Découvrez la discipline, pratiquez son art et faites part de vos idées à www.ArchitectureJournal.net
Des ressources pour créer.

THE ARCHITECTURE JOURNAL™
Des données pour de meilleurs résultats
Journal 8

Conception des données

Microsoft®

Table des matières

THE ARCHITECTURE JOURNAL™
Des données pour de meilleurs résultats
Journal 8

Avant-propos

Cher architecte,

Bienvenue dans le numéro 8 de « *The Architecture Journal* », dont le thème est la conception de données. Selon moi, les architectes que nous sommes sous-évaluent fréquemment l'omniprésence des données dans la profession, notamment si l'on considère leur utilisation dans les applications et les systèmes déployés sur plusieurs zones géographiques, fuseaux horaires et entreprises.

Pour évoquer la disponibilité des données dans un système d'information, j'ai souvent recours à l'analogie de l'eau circulant dans les canalisations d'une maison. Lorsque l'on ouvre un robinet, on s'attend à obtenir sur-le-champ une eau filtrée et propre ainsi que, la plupart du temps, une pression satisfaisante. Dans cette analogie, les canalisations constituent l'infrastructure et l'eau les données. Pour ce qui est de ces dernières et de leur rôle dans l'architecture, c'est la même approche qui me vient à l'esprit : les données fournies aux utilisateurs doivent être propres, filtrées et livrées dans les délais convenus, qu'il s'agisse d'un simple courrier électronique, d'une fiche d'abonné ou d'un ensemble de données financières mensuelles.

Bien que ce numéro n'aborde pas toutes les techniques existantes, il contient plusieurs articles intéressants traitant de l'importance des données, qui ont été rédigés par des auteurs de renom.

Nous commencerons par celui de Roger Wolter, architecte solutions chez Microsoft et auteur de nombreux articles et livres sur SQL Server et SQL Server Service Broker (SSB). Il y évoque l'importance de la fiabilité des données, notamment dans le contexte de la conception des systèmes connectés.

Suivent Tim Ewald et Kimberly Wolk avec un article qui décrit quelques-uns des modèles qu'ils ont créés pour l'intégration des données dans le système XML nouvelle génération MTPS (MSDN TechNet Publishing System), à l'origine de MSDN2. Udi Dahan nous expliquera ensuite comment utiliser les principes d'agrégation d'entités pour obtenir une vue globale de nos données et apporter ainsi des réponses concrètes aux exigences premières de l'entreprise.

Puis, nous partagerons l'expérience de deux autres auteurs, Tom Fuller et Shawn Morgan, qui ont compris que la réplication de données pouvait se révéler un «antipattern » de l'architecture orientée services (SOA), notamment à la lumière des services et applications autonomes. Dion Hinchcliffe, CTO de Sphere of Influence, nous livrera quelques modèles de composition et de consommation des données, en particulier dans le domaine des « mashups » et des applications Web 2.0.

Paul Nielsen terminera notre série d'articles sur les données par un aperçu de « Nordic », un nouveau modèle hybride objet/relationnel qui augmente la flexibilité et les performances des bases de données par rapport aux modèles relationnels traditionnels.

Pour conclure ce numéro du Journal, Andrew Filev nous fera part de sa tentative de combiner une méthodologie souple avec un modèle de développement Offshore, avant qu'Arvindra Sehmi et Beat Schwegler ne nous livrent la seconde partie de leur série intitulée « Modélisation de services pour systèmes connectés ». Si vous n'avez pas lu la première partie, téléchargez le numéro 7 de « *The Architecture Journal* » sur www.architecturejournal.net.

Ceci conclut notre numéro sur le thème des données. J'espère que ces articles vous aideront à concevoir des systèmes de données aussi fluides que l'eau qui coule de vos robinets. Mais, je ne peux, bien sûr, pas garantir que vous ne vous mouillerez pas les mains !

