o software depende. Para muitos dos cenários de composição e consumo, uma área de dependência pequena é realmente tudo que é necessário. Uma dependência completa do contrato gera uma quantidade imensa de fragilidade porque alterações que não possuem relação com os dados na estrutura de dados da qual se depende ainda irão interromper o cliente. Embora alguns kits de ferramentas de consumo de dados já estejam abrindo mão dessa dependência, geralmente existe pouco controle quanto às partes do contrato que importam para as ferramentas.

Para muitos aplicativos, somente dependências diretas dos elementos de dados internos necessários são apropriadas. Todas as outras dependências devem ser omitidas ativamente do relacionamento de dependência dentro dos dados originais. A conclusão aqui é que alterações no contrato de interface que não sejam importantes para o consumidor de dados não devem impedir o uso dos dados. O inverso também deve ser verdadeiro: alterações importantes no contrato devem tornar-se aparentes imediatamente para impedir comportamento incorreto e o uso dos dados. Os exemplos de dependências mínimas incluem: XML, caminhos principais entre esquemas e elementos de dados; dados relacionais, somente tabelas, tipos, colunas e índices usados pelo cliente; e multimídia, somente as partes da estrutura de mídia necessárias, como canais específicos, taxas de bits, parte da imagem, segmento de vídeo, etc.

Padrão 2 - Verificação de alterações no contrato no tempo de execução. Os serviços dos quais os aplicativos compostos dependem, com relação a dados, estão sujeitos a alteração inesperada . Essa alteração pode ser devido à mudança do EndPoint para um novo

local, à interrupção no uso de um protocolo suportado anteriormente, a alterações no esquema ou mesmo a alterações deliberadas na interface ou a falhas de serviço internas. Essas razões levam à necessidade de verificar o contrato para detectar alterações. Também exigem que seja capaz de tratar algumas alterações de contrato de forma transparente, pois poderão muito bem não afetar a parte do contrato que lhe interessa.

Na realidade, existem dois tipos de Verificação de contrato em tempo de execução. Uma é verificar *a própria especificação do contrato, se existir.* Essa especificação geralmente é o WSDL ou *outros metadados que são publicados oficialmente*, juntamente com o serviço. Isso pode ser verificado facilmente e de forma rápida com uma variedade de técnicas programáticas, incluindo consultas XPath para esquemas baseados em XML ou outras técnicas relativamente leves. Verificações de contrato fixas no código não são tão fáceis de implementar ou alterar e devem constituir uma segunda opção.

A segunda verificação de contrato é fazer a validação em relação às instâncias de dados que o serviço fornece. Os contratos de interface geralmente possuem os seus próprios problemas de abstração com seus mecanismos de entrega e pode ser surpreendente a freqüência com que as inconsistências aparecem entre os dados fornecidos e o contrato de interface, mesmo com kits de ferramentas de alta qualidade.

O maior dilema apresentado por esse padrão é com que freqüência executar a rotina de "verificação do contrato". Verificar o contrato e os dados das instâncias com cada requisição pode ser demorado e geralmente é desnecessário. Em última análise, determinar a freqüência da verificação do contrato depende em parte dos requisitos do aplicativo e sua tolerância a imprecisão e comportamento incorreto. Para muitos aplicativos checar de forma síncrona pode nem ser uma opção, e pode fazer sentido configurar um processo de segundo plano que periodicamente verifique mudanças no contrato, o que inevitavelmente acontecerá.

Padrão 3 - Reduzir estruturas para um formato de abstração de dados comum. Na realidade, o festival de formatos e estruturas de dados com que os portais e aplicativos compostos têm de lidar só vai continuar a crescer. O software pode manipulá-los em seus formatos nativos, o que significa perder oportunidades de manter relacionamentos e aplicar regras de negócios ou pode convertê-los todos em uma abstração comum. Essa conversão é a abordagem que bibliotecas de dados como ADO.NET utilizam com seus DataSets e XML com outras origens de dados não XML. Incorporar as diferenças nos dados de origem convertendo-as em uma abstração de dados que fornece um modelo unificado com que trabalhar pode ser atraente por uma série de razões. Primeiro, os relacionamentos entre as diversas origens de dados de base podem ser verificados e aplicados à medida que os dados são manipulados e alterados. Segundo, visualizações nos dados podem aproveitar o mecanismo de abstração tornando menos necessário criar mecanismos Model-View-Controller (MVC) personalizados e utilizar bibliotecas existentes que podem processar a abstração.

Nem todos os conjuntos de estruturas de dados formatados de Padrão 5 Acesso direto a estruturas de dados nativos. Para forma heterogênea são bons candidatos a esse padrão e pode fazer muitos aplicativos, particularmente os mais simples baseados em pouco sentido para alguns formatos de dados que apresentam alta dificuldade de abstração — dados de imagem com dados textuais, por exemplo. Existe também um custo potencialmente não trivial de conversão e restauração para o formato comum. Para muitos aplicativos, porém, esse custo é inteiramente aceitável.

Em determinados ambientes de consumo de dados como web browser, existem opções limitadas para manter um formato de dados comuns, e JSON e XML DOM tendem a ser bastante populares para as soluções baseadas nesse ambiente. No lado do servidor as existem muito mais opções, mas geralmente são dependentes de plataforma. Estruturas XML, bibliotecas O/R como Hibernate, bancos de dados relacionais e até mesmo gráficos de objetos nativos geralmente constituem excelentes modelos de abstração comuns dependendo do aplicativo. Mas a natureza bastante diferente de dados hierárquicos como XML e gráficos de objetos é um dos obstáculos clássicos na ciência da computação e deve-se tomar cuidado ao utilizar esse padrão.

"A PRÓPRIA WEB ESTÁ SE TORNANDO O FORNECEDOR MAIS IMPORTANTE DE DADOS ALTAMENTE FEDERADOS, ALGO QUE SÓ IRÁ CRESCER E TORNAR-SE MAIS IMPORTANTE NOS PRÓXIMOS ANOS"

O resultado final: Se o desempenho não for absolutamente crítico e os esquemas e formatos de dados de base forem receptivos, esse padrão pode ser bastante poderoso para trabalhar com origens de dados federados. A desvantagem é que a abordagem da abstração comum pode certamente envolver mais manutenção e produzir fragilidade porque o mapeamento e os metadados precisam ser mantidos.

Padrão 4 - Estruturas nativas mediadas com o MVC (Model- View-Controller). Converter todos os dados de origem em um formato comum não será uma opção em muitas situações porque 1) a carga adicional de processamento é expressiva uma vez que grande parte dos dados não serão utilizados, ou 2) duplicá-lo no ambiente local pode ser proibitivo em termos de recursos. Ou a razão é a não existência de um formato comum que faça sentido para todos os tipos de dados. Nesse caso, criar um MVC que realize a mediação de acesso, tradução e integridade das estruturas de dados nativos utilizados pode ser a melhor opção tanto para desempenho como para espaço de armazenamento.

O MVC é um poderoso padrão de projeto que tem sido comprovado como uma excelente estratégia para a separação entre os interesses de um aplicativo. O seu uso é particularmente apropriado quando existem diversos modelos de dados utilizados por uma aplicação. O bom projeto de software reza que uma visão unificada da fonte de dados provê uma interface única, limpa e consistente que torna fácil consultar, interagir e modificar as informações. Acesso aos dados com uma abordagem MVC também é relativamente eficiente uma vez que somente os dados necessários precisam ser processados para satisfazer a maior parte das requisições.

Enquanto existem muitas vantagens com o MVC, no entanto, as desvantagens são parecidas com as do padrão 3 de modo que a manutenção de código MVC pode ser enorme. Certamente, existem diversas bibliotecas prontas que podem auxilar os projetistas de software a construir MVC tanto no cliente como no servidor. Esteja avisado, no entanto: mapear código é frágil e tedioso.

Padrão 5 - Acesso direto a estrutura de dados nativo. Para muitas aplicações, especialmente as simples, baseadas em browser, converter dados em formatos comuns ou criar arquiteturas MVC sofisticadas simplesmente não é uma boa opção. Acesso direto aos dados faz mais sentido, e a decisão geralmente é de senso comum, realizado com as bibliotecas de sempre. Além disso, como mencionado antes, "simples" é muitas vezes melhor e de maior qualidade porque existe menor chance de erro ou pontos de manutenção.