Simon Guest

Fiabilité des systèmes connectés

par Roger Wolter

Résumé

Les applications réparties se composent de nombreux services faiblement couplés qui sont souvent répartis sur un réseau ; obtenir de hauts niveaux de fiabilité et de disponibilité pour ces applications pose un ensemble de défis architecturaux tout à fait unique. Par exemple, si une application s'arrête car l'un des dix services qui fonctionnent sur dix serveurs différents n'est pas disponible, le taux de défaillance pour l'application est environ dix fois supérieur à celui des services pris individuellement. Les défaillances des couches de données accentuent ce problème, car une source de données peut être utilisée par des dizaines de services. Cet article traite des questions de fiabilité qui doivent être prises en compte dans l'architecture d'une application à services connectés. Il explique également comment certaines des nouvelles fonctionnalités de SQL Server 2005 et des produits de messagerie de Microsoft permettent de résoudre ces problèmes.

La fiabilité est coûteuse en matériel, logiciels et maintenance. Il est donc capital de comprendre les exigences de votre application dans ce domaine. Rares sont les applications qui n'exigent pas un certain niveau de fiabilité, mais implémenter une application avec une fiabilité supérieure au niveau nécessaire peut s'avérer une perte de temps et de ressources. Il convient donc de bien cerner les questions liées à la fiabilité des systèmes connectés pour concevoir une solution capable de fournir le niveau requis tout en utilisant les ressources de manière appropriée.

Dans les architectures orientées services (SOA, service-oriented architecture) ou les systèmes connectés, les services communiquent les uns avec les autres par des formats de message bien définis, ce qui signifie que la fiabilité des applications est fortement influencée par la fiabilité de l'infrastructure de messagerie dont elles dépendent. Dans cet article, nous allons prendre l'exemple d'un distributeur automatique bancaire pour illustrer les différents degrés de fiabilité de la messagerie et la façon d'y parvenir. Le traitement des messages entre services est généralement plus complexe que dans les scénarios client/serveur, car dans ces derniers, l'utilisateur peut prendre des décisions sur la façon de gérer les diverses situations d'erreurs et les délais, tandis que le serveur qui lance l'échange de messages dans la messagerie serveur à serveur doit prendre toutes les décisions.

Importance de l'infrastructure

Les applications orientées service parviennent souvent à un meilleur débit en utilisant les services de *messagerie en mode asynchrone*. Avec la messagerie asynchrone, un service envoie un message à un autre service et continue à fonctionner sans attendre la réponse. Si ce système lui permet de traiter beaucoup plus de requêtes (car il ne perd pas de temps à attendre les réponses aux requêtes soumises aux autres services), il implique aussi que l'infrastructure de messagerie assume la charge de la livraison et du traitement des messages. Le choix d'un traitement synchrone ou asynchrone des messages dépend généralement des besoins métier de l'application.

À titre d'exemple, lorsqu'un client essaie de retirer de l'argent d'un distributeur automatique de billets, effectuer une requête asynchrone pour vérifier son solde et lui donner l'argent sans attendre la réponse n'est pas une décision commerciale judicieuse. Pour autant, ce scénario n'exclut pas forcément une requête asynchrone. Le distributeur peut émettre une demande de solde dès que le client sélectionne l'option de retrait et le laisser saisir un montant tandis que la demande de solde est traitée de façon asynchrone. Une fois la somme saisie et le solde reçu, le distributeur automatique peut décider ou non de remettre l'argent au client.

Examinons à présent la façon dont le distributeur automatique gère le message de demande de solde. Le service de distribution de billets envoie le message au service de gestion du compte pour obtenir le solde du compte du client, et trois situations sont possibles : le solde est renvoyé avec succès, une erreur est renvoyée, ou la demande expire car le résultat n'est pas renvoyé dans le délai imparti. Si un solde est renvoyé, le service du distributeur continue la transaction. En cas d'erreur, le distributeur a recours à sa logique applicative pour la résoudre : en utilisant parfois une copie mise en cache du solde ou en remettant ou non l'argent en fonction du montant demandé par le client. Le cas le plus épineux est celui du délai d'expiration. Un dépassement peut signifier que le message s'est perdu pendant la transmission, qu'il est arrivé au service du compte mais que la réponse a été retardée pour une raison quelconque ou encore que la réponse a été envoyée mais perdue au retour.

Dans la plupart des cas, la meilleure solution est de réessayer en espérant que cela fonctionne mieux. En effet, si le message a été perdu au cours du premier envoi, il parviendra sans doute à sa destination la fois suivante. Si la réponse a été lente, la première réponse peut arriver avant que la deuxième demande n'expire. Une nouvelle tentative devrait aboutir, à moins que le service de gestion du compte ne soit injoignable ou même arrêté. Selon la cause du problème, la demande de solde peut avoir été traitée plusieurs fois ou plusieurs résultats peuvent être reçus, mais tant que le service du distributeur est préparé pour gérer ce type de situation, il ne s'agit pas là de problèmes graves.