Tabela 1 **Abstrações de dados comuns**

Abstração	Padrão do contrato	Vantagens	Desvantagens
Texto, JSON, binário	Informal, textual	Relativamente eficiente	Não autodescritivo, não idealmente eciente
Objetos nativos	Denição de classe	Comportamento e dados unificados, encapsulamento, abstração de alto nível, composição anLigarad	Exige a conversação da maioria dos dados em objetos, necessitando de uma técnica de mapeamento
XML	Esquemas XML (XSD), Relax NG, WSDL e muitos outros	Autodescritivo, descrição de esquema rica em recursos e extensível sem romper a compatibilidade com a versão anterior	Muito ineciente em tamanho, não há como distribuir comportamento e as descrições de esquema são limitadas mesmo com XSD
Imagens	Especificações para JPEG, TIFF, GIF, BMP, PNG e muitos outros	N/A	N/A
Áudio	Especificações para WAV, WMA, MP3 e AAC	N/A	N/A
Vídeo	Especificações para AVI, QuickTime, MPEG e WMF	N/A	N/A

Nesse padrão, que funciona melhor com dados menos estruturados, as estruturas de dados nativos podem ser usadas diretamente sem um intermediário ou qualquer conversão de dados, o que significa que dados armazenados em texto, XML, JSON etc. são manipulados de forma nativa. A desvantagem, naturalmente, é que não é possível depender dos recursos que as bibliotecas de abstração de dados podem oferecer para aplicar integridade ou controle de alterações. No entanto, esse padrão geralmente exige a menor quantidade de processamento ou de conhecimento de bibliotecas de terceiros. Pode ser fácil de desenvolver e, como não existe conversão ou camadas de acesso a dados a percorrer, também é bastante rápido.

Padrão 6 - Toda modificação de dados é atômica. As visualizações de dados e os serviços mais sofisticados prescrevem propriedades conhecidas como ACID (Atomicity, Concurrency, Isolation, and Durability) para atomicidade, concorrência, isolamento e durabilidade. Geralmente atribuído a sistemas de bancos de dados, ACID é um princípio básico prático e excelente para praticamente qualquer sistema de acesso a dados simultâneos. Infelizmente, quase ninguém ainda no mundo dos portais e serviços da Web tem a noção de transações em seus protocolos, o que iria conferir muitas das propriedades de ACID à modificação de dados. Longe de ignorar o problema, desenvolvedores de portais e aplicativos compostos devem estar muito cientes de que trabalhar com seus dados é semelhante a equilibrar-se em uma corda banda.

Essa percepção apresenta problemas significativos e com cenários complexos de modificação de dados, um deles sendo que qualquer recuperação e armazenamento de dados que seja dependente de uma conversação estendida com algum serviços quase certamente não será protegida pelo mesmo limite de transação e pelas propriedades ACID relacionadas. O código do software deve esperar a ocorrência de problemas de integridade de dados, particularmente em uma conversação prolongada que poderá nunca chegar até a conclusão ou falhar no meio. Evitar de uma vez conversações de longa execução é uma opção. Tornar a operação do software dependente, tanto quanto possível, em interação atômica individual com serviços é uma outra boa maneira. Cada etapa da interação é um sucesso visível e discreto que permite ao software oferecer aos seus usuários opções claras quando uma conversação com o serviço fornecedor de dados falha. Essas duas opções permitem aos desenvolvedores de software respeitar o princípio básico de ACID.

Um Retorno à Simplicidade

A própria Web está se tornando o fornecedor mais importante de dados altamente federados, algo que só irá crescer e tornar-se mais importante nos próximos anos. Embora XML e formatos de dados leves provavelmente constituirão a estrutura de dados dominante com a qual a maioria dos softwares terá de trabalhar no futuro previsível, haverá situações imprevistas no caminho. Essas situações imprevistas incluirão muitas otimizações necessárias no XML, como XML binário, progressos em microformatos, novos codecs de multimídia que irão mudar de repente todo um cenário de áudio/visual e protocolos de transporte completamente novos como Bittorrent, que irão tornar muitos dos padrões aqui problemáticos, para dizer o mínimo.

Um retorno à simplicidade no projeto de dados está de novo em voga, exemplificado pelo aumento de interesse em formatos como JSON, microformatos e linguagens dinâmicas como PHP e Ruby. Essa simplicidade de projeto pode tornar os dados mais maleáveis e mais fáceis de interconectar. Também oferece menos dificuldade ao escrever software, para cuidar e gerenciar. Embora os padrões aqui representados sejam abrangentes, podendo resultar em menor fragilidade, mais flexibilidade, e composição e consumo de dados de alta integridade, a história está realmente apenas começando. À medida que a Web tornase menos voltada para páginas da Web visuais e mais para serviços, conteúdo e dados puros, tornar-se um consumidor e fornecedor ágil do ecossistema de informações será cada vez mais um fator crítico para o sucesso.

Recursos

microformatos
http://m icrofo rmats.org

Wikipedia
http://en.wikipedia.org/wi ki/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/H PL-2005-83.html

Sobre o Autor

Dion Hinchcliffe é o diretor de tecnologia da Sphere of Influence Inc.

Nordic - Um Design Objeto/Relacional De Banco de Dados

por Paul Nielsen

Resumo

O Nordic (New Object/Relational Database Design), assim como várias ferramentas, não é a solução mais adequada ou conveniente para todos os tipos de problemas de banco de dados. No entanto, o modelo híbrido de objeto/relacional pode fornecer mais poder, maior flexibilidade, melhor desempenho e integridade de dados ainda mais alta do que os modelos relacionais tradicionais, especialmente para os bancos de dados que se beneficiam da herança, extração de dados criativa, flexíveis interações de classe ou restrições de workflow. Descubra algumas das inovações que são possíveis quando a tecnologia orientada a objetos for modelada usando os bancos de dados relacionais maduros de hoje.

As diferenças entre o desenvolvimento orientado a objetos e o modelo de banco de dados relacional criam uma tensão geralmente chamada de *obstáculo da incompatibilidade objeto-relacional*. A hereditariedade simplesmente não é bem traduzida em um esquema relacional. O obstáculo técnico da incompatibilidade é agravada pela incoerência cultural entre os codificadores de aplicativos e os DBAs (Administradores do Banco de Dados). Geralmente nenhum dos lados entende totalmente nem respeita o léxico do outro. Uma maneira certeira de ficar sob a pele de um DBA é referir-se ao banco de dados como o "utilitário de persistência de objetos". Essa relação é desagradável porque cada lado traz um conjunto exclusivo de vantagens para o problema de arquitetura dos dados.

Em muitos aspectos o design orientado a objetos é superior ao modelo relacional. Por exemplo, a projetar hereditariedade entre as classes pode ser realizada usando um padrão relacional de supertipo/subtipo, mas o design orientado a objetos é uma solução mais elegante. Além disso, quase todo código de aplicativo é orientado a objetos, e um banco de dados orientado a objetos faria interface com o aplicativo de um modo muito mais fácil do que com um banco de dados relacional.

Tão sofisticada quanto a tecnologia orientada a objetos na modelagem de realidade, o lado relacional não fica sem vantagens significativas. Os mecanismos de banco de dados relacionais oferecem opções de desempenho, de escalabilidade e de alta disponibilidade e o impulso financeiro para assegurar que a plataforma de banco de dados ainda estará aqui em algumas décadas. A tecnologia relacional é bem entendida e os bancos de dados relacionais oferecem ferramentas de relatório e consulta mais poderosas do que os bancos de dados de objetos. Algumas empresas de bancos de dados de objeto puros e disponíveis simplesmente não têm os recursos para competir com a Microsoft, a Oracle ou a IBM.

O enigma da diferença de impedâncias entre objeto e relacional então está em como englobar melhor a elegância das tecnologias orientadas a objetos enquanto retém o poder, a flexibilidade e a estabilidade de longo prazo de um mecanismo de banco de dados relacional maduro. Como os programadores de aplicativos geralmente são os mais interessados em resolver esse problema e a natureza humana tende a resolver os problemas utilizando o conjunto de habilidades mais confortável, não é surpreendente que a maioria das soluções seja implementada em uma camada de mapeamento entre o banco de dados e o código de aplicativo que converte os objetos em tabelas relacionais para os objetos persistentes.

A Proposta Nordic

Proponho que um modelo relacional, projetado para emular os recursos orientados a objetos, possa ser extremamente bem executado dentro dos gerenciadores de banco de dados relacionais atuais e que a manipulação da herança de classes e das associações complexas feitas diretamente no banco de dados, perto dos dados, é realmente muito eficaz. Essa eficiência nem sempre era o caso. Eu fui contratado para otimizar um design de banco de dados O/R (objeto/relacional) com volume intenso de T-SQL (Transact-SQL) implementado com o SQL Server 6.5 e falhei. O desenvolvimento do Nordic Object/Relational Database Design envolveu o um ano de iterações, um esquema de metadados simplificado e a maturidade do T-SQL do SQL Server.

"TÃO SOFISTICADA QUANTO A TECNOLOGIA ORIENTADA A OBJETOS NA MODELAGEM DA REALIDADE, O LADO RELACIONAL NÃO FICA SEM VANTAGENS SIGNIFICATIVAS"

Como com qualquer projeto de banco de dados, uma camada de abstração de dados estritamente forçada, que encapsula o banco de dados, é necessária para a capacidade de extensão em longo prazo. Para um banco de dados híbrido de O/R, a camada de abstração de dados também fornece a fachada para os recursos orientados a objetos. Por trás do código da fachada estão o esquema de metadados e a geração de código para as classes, os objetos e as associações (consulte a Figura 1). Existem algumas decisões principais de design ao implementar essa solução.

Gerenciamento de classe. Dentro do esquema relacional, a classe e os metadados de atributo são facilmente modelados usando uma relação comum de um para vários. A relação de superclasse/subclasse é modelada como uma árvore hierárquica, utilizando o padrão de lista de proximidade mais comum ou o padrão de caminho materializado mais eficiente.

Navegar para cima e para baixo da hierarquia de classe com as funções definidas pelo usuário permite que as consultas SQL sejam facilmente

associadas a classes ascendentes ou descendentes de qualquer classe. Essas funções definidas pelo usuário são aproveitadas em toda a camada da fachada. Por exemplo, ao selecionar todas as propriedades para uma classe, a associação à função definida pelo usuário superclasses() retornará todas as superclasses, e a consulta pode então selecionar todas as propriedades de uma determinada classe incluindo as propriedades herdadas das superclasses.

No contexto de trabalho com os objetos persistidos, o *polimorfismo* se refere à capacidade do método de seleção para recuperar não apenas os objetos da classe atual, mas também todos os objetos de subclasse. Por exemplo, a seleção de todos os contatos deve selecionar não apenas os objetos da classe de contato, mas também os objetos do cliente e as principais subclasses do cliente. Uma função definida pelo usuário que retorna uma variável de tabela de todas as subclasses de uma determinada classe faz com que escrever essa consulta seja eficiente e reutilizável.

Gerenciamento de objetos. Os objetos são melhor modelados usando uma única tabela de objetos que armazena os dados comuns do objeto, como o ID exclusivo do objeto, a classe do objeto, os dados de auditoria e alguns atributos de pesquisa comuns a quase toda classe, como o nome, um atributo de data e assim por diante. Os atributos adicionais são armazenados em tabelas de classes personalizadas que usam uma chave externa do ID do objeto para relacionar os atributos comuns à tabela de objetos. O construtor e outros procedimentos armazenados na fachada e de gerenciamento de classe executam o código DDL (Data Definition Language) para criar ou modificar as tabelas de classes personalizadas e geram o código de fachada personalizada para selecionar, inserir e atualizar os objetos.

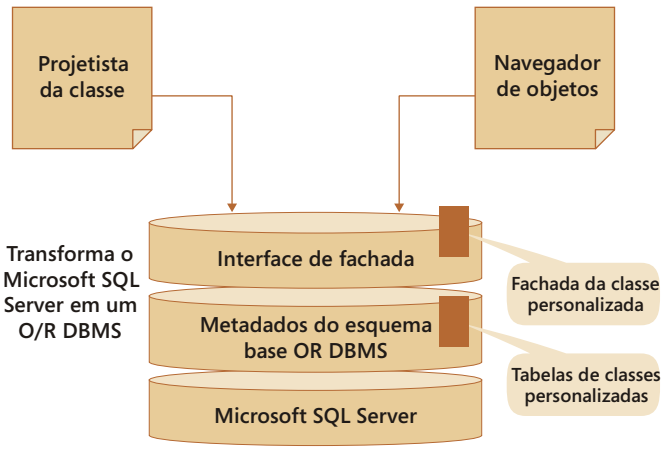
A principal decisão de design para definir como os objetos são armazenados está em como representar os dados de atributos personalizados. Existem três métodos possíveis: o padrão valor-par, as tabelas de classes personalizadas e concretas e as tabelas de classes personalizadas em cascata. O *padrão valor-par*, também chamado de *padrão genérico*, utiliza um padrão em forma de diamante que consiste na classe, na propriedade, no objeto e no valor. A tabela de valores usa uma única coluna para armazenar todos os valores. Essa tabela longa e estreita usa uma linha para cada atributo. Dez milhões de objetos com quinze atributos usariam 150 milhões de linhas na tabela de valor. O SQL Server é mais do que capaz de trabalhar com grandes tabelas; isso não é um problema. Esse modelo parece oferecer maior flexibilidade porque os atributos podem ser incluídos sem modificar o esquema relacional; no entanto, esse modelo sofre com a falta de tipagem de dados (ou, no melhor dos casos, fraca) e é difícil de ser consultado usando SQL.

Uma Tabela para Cada Classe

O modelo de classe personalizada e concreta usa uma tabela para cada classe com colunas para cada atributo personalizado, incluindo atributos herdados e não abstratos. Um objeto existe em duas tabelas apenas (a tabela de objetos e a tabela de classe personalizada e concreta) enquanto os atributos são replicados nas tabelas de classes personalizadas e concretas de cada subclasse. Portanto, se a classe animal tiver um atributo de data de nascimento e a subclasse de mamíferos tiver um atributo de sexo (uma vez que alguns animais não possuem um sexo), a tabela de classe personalizada de mamíferos incluirá as colunas de ID do objeto, data de nascimento e sexo.

Esse padrão tem a vantagem de que a seleção de todos os atributos para uma determinada classe requer, apenas unindo a tabela de metadados do objeto com uma única tabela de classe personalizada. A desvantagem é ter que implementar o polimorfismo; selecionar todos os animais que precisam executar uma união de cada subclasse e eliminar os atributos de subclasse ou adicionar os atributos de superclasse hospedeiros de forma que todas as seleções na união tenham colunas compatíveis.

Figura 1 O design híbrido de O/R usa uma fachada para encapsular a funcionalidade orientada a objetos executada dentro de um esquema de banco de dados relacional.



Em um aprimoramento no padrão de valor-par, as tabelas de classes personalizadas e concretas utilizam uma coluna relacional para cada atributo de modo que a tipagem de atributo possa implementar facilmente os tipos de dados nativos do banco de dados relacional.

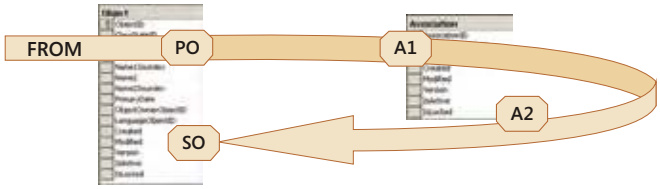
A terceira opção implementada no Nordic Object/Relational Database Design, *tabelas de classes personalizadas em cascata*, utiliza uma tabela para cada classe como a solução de classe concreta. No entanto, em vez de replicar os atributos, cada atributo é representado apenas uma vez em sua própria classe, e cada objeto é representado apenas uma vez em cada classe em cascata. Usando o exemplo de mamíferos e de animais, a tabela de animais contém o ID do objeto e a data de nascimento, e a tabela de mamíferos consiste no ID do objeto e no sexo. Uma instância do objeto mamífero é armazenada nos metadados do objeto, tabela de animais e tabela de mamíferos. O polimorfismo é muito fácil nessa opção; no entanto, são necessárias mais junções para selecionar todos os atributos de subclasses. Como o SQL Server é otimizado para junções, as tabelas de classes personalizadas em cascata apresentam um desempenho muito bom. Assim como nas tabelas de classes personalizadas e concretas, a tipagen forte de dados é suportada pelo banco de dados.

Gerenciamento de associação. As associações da tecnologia orientada a objetos são muito semelhantes às restrições de chave estrangeira da tecnologia do banco de dados relacional e certamente é possível definir as associações dentro de um modelo híbrido de O/R, incluindo os atributos de chave estrangeira, e atribuindo as restrições de integridade referenciais declarativas às tabelas de classes personalizadas.

No entanto, armazenar cada objeto em uma única tabela oferece algumas alternativas interessantes para modelar as associações. Sendo que um banco de dados relacional normalizado pode conter dúzias de chaves estrangeiras, cada uma com uma diferente tabela de chave externa e coluna, e cada uma fazendo referência a uma chave primária diferente, uma tabela de associação híbrida de O/R precisa fazer referência a apenas uma única tabela. Cada relação de chave estrangeira no banco de dados pode ser generalizada em uma única chave externa association.objectid e object.objectid.

A tabela de associação pode ser projetada como uma lista emparelhada (ObjectA_id, ObjectB_id), mas esse design é muito limitado para coleções complexas, e as consultas devem identificar objectA e objectB. A alternativa mais flexível utiliza uma lista de associação de objetos que consiste em um único ID de objeto para fazer referência ao objeto e a um ID de associação para agrupar os objetos relacionados. Uma coluna associationtypeid pode fazer referência aos metadados de associação que descrevem as suas restrições.

Figura 2 Uma consulta genérica de três junções localiza todos os objetos associados, independentemente da classe.




```
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 PO.ObjectID = A1.ObjectID
```

Uma única tabela para cada objeto e outra tabela para cada associação soa radicalmente diferente de um esquema normalizado, e é. De qualquer forma, a estrutura física não é um problema; o SQL Server excede sobre as longas tabelas estreitas e esse design permite ajustar o índice de cobertura com cluster e sem cluster, o que produz o alto desempenho.