Si le message a été envoyé de façon asynchrone, il est possible que le service du distributeur ne dispose plus des informations nécessaires pour reconstruire le message lorsqu'une nouvelle transmission est requise. L'infrastructure de messagerie devra donc conserver une copie des messages qu'elle envoie et les renvoyer si nécessaire. Pour ce type de message qui correspond à une requête simple, il est conseillé de conserver une copie en mémoire, car si le distributeur automatique s'arrête de fonctionner, le client devra de toute façon recommencer. Si le service du distributeur automatique a besoin d'une réponse même après un arrêt inopiné, le message doit

être placé dans un stockage persistant et le système de messagerie doit le renvoyer lors du redémarrage. La figure 1 présente trois réponses possibles à une demande de solde.

Livraison du message

Pour la version synchrone de cette requête, un simple service Internet SOAP apporte le degré de fiabilité nécessaire. La gestion des erreurs et des dépassements de délais incombe au service du distributeur automatique ; c'est donc lui qui détermine le niveau de fiabilité pour la requête. Pour la version asynchrone de la demande de solde, le canal WS-RM de Windows Communication Foundation (WCF) ou le mode de remise rapide des messages de MSMQ apporte la fiabilité requise si la requête n'a pas besoin de supporter des défaillances système. Si, au contraire, le message doit pouvoir supporter un redémarrage du système, le mode de remise récupérable de MSMQ est un choix adéquat. Ces systèmes de messagerie gèrent les délais d'expiration et les nouvelles tentatives nécessaires pour livrer le message et la réponse de sorte que le service du distributeur n'a pas à inclure cette logique.

Maintenant que nous comprenons les problèmes de fiabilité tels qu'ils se posent pour une demande de solde, passons à la demande de modification du solde qui intervient après le versement de l'argent au client. Les exigences de fiabilité pour ce message sont nettement supérieures, car s'il n'est pas livré avec succès, la banque aura effectué un versement sans avoir réduit le solde du compte client. Certes, le client ne s'en plaindra pas, mais la banque y trouvera sans doute à redire si cette situation se reproduit souvent.

Le message de modification du solde doit être remis et traité en dépit de défaillances du réseau ou du système. Le mécanisme de réémission d'un message persisté en mémoire ne convient évidemment pas à cette tâche puisqu'une défaillance système entraîne la perte de la copie en mémoire du message qui est conservé pour les nouvelles tentatives. Garder une copie persistante pour les nouvelles tentatives peut également poser problème, car plusieurs copies du message risquent d'être livrées. Si le message réduit le solde du client de 200 euros, la livraison multiple de ce message entraînera un vif mécontentement du client.

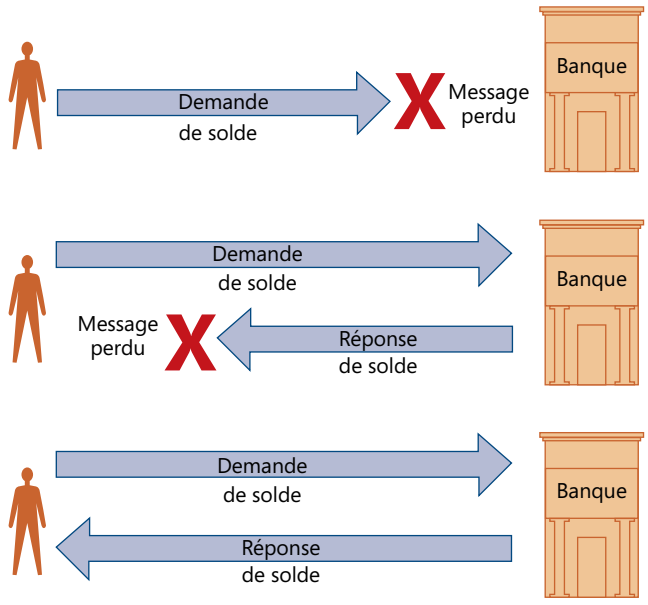
Les nouvelles tentatives nécessaires pour assurer une livraison fiable du message ne pourront satisfaire cet impératif que si les messages sont idempotents. Un message *idempotent* peut être livré plusieurs fois sans violation des contraintes métier de l'application. Certains messages sont, par nature, idempotents. Par exemple, un message qui modifie l'adresse d'un client peut être exécuté plusieurs fois sans effets néfastes. S'agissant des messages qui ne le sont pas par nature, le système doit les rendre idempotents pour éviter tout dommage pouvant survenir lorsqu'un message est traité plus d'une fois (voir les figures 2 et 3). Cette transaction se fait normalement en conservant une liste des messages déjà traités. Si le même message est reçu plus d'une fois, il n'est pas traité plusieurs fois.