Encontre Todas as Associações

Para o arquiteto do banco de dados, o padrão da lista de associação de objetos fornece possibilidades impressionantes. Primeiramente, juntar um objeto com um número n de outros objetos de qualquer classe sempre utiliza a mesma consulta (consulte a Figura 2). Adicionar outras tabelas a uma consulta relacional adiciona n-1 junções, mas a lista de associação de objetos usa consistentemente as mesmas três junções, independentemente do número de classes envolvidas. Dependendo do aplicativo, esse estilo de relacionar os objetos pode ser escalado de uma forma consideravelmente melhor do que um modelo relacional normalizado. Conforme as novas classes são adicionadas ao modelo de dados ou à associação, elas são automaticamente incluídas nas consultas para "encontrar todas as associações" sem modificar qualquer código existente.

Encontrar todas as associações entre as classes ou todos os objetos que não estejam participando de qualquer associação à outra classe ou outras consultas de extração de dados criativas, porém poderosas, são todas as consultas baseadas em conjunto trivial e reutilizável. As funções definidas pelo usuário do SQL Server podem encapsular o trabalho com as associações e as várias combinações lógicas de objetos que participem, ou não, das associações.

As coleções complexas também são possíveis com uma lista de associação de objetos. Por exemplo, uma coleção de sala de aula pode incluir uma sala de aula, um ou mais instrutores, uma ou mais mesas, o currículo e um ou mais alunos, conforme definido nos metadados de associação.

As associações generalizadas abrem mais possibilidades. As páginas da Web também são vinculadas usando um método generalizado; cada hiperlink usa uma marca de âncora e uma URL. Isso é essencialmente uma lista de associação de objetos incorporada dentro do código HTML. É trivial mapear graficamente uma página da Web e exibir a navegação entre as páginas, independentemente do seu conteúdo. Da mesma forma, é trivial saltar entre as tabelas de objeto e de associação e localizar instantaneamente os objetos associados pelos diversos graus de separação, independentemente da classe.

Estou atualmente desenvolvendo um banco de dados de gerenciamento de patrocínio para crianças para ajudar as organizações a lutarem contra a pobreza. Usando a lista de associação de objetos, uma única função definida pelo usuário que localiza associações para qualquer objeto pode descobrir que Joe ampara uma criança no Peru, que está escalado para participar de uma reunião sobre a pobreza, escreveu três cartas para a criança, enviou um presente no ano passado e indagou sobre uma criança na Rússia.

Tecer a lista de objetos de associação para vários graus de separação também revela que Greg está escalado para visitar a cidade no Peru onde vive a criança de Joe, outras 14 pessoas participaram da mesma reunião sobre pobreza que Joe e três delas amparam crianças no Peru. Essa flexibilidade para qualquer objeto com uma consulta é impossível em um design relacional.

Estado do Workflow do Objeto

A generalidade da lista de associação de objetos presta-se a outra inovação de banco de dados — integrando o estado do workflow do objeto no banco de dados. Embora o estado do workflow não se aplique a todas as classes, para algumas delas o workflow é uma dimensão da integridade de dados ausente no modelo de banco de dados relacional.

Um workflow típico para um pedido pode ser um carrinho de compras, pedido confirmado, pagamento confirmado, inventário alocado, em andamento, pronto para envio e enviado. Uma chave estrangeira relacional apenas restringe a tabela secundária fazendo referência a um valor válido de chave primária. Usando um banco de dados relacional, pode ser criada uma linha de detalhes de envio que faça referência ao pedido, independentemente do estado do workflow do pedido. Código personalizado deve validar se o pedido concluiu certas etapas antes do envio.

"COM OS ESTADOS DO WORKFLOW CAPAZES DE SEREM HERDADOS DEFINIDOS COMO PARTE DOS METADADOS DE CLASSE, OS METADADOS DE ASSOCIAÇÃO PODEM RESTRINGIR OS OBJETOS AO ESTADO DE CLASSE E DE WORKFLOW QUE INTEGRAM O WORKFLOW NOS DADOS DO OBJETO"

Com os estados do workflow capazes de serem herdados definidos como parte dos metadados de classe, os metadados de associação podem restringir os objetos ao estado de classe e de workflow que integram o workflow nos dados do objeto.

Destaquei algumas das inovações possíveis quando a tecnologia orientada a objetos for modelada usando os bancos de dados relacionais maduros de hoje. Assim como qualquer ferramenta, o Nordic Object/Relational Database Design não é a melhor solução para todo problema de banco de dados; no entanto, para os bancos de dados que se beneficiam da hereditariedade, extração criativa de dados, interações de classe flexível ou restrições de workflow, o modelo híbrido de O/R pode fornecer mais poder, flexibilidade, desempenho e até integridade de dados do que os modelos relacionais tradicionais.

Sobre o Autor

Paul Nielsen é um SQL Server MVP, autor da série *SQL Server Bible (Wiley, 2002)*, e está escrevendo o *Nordic Object/Relational Design In Action (Manning)*, que deve ser publicado ao final deste ano. Ele ministra workshops sobre design do banco de dados e otimização e pode ser contatado neste site, *www.SQLServerBible.com*.

Adote e Aproveite os Processos Ágeis no Desenvolvimento de Software Internacional

por Andrew Filev

Resumo

No desenvolvimento de software moderno, há duas tendências que permitem obter mais por menos: desenvolvimento ágil e terceirização internacional. Vamos ver como e quando combinar com êxito esses dois fatores para aumentar a competitividade de seus negócios.

Durante a era "pós-bolha", os orçamentos de TI caíram mais do que a demanda por seus serviços, e isso levou os gerentes a buscar soluções com melhor custo/benefício e deu força à tendência de terceirizar o desenvolvimento de software para países emergentes (desenvolvimento internacional). A motivação econômica não é a única força desta tendência. O crescimento rápido ocorrido recentemente, desencadeado por uma infra-estrutura de comunicações aprimorada, também desempenha um papel importante.

Em relação ao trabalho em equipes distribuídas em geral e à terceirização internacional em particular, os softwares de VoIP (Voz em IP), as mensagens instantâneas, os clientes de email e wikis facilitaram a comunicação online. Além disso, atualmente, muitas vezes é melhor usar ferramentas online, como wikis, do que as comunicações pessoais, já que essas ferramentas não só ajudam a comunicar informações, mas também ajudam a estruturá-las e armazená-las. Essas ferramentas também são eficientes para distribuir informações a vários destinatários.

Essas conexões com a Internet, rápidas e disponíveis globalmente, permitem o uso de outras ferramentas, o que contribui para essa tendência. As ferramentas de modelagem ajudam a tornar a documentação mais auto-explicativa em equipes distribuídas. Rastreadores de bugs, servidores de controle de fonte, portais da Web e ferramentas de colaboração online ajudam a coordenar projetos distribuídos. Serviços de terminal e máquinas virtuais facilitam a administração e o teste remoto.

A Internet também levou os países emergentes ao caminho da alta tecnologia. Como a Internet transcende as fronteiras políticas, milhares de jovens de países em desenvolvimento, como a Rússia e a China, a usam para aprender tecnologias de ponta e melhorar o conhecimento de inglês. Essa nova era de engenheiros de software conhecedores da Internet veio na hora certa para reforçar a tendência do desenvolvimento internacional.

O aumento recente da terceirização internacional a tornou forte o suficiente para se tornar objeto de debates políticos. Para os fins dessa discussão, suponha que o desenvolvimento internacional de software é uma realidade, e enfocaremos a maximização do retorno desses trabalhos de terceirização. Deixaremos a política de lado, mas iremos consultar a lista de recursos da pesquisa do McKinsey Global Institute que quantifica os benefícios da terceirização internacional para a economia dos EUA e acaba com vários mitos a esse respeito.

Tendências do Desenvolvimento Ágil de Software

Vamos voltar à nossa terra. Aqui, corações e mentes de vários engenheiros são conquistados por outra tendência moderna: o desenvolvimento ágil de software. Os métodos pesados e lentos não servem no dinâmico ambiente de negócio atual. Orçamentos menores exigem mais resultados, e a burocracia nunca foi a melhor opção em termos de ROI (Retorno de Investimento). O poder dos métodos ágeis está na colaboração, flexibilidade e dedicação ao valor de negócio do software, refletido nos princípios centrais do Manifesto Agil: indivíduos e interações de processos e ferramentas, que trabalham com software usando uma documentação abrangente, colaboração com o cliente na negociação do contrato e resposta a mudanças quando se segue um plano (ver Recursos).

Os métodos ágeis se adaptam muito bem à onda de novas empresas baseadas na Internet (também conhecidas como Web 2.0). O desenvolvimento ágil de software permite que essas novas empresas consigam mais por menos e realizem projetos significativos com equipes pequenas e orçamentos baixos. As iterações curtas e os princípios de funcionamento do software se refletem em uma prática chamada beta constante, por causa de vários produtos Google que têm a palavra "beta" incorporada ao logotipo.

Entretanto, os métodos ágeis não são adequados a todas as situações. Funcionam bem para equipes pequenas, reunidas no mesmo local, que enfrentam condições de mudanças rápidas. Embora existam casos em que a aplicabilidade do desenvolvimento ágil de software é questionável como no desenvolvimento distribuído com terceirização internacional meus cinco anos bem-sucedidos de experiência com a aplicação dos princípios do desenvolvimento ágil em equipes distribuídas prova que isso é possível e que dá um retorno ótimo quando usado corretamente.

"OS MÉTODOS ÁGEIS NÃO SÃO ADEQUADOS A TODAS AS SITUAÇÕES. FUNCIONAM BEM PARA EQUIPES PEQUENAS, REUNIDAS NO MESMO LOCAL, QUE ENFRENTAM CONDIÇÕES DE MUDANÇAS RÁPIDAS.

Existem outros cenários em que o uso dos processos de desenvolvimento ágil de software permanece questionável. Temos como exemplo disso as equipes grandes de desenvolvimento (mais de 20 pessoas trabalhando em um projeto independente), sistemas em que a previsibilidade é muito

Figura 1 Integração ponto a ponto e de barramento de serviços

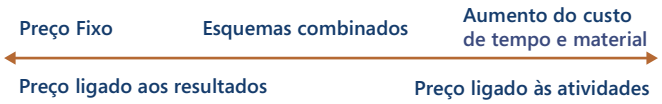


Figura 2 Três camadas de integração

importante (aplicativos críticos para a vida) e ambientes burocráticos. Não abordaremos esses cenários neste artigo e, além disso, supomos que a empresa tem uma cultura corporativa que favorece o desenvolvimento ágil e pretende aplicar as idéias apresentadas aqui a equipes de software com menos de 20 pessoas (isto é, 20 pessoas em um determinado projeto ou equipe, não na equipe inteira de desenvolvimento). Em vez disso, abordaremos a aplicação de métodos ágeis ao desenvolvimento distribuído em geral e à terceirização internacional em particular.

Combinando as Tendências

O desenvolvimento de software em outros países representa uma gama de atividades diferentes, desde a contratação de um desenvolvedor internacional pelo *rentacoder.com*, por um lado, até os trabalhos de bilhões de dólares com as empresas dos EUA que têm subsidiárias estrangeiras, por outro. Alguns desses trabalhos são feitos de forma a impedir que as empresas usem um processo de desenvolvimento ágil de software, mesmo se uma das partes quiser isso.

Para implementar um processo ágil, o modelo de terceirização selecionado deve incentivar as comunicações e a colaboração, pressupor que haverá flexibilidade e justificar liberações freqüentes. Embora seja possível aplicar dezenas de critérios aos trabalhos de terceirização, há poucos que são tão importantes para as nossas discussões que o modelo de determinação de preço. Consulte a Figura 1 para ver o mapeamento dos esquemas de preço mais comuns.

Resultados previsíveis implicam processos previsíveis. Na Figura 2, você vê grupos do processo de desenvolvimento alinhados na escala preditiva/adaptativa. Se usarmos um critério adaptativo/preditivo, os esquemas previsíveis mais próximos à extremidade esquerda da escala (ver a Figura 1) requerem mais previsibilidade do processo de desenvolvimento de software, ao passo que os processos de desenvolvimento ágil ficam no lado oposto do espectro dos processos de desenvolvimento de software.

A preocupação com a previsibilidade está fortemente relacionada a atividades com *preço e escopo fixos*. Quando se aplica esse tipo de contrato a um trabalho de terceirização, o cliente e o fornecedor ficam naturalmente presos a processos previsíveis de desenvolvimento de software e suas conseqüências. Portanto, esse contrato não se ajusta bem ao desenvolvimento ágil de software. Ao projetar um trabalho de terceirização, lembre-se de que a capacidade de responder a mudanças favorece o uso de modelos de determinação de preço tais como tempo e material, dando flexibilidade ao cliente e ao fornecedor, além de se ajustar melhor ao desenvolvimento ágil.

Ao estruturar o trabalho, leve em conta a influência do esquema selecionado sobre as comunicações e a colaboração. Um processo de desenvolvimento ágil requer um ambiente aberto, uma boa integração da equipe, compartilhamento de metas em comum, compreensão do valor de negócio e comunicação freqüente. Quanto mais barreiras houver entre engenheiros, clientes, usuários, gerentes e outros grupos envolvidos, maior será a dificuldade de se estabelecer uma base para o desenvolvimento ágil de software, o que significa reduzir a quantidade de intermediários, permitindo o máximo de transparência e integração entre as equipes.

Os grandes agentes conseguem isso estabelecendo filiais em outros países algo que faz sentido economicamente se a companhia quiser mais de 100 engenheiros no seu centro de desenvolvimento no exterior. A quantidade exata depende muito do outro país e de fatores como a facilidade que a empresa terá para recrutar pessoas talentosas lá.

Ao considerar essa alternativa, não repita os erros de algumas empresas de pequeno e médio porte, que subestimaram despesas ocultas como tempo despendido pela alta gerência, orçamento de viagens, honorários advocatícios locais e outros custos. Além disso, o rápido crescimento das economias dos países em desenvolvimento provoca escassez de talentos, dificultando a vida de estrangeiros recém-chegados. Embora os agentes locais tenham uma rede de contatos melhor em sua comunidade local, as filiais de grandes empresas podem resolver esse problema por meio de marcas conhecidas, maiores salários e melhores benefícios sociais.

"ESCOLHER O MODELO CORRETO TAMBÉM É MUITO IMPORTANTE, MAS NÃO GARANTE O SUCESSO"

Alguns luxos costumam estar fora do alcance das empresas pequenas para as quais é conveniente ter apenas 15 desenvolvedores no exterior. Agentes bem-sucedidos de pequeno e médio porte superam a barreira dos custos do trabalho de instalação de um escritório remoto usando equipes de desenvolvimento dedicadas, ODCs (Centros de Desenvolvimento no Exterior), modelos BOT (Criar/Operar/Transferir), escritórios virtuais etc. Independentemente do nome e dos detalhes, há um ponto em comum entre esses modelos: o fornecedor do exterior está mais focado em proporcionar estrutura física, jurídica, de TI e de RH ao cliente do que em participar do processo de desenvolvimento. A responsabilidade da entrega do projeto nesses trabalhos é dividida entre o cliente e o fornecedor.

Para obter todos os benefícios nessas situações, os engenheiros devem encarregar-se de um cliente, e as taxas de retenção da equipe devem ser boas nas duas empresas. O fornecedor deve proporcionar comunicações transparentes ao cliente em todos os níveis, inclusive nas comunicações entre desenvolvedores. É necessário que se fale a língua sem precisar de um tradutor.

Em trabalhos de sucesso, muitas vezes vemos que, embora as pessoas que trabalham em escritórios remotos sejam pagas pelo fornecedor, elas se associam mais ao cliente e compartilham os valores e a cultura corporativa das duas empresas.

Usando as Práticas e Ferramentas Corretas

Há práticas bem conhecidas que são usadas pelas equipes de desenvolvimento de software bem-sucedidas: padrões de código em comum; um servidor de controle de fonte; scripts de um clique, no estilo "criar e distribuir"; integração contínua; teste das unidades; controle de bugs; padrões de projeto e blocos de aplicativos. Essas práticas devem ser aplicadas com mais rigor às equipes distribuídas do que às equipes locais.

Considere, por exemplo, a integração contínua. Pode ser extremamente frustrante vir para o trabalho e pegar um build com defeito no servidor de controle de fonte quando a pessoa responsável está a milhares de quilômetros de distância e pode estar dormindo no momento. Esse problema pode não ser grande se a tarefa for realizada pelo cara da sala ao lado, mas pode ser enorme em um cenário distribuído, prejudicando a produtividade e as comunicações. É possível minimizar esses riscos implementando práticas de integração contínua em toda a equipe e instalando o servidor correspondente (como Microsoft Team Foundation Server, CruiseControl.NET e CruiseControl).

Equipes que trabalham na plataforma Microsoft .NET estão em ótima situação com os recursos oferecidos pelo Microsoft Visual Studio Team System que se integra facilmente ao sistema existente. Você tem o Microsoft Solutions Framework para Desenvolvimento Ágil prescritivo e as ferramentas de suporte. Esse produto é extremamente útil para equipes que precisam de mais orientação devido ao desenvolvimento ágil em ambientes distribuídos. Para as equipes experientes, é uma solução integrada que proporciona um ROI excelente.

O WSS (Windows SharePoint Services) é outro produto Microsoft que proporciona um grande valor às equipes distribuídas. Os wikis se ajustam naturalmente ao desenvolvimento ágil em equipes distribuídas e o auxiliam, e há um planejamento para que a próxima versão do WSS tenha o wiki entre seus avanços. O WSS também está fortemente integrado ao Visual Studio Team System, o que faz dele a melhor opção para o portal da Web da equipe.

Do ponto de vista da infra-estrutura de TI, eu recomendo o uso de uma VPN (Rede Privada Virtual), dando às equipes um acesso igualitário aos recursos compartilhados. O ambiente da VPN, por ser menos restrito que uma rede pública, permite o uso de recursos como o compartilhamento do aplicativo Windows Live Messenger, chamadas de vídeo e voz, assistência remota e quadro branco (whiteboard).