La plupart des systèmes de messagerie ne conservent pas de copie des messages entrants, mais l'émetteur attribue un identificateur à chaque message et le destinataire garde la trace des identificateurs qu'il a déjà traités. Ces derniers doivent être conservés jusqu'à ce que le destinataire soit certain que l'émetteur s'est débarrassé du message, car si la source de la transaction échoue, le message peut être renvoyé plusieurs jours après, lors du redémarrage de l'émetteur. La liste des messages traités doit également être persistante, car le service de destination risque d'échouer entre deux nouvelles tentatives. La destination doit aussi garder trace du message qui a été envoyé en réponse au message entrant, car la source continue d'envoyer le message d'origine jusqu'à ce qu'elle reçoive la réponse et la nouvelle tentative peut être due au fait que le message de réponse d'origine a été perdu.

Choix de l'infrastructure de messagerie

Pour obtenir ce niveau de fiabilité, il est indispensable de disposer d'une infrastructure de messagerie fiable, sauf si vous êtes prêt à gérer vous-même la persistance et à garder une trace des messages dans la logique de votre service. Microsoft offre trois options pour cette infrastructure : la messagerie récupérable de MSMQ, SQL Server Service Broker et BizTalk.

Figure 1 Demande de solde



Votre choix dépendra de vos exigences de fiabilité et de la conception de l'application.

Service Broker et BizTalk offrent un stockage plus fiable des messages que MSMQ, car ils les stockent dans une base de données SQL Server tandis que MSMQ les stocke dans un fichier. Si votre application peut fonctionner avec des pertes de messages occasionnelles lorsqu'un fichier est perdu ou corrompu, MSMQ devrait satisfaire vos besoins. Certaines applications MSMQ offrent une protection contre la perte potentielle de messages en en stockant une copie dans une base de données. Si vous souhaitez stocker les messages, l'utilisation de Service Broker, qui effectue nativement cette opération dans la base de données, est nettement plus efficace.

Service Broker et BizTalk offrent globalement le même degré de fiabilité et de défense contre les pertes de messages, car ils les stockent en base. Le traitement des messages par Service Broker est intégré dans la logique de serveur de la base de données, de sorte que cet outil communique directement depuis le processus SQL Server vers les sockets TCP/IP, ce qui est beaucoup plus efficace que l'approche de BizTalk basée sur un processus externe qui appelle la base de données pour stocker les messages dans une table. Même si Service Broker peut fournir un plus grand nombre de messages par seconde que BizTalk, ce dernier offre de nombreuses fonctionnalités dont Service Broker est dépourvu : transformation des messages, routage des messages dépendant des données, transports multiples des messages ou encore orchestration.

En règle générale, si la fiabilité du transfert des messages entre les bases de données est la seule priorité de votre application, Service Broker est la meilleure solution, car plus légère et plus performante que BizTalk pour la transmission des messages. En revanche, si votre application exige les fonctionnalités de manipulation des messages, d'intégration de données ou d'orchestration offertes par BizTalk, alors Service Broker n'est sans doute pas adapté.

La messagerie récupérable de MSMQ n'offre pas les niveaux de fiabilité proposés par les options basées sur SQL Server, mais elle a l'avantage de ne pas exiger SQL Server aux deux points terminaux de la messagerie. S'il ne vous est pas possible d'implémenter une base de données aux deux points terminaux et si les exigences de fiabilité de l'application peuvent être satisfaites par MSMQ, alors MSMQ est certainement l'infrastructure de messagerie la mieux adaptée à vos besoins. Toutefois, si les deux services de communication exigent un stockage des données, la fiabilité supplémentaire que représente le stockage des messages dans la base de données est intéressante.