Comunicação, Comunicação e Comunicação

Quando se trabalha remotamente, pequenos mal-entendidos se transformam rapidamente em problemas maiores. No desenvolvimento distribuído, os gerentes devem prestar atenção nas práticas de comunicação, algo que às vezes eles omitem sem sofrer conseqüências negativas no desenvolvimento local. Essa atenção envolve relatórios regulares (diários/semanais) e reuniões de atualização de status, que permitem aos integrantes da equipe sincronizar-se, discutir as realizações e revelar problemas. Os gerentes também devem procurar criar relações pessoais nas equipes, por meio de reuniões de apresentação, visitas ao local e outros métodos.

Em trabalhos de terceirização internacional, os gerentes de desenvolvimento devem estar cientes das barreiras ligadas a idioma, cultura e fuso horário e encontrar maneiras de superar esses obstáculos. De forma lenta mas constante, a globalização apaga as diferenças culturais no ambiente profissional, mas ainda há casos em que as diferenças culturais causam confusão. Há várias questões específicas de cada país nesse aspecto, mas isso está fora do escopo desta explanação. Os problemas ligados ao idioma são muito mais fáceis de detectar, mas isso não significa que são mais fáceis de resolver. Quando as empresas enfrentam barreiras lingüísticas, é comum e altamente recomendável que as empresas ofereçam aulas de idiomas aos funcionários. Na maioria dos países que participam do desenvolvimento internacional de software, os profissionais estão motivados para aprender inglês; portanto, geralmente são as pessoas desses locais que têm aulas de idiomas.

"UM PROCESSO DE DESENVOLVIMENTO ÁGIL REQUER UM AMBIENTE ABERTO, UMA BOA INTEGRAÇÃO DA EQUIPE, COMPARTILHAMENTO DE METAS EM COMUM, COMPREENSÃO DO VALOR DE NEGÓCIO E COMUNICAÇÃO FREQÜENTE"

Especificamente, as variações de fuso horário dificultam o processo. Entretanto, em países com o setor de terceirização desenvolvido, os engenheiros de software geralmente estão dispostos a adaptar o horário de trabalho para trabalhar com parceiros do exterior. Existem duas estratégias para lidar com diferenças de fuso horário. A primeira é separar as equipes de acordo com a atividade por exemplo: ficar com a garantia de qualidade e os gerentes de produto na empresa e os desenvolvedores no exterior. Esse arranjo permite implementar um ciclo, no qual os desenvolvedores implementam requisitos fixos e novos enquanto seus colegas dormem e vice-versa. É claro que deve haver intersecções nos horários de trabalho (no início/final de um dia de trabalho). A segunda abordagem é dividir os projetos em blocos e procurar atribuir cada bloco a um local, delegando a maior quantidade possível de funções a esse local. A segunda abordagem força uma melhora na comunicação e, conseqüentemente, é mais adequada para o desenvolvimento ágil mas as duas funcionam e, às vezes, não há escolha.

Escolher o modelo correto também é muito importante, mas não garante o sucesso. É altamente recomendável que pelo menos uma das partes tenha experiência com o desenvolvimento, de preferência em ambiente distribuído. A falta de comunicação "cara a cara", juntamente com as diferenças de horário, cultura e idioma, requer atenção e esforços adicionais para obter os resultados desejados. As vantagens de ter um bom parceiro no exterior economia de custos, aumento da equipe sob demanda e tarefas relacionadas à infra-estrutura da terceirização (que podem ser resumidas como "obter mais por menos") são bem mais significativas que o investimento na criação de relações produtivas. Esse saldo positivo seria improvável sem as ferramentas modernas, que são possíveis devido à grande infraestrutura de comunicações que hoje está disponível globalmente.

Recursos

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

Manifesto for Agile Development
http://agilemanifesto.org

Sobre o Autor

Andrew Filev (MCA, MVP) é o vice-presidente responsável pelas operações internacionais da Murano Software. Ele estabeleceu centros de desenvolvimento no exterior, e também lidera e motiva as equipes. Excelente comunicador, Andrew preenche as lacunas entre culturas diferentes e cria parcerias duradouras com os clientes.

Modelagem Orientada a Serviços para Sistemas Conectados - Parte 2

por Arvindra Sehmi e Beat Schwegler

Resumo

Como arquitetos, podemos adotar uma nova maneira de pensar para forçar essencialmente uma consideração explícita de artefatos de modelos de serviços nos processos de projeto, o que nos ajuda a identificar os artefatos corretamente e no nível de abstração certo para atender e alinhar as necessidades de negócios das nossas organizações. Na parte 1, oferecemos uma abordagem em três etapas para sistemas orientados a serviços conectados por modelagem de uma maneira que promova um alinhamento exato entre a solução de TI e as necessidades dos negócios. Nós examinamos a perspectiva atual do pensamento orientado a serviços e explicamos como o pensamento atual e a conceitualização deficiente da orientação a serviços resultou em muitas falhas e, de modo geral, em baixos níveis de ROI (Retorno do Investimento). Também examinamos os benefícios da inserção de um modelo de serviços entre os modelos de tecnologia e de negócios convencionais que são familiares para a maioria dos arquitetos e discutimos a metodologia Microsoft Motion e o mapeamento de capacidades para identificar capacidades de negócios que possam ser mapeadas para serviços. Nesta segunda parte mostraremos como implementar esses serviços mapeados.

Na parte 1, terminamos a nossa discussão com o processo SOAD (Design e Análise Orientada a Serviços) pragmático que é utilizado para extrair todas as peças necessárias para construir o seu modelo de serviços. Essa extração inclui contratos de serviços, SLAs (Contratos de Nível de Serviço) derivados da SLE (Expectativa de Nível de Serviço) definida para cada capacidade de negócios e os requisitos de orquestração de serviços. Com um modelo de serviços detalhado e estreitamente alinhado com o modelo de negócios e dele derivado, você deverá estar bem situado para mapear o modelo de serviços para um modelo de tecnologia que identifique como cada serviço será implementado, hospedado e implantado.

Utilizando a abordagem anterior e criando o modelo de serviços, você pode passar para o seu departamento de TI esquemas de dados, contratos de serviços e requisitos de SLA. No entanto, antes de criar o serviço, deve-se levar em consideração o suporte à autonomia do serviço no nível de tecnologia, separando claramente as interfaces da implementação e dos mecanismos de transporte de base. Desenvolver estratégias de implementação de serviço que separem o endpoint do serviço da implementação do serviço ajuda a planejar para a mudança. A seleção dos hosts apropriados e das opções de gerenciamento de serviço para atender SLAs específicos também exige uma consideração cuidadosa. As suas opções nessas áreas são capturadas e definidas pelo modelo de tecnologia.

Criando um Modelo de Tecnologia

O modelo de tecnologia consiste em vários artefatos: interface de serviços, implementação de serviços, host de serviços, gerenciamento de serviços e mecanismo de orquestração. Examinaremos cada um em detalhe.

A **interface de serviços** especifica como um documento ou uma mensagem pode ser recebido. A interface permite especificar quais transportes serão fornecidos, independentemente da implementação. Mais de uma interface de serviços pode implementar um contrato de serviços, mas toda interface de serviços implementa um binding especifico como SOAP em HTTP.

Se necessário você pode fornecer várias interfaces de serviços utilizando transportes diferentes. Por exemplo, você poderá ligar uma única interface a um transporte de serviços da Web e também a um transporte de filas de mensagens do Windows. Cada transporte fornece capacidades referentes como interoperabilidade e suporte transacional e diferentes restrições. Por exemplo, a fila de mensagens não suporta diretamente o padrão solicitação/resposta. Pelo menos uma interface deve ser fornecida para interoperabilidade garantindo que a interface esteja em conformidade com o WS-I BP (Web Services Interoperability Basic Profile) versão 1.x.

A **implementação de serviços** é a implementação de uma capacidade de negócios independente do respectivo host. Ela também não deve ter dependências nas suas interfaces. A implementação de serviços pode chamar outros serviços utilizando proxies dependentes de binding e, para atingir proxies que não são dependentes deve cria-los utilizando fábricas.

O **host de serviços** fornece um endpoint para as interfaces de serviços. A escolha do host deve ser feita com base em requisitos do SLA especificado. Por exemplo, se a empresa espera operações 24h por dia, sete dias por semana, essa característica tem um impacto imenso na escolha do host, e, nesses cenários, tecnologias de enfileiramento como Microsoft Message Queue ou SQL Server Service Broker geralmente são necessárias. No modelo de tecnologia, os hosts representam opções de custo. No modelo de negócios, o SLE representa o valor. Se for possível mapear a escolha do host de volta até um SLA capturado pelo modelo de serviços e no final até um SLE voltado aos negócios, o custo de um host determinado pode ser justificado e quantificado.

Um benefício complementar de mudar para serviços e para longe de aplicativos que funcionem como silos é que se você tiver uma capacidade de negócios específica que exija operações 24h por dia durante sete dias por semana, poderá mudar esse serviço para um host que forneça o desempenho e a redundância necessários.

Você poderá conseguir utilizar hosts menos dispendiosos para outras capacidades. Com a abordagem baseada em silos, você é forçado a selecionar um host para o aplicativo inteiro.

Para o **artefato de gerenciamento de serviços**, uma ação apropriada deverá ser tomada se um SLA não puder ser atendido e, para dar suporte a essa ação, você deverá ser capaz de monitorar e gerenciar o serviço. As interfaces de serviços, as implementações e os hosts devem ser instrumentados, por exemplo, utilizando o WMI (Windows Management Instrumentation). O MOM (Microsoft Operations Manager) pode ser utilizado em seguida para monitorar e gerenciar o serviço, enquanto que o SMS (Microsoft Systems Management Server) é utilizado para dar suporte ao gerenciamento de mudanças e de configuração no ecossistema "de fronteira" do serviço. O mecanismo de orquestração que você escolher também deverá fornecer suporte de monitoramento como a funcionalidade BAM (Business Activity Monitoring) fornecida pelo BizTalk Server. No futuro, o WS-Management irá exercer uma função central no gerenciamento de recursos independentemente da plataforma de hospedagem e gerenciamento.

Para o **mecanismo de orquestração**, os requisitos de orquestração de artefatos devem ser definidos de uma maneira independente da plataforma pelo modelo de serviço, mas no final você precisa utilizar um mecanismo de orquestração específico. O modelo de tecnologia identifica a plataforma de destino, como Microsoft BizTalk Server.

Criando o modelo de tecnologia você pode capturar explicitamente os artefatos anteriores, mas como deve ser abordado o desenvolvimento do serviço? Como você mapeia esses artefatos para uma implementação de serviços e como você implementa o contrato especificado pelo modelo de serviços? Discutiremos em breve uma abordagem que ajuda a construir os serviços de uma maneira que satisfaça os fundamentos de serviços e os princípios de modelagem discutidos anteriormente.

Criando Serviços Hoje

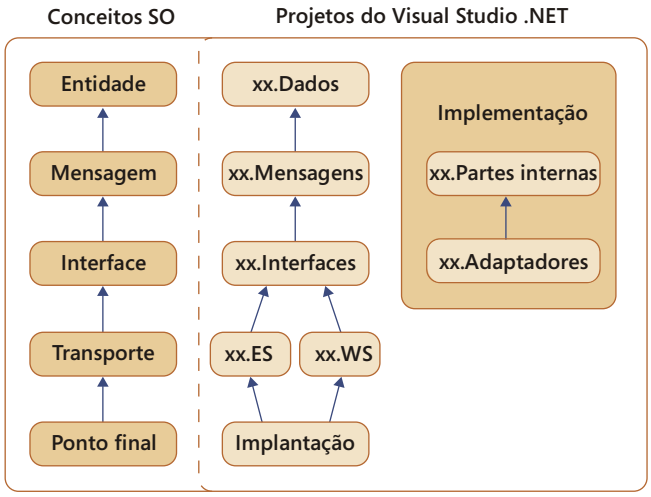
Dado um conjunto de artefatos de serviços conceituais, como você pode defini-los em código, e como o código deve ser organizado dentro do Visual Studio 2005? O ponto central a ser considerado é a necessidade da separação entre dados, mensagens, interfaces, partes internas da implementação e ligações de transporte. A FigurA 1 mostra uma abordagem de mapeamento de artefatos de serviços para projetos e soluções Visual Studio 2005.

"CADA TRANSPORTE FORNECE CAPACIDADES REFERENTES COMO INTEROPERABILIDADE E SUPORTE TRANSACIONAL E DIFERENTES RESTRIÇÕES"

Os espaços reservados "xx" mostrados na Figura 1 representam um namespace apropriado baseado no nome do serviço — por exemplo, OrderService. Utilizar um único arquivo de solução Visual Studio 2005 para um serviço determinado e como um recipiente para os vários projetos observados na Figura 1 é uma prática recomendada. Iremos analisar esses projetos.

- O projeto **xx.Data** é utilizado para conter as definições do esquema de dados na conexão que são obtidas da entidade de modelo de serviços e também contém a representação CLR (Common Language Runtime) desse esquema.
- O projeto **xx.Messages** é utilizado para conter definições das mensagens necessárias para comunicar com o serviço, o que inclui mensagens de entrada e de saída. Uma mensagem é representada como um elemento do esquema que contém elementos do contrato de dados.

Figura 1 Mapeamento de artefatos de serviço para projetos do Visual Studio



- O projeto **xx.Interfaces** contém as definições de interface.
- Os projetos **xx.ES e xx.WS** contêm endpoints de transporte para interfaces. No exemplo mostrado na Figura 1, xx.ES fornece um transporte de serviços corporativos e xx.WS fornece um transporte de serviços da Web.
- Os projetos **xx.Adapters e xx.Internals** contêm a implementação do serviço. Um adaptador contido no projeto xx.Adapters fornece um nível de separação entre a interface e a implementação interna. O adaptador analisa a mensagem de entrada e transmite as entidades de dados relevantes para o código de implementação interno. O adaptador também embala a saída de dados da implementação na mensagem de resposta. Observe que com essa abordagem a implementação não sabe nada sobre as mensagens transmitidas por meio da interface, o que permite alterar a infra-estrutura de mensagens sem causar impacto na implementação de serviços interna.

O endpoint mostrado na Figura 1 é puramente um artefato para instalação e é dependente da opção de transporte. Por exemplo, com um transporte de serviços da Web, você implementaria o seu serviço em um servidor da Web IIS (Internet Information Services) executando ASP.NET.

Adaptadores. O adaptador é um artefato importante que permite separar a implementação das mensagens. Também é um bom lugar para realizar transformações de mensagens se for necessário. Por exemplo, no adaptador você poderia transformar uma representação externa da ID de um cliente (por exemplo, 10 caracteres) na sua representação interna (por exemplo, 12 dígitos). Criando um novo adaptador por versão do seu serviço, o adaptador também fornece uma maneira de atualizar a versão dos serviços sem afetar a implementação ou as interfaces de serviços publicadas anteriormente.

Versão. Ao considerar a versão, é necessário distinguir entre versão do contrato abstrato ou concreto e a versão da implementação interna. Uma meta arquitetônica importante é fornecer versões independentes entre os dois. O contrato mesmo consiste nos dados, na mensagem, na interface e nos endpoints. Você pode gerenciar cada um desses artefatos de forma independente se você arquitetar o serviço de forma que cada artefato da composição refira a uma única versão do artefato composto. Por exemplo, o contrato de mensagens que é composto pelo contrato de dados deve referir-se a exatamente uma versão do contrato de dados. Essa referência deve ser verdadeira para o XSD/WSDL e a representação

Os contratos de dados e de mensagens devem ter a versão de acordo com a versão XML e a política de capacidade de extensão (consultar Recursos). Você pode fazer a versão das interfaces de serviços fornecendo uma nova interface que herde da anterior (CLR) e que esteja mapeada para um novo tipo de porta WSDL.

Seis Etapas para Criar um Serviço

Para chegar a esses projetos do Visual Studio 2005 e criar um serviço atualmente utilizando essa abordagem, adote este processo de seis etapas:

1. Projetar o contrato de dados e de mensagens.
2. Projetar o contrato de serviços.
3. Criar os adaptadores.
4. Implementar as partes internas dos serviços.
5. Conectar as partes internas aos adaptadores.
6. Criar as interfaces de transporte.

Vamos observar os detalhes de cada etapa. Para a primeira etapa — projetar o contrato de dados e de mensagens — tome o esquema de dados canônico que define a representação dos dados na conexão (saída do projeto e análise orientadas a serviços) e defina as classes de dados e as classes de mensagens. Existem duas abordagens para criar esquemas de dados e classes de dados. Utilizando a abordagem de "primeiro esquema," você pode criar um esquema XSD e, em seguida, utilizar uma ferramenta como Xsd.exe ou XsdObjectGen.exe para gerar as classes de dados automaticamente. Como alternativa, se você preferir uma abordagem de "primeiro código", poderá definir as suas classes de dados em C# ou Visual Basic e, em seguida, utilizar Xsd.exe para criar um esquema XSD equivalente. Aqui está um exemplo de um contrato de dados:

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

Após definir as classes de dados você pode definir as mensagens de entrada e de saída, que contêm as classes de dados como seu corpo. Por exemplo, este trecho de código mostra uma mensagem de entrada chamada OrderMessage para um serviço de pedido hipotético que contém um pedido como sua corpo:

```
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;
   }
}
```

Na etapa 2 — projetar o contrato de serviço — defina o contrato de serviço abstrato utilizando uma abordagem de "primeiro WSDL" ou definindo as interfaces utilizando C# ou Visual Basic. As interfaces definem quais mensagens o seu serviço recebe e quais mensagens (se for o caso) ele devolve. Para gerar a interface do WSDL, é necessário usar Wsdl.exe com o parâmetro /si:

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

Observe que esse parâmetro funciona somente com o Microsoft .NET Framework versão 2.0. Este próximo exemplo define uma interface para o serviço de pedidos que define um único método PlaceOrder() que aceita uma mensagem OrderMessage e retorna uma mensagem

```
namespace Interfaces
{
   using MessageContracts;

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

Observe que neste caso é utilizado um padrão de mensagem de solicitação/resposta e, portanto, você precisa garantir que o tempo de processamento da mensagem (o tempo entre a solicitação e a resposta) deve ser de subsegundos. Se esse tempo de processamento não puder ser garantido, utilize outro padrão de troca de mensagens como troca de mensagens duplex, que correlaciona duas mensagens unidirecionais com um padrão lógico de solicitação/resposta.

Para gerar o WSDL a partir da interface anterior, você pode criar uma classe de serviço da Web ASMX que implementa a interface e, em seguida, chama o serviço da Web transmitindo ?WSDL — por exemplo, http://localhost/Order-Service/OrderService.asmx?wsdl. Esse serviço da Web utiliza o WSDL a ser gerado e retornado do Web Service.

Na etapa 3 — criar os adaptadores — crie a classe de adaptadores que fornece o desaclopamento entre a interface e as partes internas. Essa classe garante que as partes internas não saibam nada sobre o contrato de mensagens. O adaptador desembala a mensagem de entrada e transmite os dados do corpo para a implementação interna realizando quaisquer transformações de formatos de dados necessárias da mensagem para o formato interno. De forma semelhante, no caminho de volta o adaptador realiza quaisquer transformações de formato necessárias nos dados retornados da implementação de serviços e as embala em uma mensagem de serviço de saída, se houver. Aqui está um exemplo de adaptador:

```
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(PlaceOrderM
sg);
      }
   }
}
```

Lista 1 A interface IOrderService

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

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

Observe que o adaptador está sempre em processo com o chamador, e a identidade do processo depende da escolha do host. Se a implementação interna for hospedada no IIS, a interface ASMX ("fronteira") instancia a parte interna do serviço. Se a parte interna for hospedada nos serviços de componente mas as chamadas chegam por meio de um serviço da Web ASMX, a interface ASMX delega a chamada para a interface de serviços de componente, que instancia o adaptador e transmite a mensagem para ele. Como todos os transportes implementam a mesma interface, você pode encadear as chamadas.

Na etapa 4 — implementar as partes internas dos serviços — crie a implementação das partes internas dos serviços. Existe uma única implementação independentemente do transporte ou dos transportes escolhidos. Este trecho de código mostra o código de estrutura necessário para fornecer uma implementação do serviço de processamento de pedidos:

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

Observe como a implementação interna não tem conhecimento da mensagem. Ela só conhece o contrato de dados. Neste caso é transmitido um objeto de pedido único. Se você deseja ser completamente independente do formato da conexão, a parte interna não teria acesso nem mesmo ao contrato de dados. Nesse cenário, o adaptador mapearia o contrato de dados externos para os tipos de dados internos.

Na etapa 5 — conectar as partes internas aos adaptadores — chame a sua implementação interna a partir do código do adaptador. Observe que a mensagem não é transmitida para a implementação, o que separa a implementação interna do contrato de mensagens e permite que um seja alterado sem causar impacto no outro. Este trecho de código mostra novamente o código do adaptador, desta vez com um chamado para a implementação do serviço interno:

```
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(
                place OrderMsg.Order);
            return otm;
        }
    }
}
```

Na etapa 6 — criar as interfaces de transporte — ligue a interface de serviços abstratos definida na etapa 2 a um transporte específico. A Lista 1 mostra a interface IOrderService definida na etapa 2 ligada a um transporte de serviços da Web.

Utilizando Orientação Automatizada por Processo

Seguindo o processo de seis etapas discutido acima, você pode criar serviços que atendam todos os princípios e fundamentos analisados aqui. Porém, como todos os itens descritos no processo de seis etapas podem ser descritos por metadados, é possível automatizar grandes partes da geração dos serviços. Podem ser utilizadas ferramentas como GAT (Guidance Automation Toolkit) para ajudar a automatizar as tarefas (consulte Recursos). A orientação automatizada por processo é particularmente útil para realizar transformações entre artefatos XML e CLR semanticamente idênticos como classes XSD e CLR, transformações entre o tipo de porta WSDL e interfaces CLR, gerar adaptadores baseados na descrição de interface e gerar diferentes tecnologias de endpoint baseadas na descrição de interface.

A maneira antiga de pensar sobre orientação a serviços não está funcionando e uma nova maneira de pensar é necessária. Adotando essa nova maneira de pensar, como arquitetos podemos forçar a consideração explícita de artefatos do modelo de serviços no processo de projeto, o que ajudar a identificar os artefatos corretamente e no nível de abstração certo para satisfazer as necessidades de negócios e alinhar com elas.

De uma perspectiva de modelagem, a lacuna entre os negócios convencionais e os modelos de tecnologia é muito grande, o que constitui um fator de contribuição principal para o insucesso de muitas iniciativas de orientação a serviços. Nós apresentamos um modelo em três partes com a introdução de um modelo de serviços entre os modelos de negócios e de tecnologia para promover um alinhamento mais estreito dos nossos serviços com as necessidades dos negócios. Com um modelo de serviços detalhado e alinhado estreitamente com o modelo de negócios e dele derivado, você está bem posicionado para mapear o modelo de serviços para um modelo de tecnologia que identifica como cada serviço será implementado, hospedado e implantado. O mapeamento das capacidades e a metodologia Motion fornecem uma maneira efetiva de identificar as capacidades de negócios e, no final, os serviços. A decomposição dos negócios em capacidades fornece uma separação de alto nível dos contratos de serviço e não o contrário como geralmente é hoje.

Os sistemas conectados são instâncias de modelos inteiros, divididos em três partes, e respeitam os quatro princípios da orientação a serviços. Eles podem ser implementados de uma forma mais completa com a utilização dos cinco pilares das tecnologias de plataforma da Microsoft. Lembre-se do que perguntamos no início: Como fazemos para evitar cometer com as SOAs os mesmos enganos cometidos com iniciativas anteriores que ofereciam grandes esperanças? Como fazemos para garantir que a arquitetura de implementação escolhida relaciona-se ao estado real ou desejado dos negócios? Como fazemos para garantir uma solução sustentável que possa reagir à natureza de transformação dinâmica dos negócios em outras palavras, como ativamos e sustentamos negócios ágeis? Como fazemos para migrar para esse novo modelo de uma maneira refinada e em um ritmo que possamos controlar? E como fazemos para realizar essa mudança com uma boa percepção de onde podemos agregar o maior valor aos negócios desde o início?

A orientação a serviços como serviços da Web é somente a implementação de um modelo específico. A qualidade e os fundamentos do modelo determinam as respostas a essas perguntas.

Agradecimentos

Os autores desejam agradecer a Ric Merrifield, diretor do, Microsoft Business Architecture Consulting Group; David Ing, arquiteto de software independente; Christian Weyer, arquiteto da, thinktecture; Andreas Erlacher, arquitet da, Micr-soft Áustria; e Sam Chenaur, arquiteto da Microsoft Corporation pelos seus comentários sobre os primeiros rascunhos deste artigo. Desejamos também estender os nossos agradecimentos a Alex Mackman, tecnologista principal da CM Group Ltd., um excelente pesquisador e escritor que nos ajudou imensamente.

Recursos

"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*

Obtenha um estudo de caso sobre a metodologia Microsoft Motion enviando uma solicitação para motion@microsoft.com.

"Modelagem orientada a serviços para sistemas conectados Parte 1," Arvindra Sehmi e Beat Schwegler, O Jornal de Arquitetura, Edição 7, março de 2006. *http://www.thearchitecturejournal.net*

Sobre os Autores

Arvindra Sehmi é chefe de arquitetura corporativa na Microsoft EMEA Developer and Platform Evangelism Group. Ele se dedica à adoção de práticas recomendadas de engenharia de software corporativo por toda a comunidade de arquitetos e desenvolvedores da EMEA e lidera o evangelismo de arquitetura na EMEA para a indústria de serviços financeiros. Arvindra é editor emérito do Microsoft Architecture Journal. Ele é Doutor em Engenharia Biomédica e mestre em Administração.

Beat Schwegler é arquiteto na Microsoft EMEA Developer and Platform Evangelism Group. Ele dá suporte e consultoria a empresas em arquitetura de software e tópicos relacionados e é orador freqüente em eventos e conferências internacionais. Ele tem mais de 13 anos de experiência em arquitetura e desenvolvimento de software profissional e esteve envolvido em uma ampla variedade de projetos, de sistemas de controle de construção em tempo real e produtos finalizados de grande vendagem a sistemas CRM e ERP de grande escala. Nos últimos quatro anos, a sua atividade principal tem sido na área de orientação a serviços e serviços da Web.

ARC

Microsoft

THE ARCHITECTURE JOURNAL

Inscreva-se no endereço: www.architecturejournal.net