Figure 2 Nouvelle tentative sans gravité d'une demande de retrait d'argent

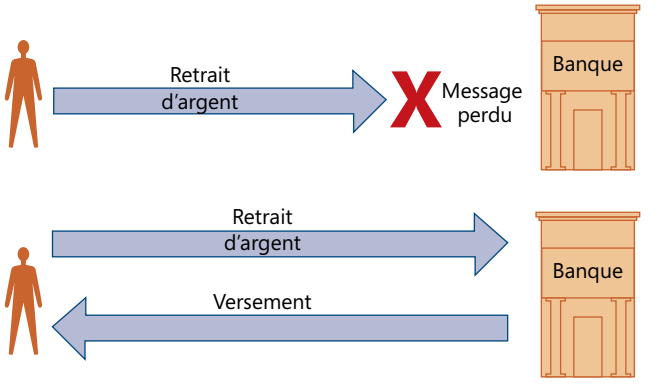
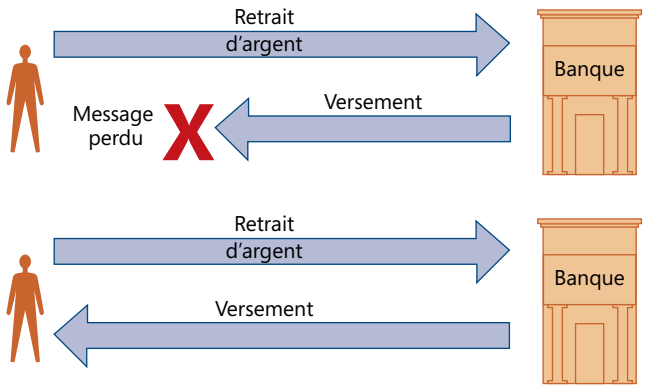


Figure 3 Nouvelle tentative de demande de retrait où le client perd de l'argent



Dans notre exemple de distributeur automatique de billets, le service du distributeur exige le stockage local des données pour des raisons d'audit, le stockage local des données pour les opérations hors-ligne et le stockage des données de référence. Il y aura donc sans doute une base de données dans le distributeur automatique et l'une des options basées sur SQL Server est adéquate. Le choix entre Service Broker et BizTalk dépend des exigences autres que celles liées à la messagerie, des ressources disponibles au distributeur automatique et des besoins en volume de messages.

Fiabilité d'exécution

Nous avons abordé le thème de la fiabilité dans la remise des messages entre plusieurs services et nous avons constaté, sans surprise, que le niveau de fiabilité exigé dépend de l'objet de l'application et de l'importance des données liées aux messages pour l'application. Nous allons partir du principe que les messages sont transférés vers un service avec le degré de fiabilité requis et examiner les exigences de fiabilité pour que ledit service puisse traiter les messages.

L'une des exigences particulières d'un service qui traite les messages asynchrones est que la réception d'un message depuis la file d'attente soit une opération de type « pull ». En d'autres termes, lorsqu'un message arrive dans la file d'attente, il y reste jusqu'à ce qu'une application exécute une opération de réception pour récupérer et traiter le message. Cette exigence implique qu'un service asynchrone doit être exécuté lorsque la file d'attente contient des messages. La méthode la plus courante pour répondre à cette exigence est de faire en sorte que le service soit géré par le Gestionnaire de contrôle des services de Windows (SCM). SCM s'assure que le service est lancé au démarrage de Windows. Il peut aussi être configuré pour le redémarrer en cas d'une défaillance quelconque.

Cette configuration, qui offre généralement le niveau de fiabilité requis, représente la solution de choix lorsque les messages arrivent à un débit constant ; elle peut induire des problèmes si la charge des messages varie fortement. Si le service est configuré avec suffisamment de ressources pour gérer les charges de pointe, il gaspillera ces ressources lorsque la charge sera faible. Et s'il est configuré pour gérer la charge moyenne, il se retrouvera en retard pendant les périodes de charges maximales. Puisque la messagerie BizTalk fonctionne en tant que service Windows, une application BizTalk peut compter sur le service Windows pour gérer les messages entrants.

MSMQ aborde le problème de la charge des messages à l'aide de déclencheurs qui lancent un service de traitement de la messagerie chaque fois qu'un message arrive dans la file d'attente. Ce traitement fonctionne efficacement lorsque les messages arrivent peu fréquemment, mais en cas de charge élevée, la surcharge générée par le démarrage de milliers de copies du service peut être supérieure à la logique du service.

Service Broker offre une fonctionnalité appelée *activation* qui vise à résoudre ce problème. Lorsqu'un message arrive dans une file d'attente vide, Service Broker lance une procédure stockée pour gérer le message. Cette procédure va attendre dans une boucle l'arrivée d'autres messages et va y rester jusqu'à ce que la file d'attente soit vide. Si Service Broker détermine que la procédure stockée ne supporte pas le flux des messages entrants, il démarre d'autres copies de la procédure stockée jusqu'à ce que le nombre de copies soit suffisant pour maintenir la cadence. Lorsque la vitesse d'entrée des messages baisse, la file d'attente se vide et les copies supplémentaires quittent la procédure. Ainsi, le nombre de ressources disponibles est toujours plus ou moins adéquat pour gérer les messages entrants. En tant qu'initiateur de ces procédures, Service Broker est notifié lorsque l'une d'entre elles échoue et remplace la copie défaillante. Si le service est une application externe plutôt qu'une procédure stockée, Service Broker fournit des événements auxquels une application externe peut souscrire. Ces événements vont notifier le service qu'une plus grande quantité de ressources est nécessaire pour traiter les messages dans la file d'attente.

Messages perdus

L'autre problème de fiabilité réside dans la défaillance du service au cours du traitement d'un message. Si un service supprime un message de la file d'attente dès qu'il le reçoit, puis qu'il subit une défaillance avant de l'avoir traité entièrement, ce message est perdu. De la même manière, si le service attend d'avoir traité entièrement le message avant de le supprimer de la file d'attente, une défaillance entre l'étape de traitement et la suppression du message implique que le message est toujours dans la file d'attente lorsque le service redémarre. Il sera donc traité une nouvelle fois.

Comme mentionné plus haut, traiter le même message plusieurs fois n'est pas un problème s'il s'agit d'une demande de solde, mais traiter deux fois un retrait peut irriter le client. La seule façon de s'assurer que chaque message est traité « exactement une fois » est d'englober le traitement du message et sa suppression dans la file d'attente dans une transaction. En cas de défaillance dans le traitement, les modifications du traitement et la suppression du message sont annulées de sorte que l'état est identique à celui précédant la réception du message.

Les actions de réception et de traitement du message sont donc englobées dans un même commit. Naturellement, si le traitement du message génère un message sortant, l'« envoi » de ce message sortant devra faire partie de la transaction. Ce type de traitement des messages est appelé *messagerie transactionnelle*. La plupart des infrastructures de messagerie fiables prennent en charge la messagerie transactionnelle.

Dans le cas de MSMQ, les messages sont stockés dans un fichier autre que la base de données. La messagerie transactionnelle de MSMQ impose donc un commit à deux phases pour que les deux parties de la transaction soient effectivement committées. Les commandes SEND et RECEIVE de Service Broker sont des commandes de type TSQL ; les opérations de mise à jour de la messagerie et des données dans une application Service Broker peuvent donc être exécutées à partir de la même connexion SQL Server et faire partie de la même transaction SQL Server. Un commit à deux phases n'est pas nécessaire, ce qui rend la mise en œuvre par Service Broker de la messagerie transactionnelle beaucoup plus performante que pour MSMQ.

La messagerie transactionnelle est encore une fois requise pour qu'un service non idempotent se comporte comme un service idempotent afin de supprimer les problèmes provoqués par les nouvelles tentatives d'envoi de messages. Dans le cas d'un service idempotent par nature, la messagerie transactionnelle n'est pas nécessaire. Si celle-ci n'est pas utilisée, le demandeur doit s'apprêter à recevoir plusieurs fois la réponse au message, parfois sur une longue période.

Fiabilité des données

Examinons à présent l'impact des données sur la fiabilité du service. La plupart des services accédant à des données pendant le traitement des messages, la fiabilité des données est étroitement liée à celle du service.

L'une des particularités de l'exécution d'un service asynchrone est que, très souvent, les messages du service sont des objets métier utiles. Dans notre exemple du distributeur automatique de billets, si les messages de modification du solde sont perdus pour cause de défaillance, le solde du compte client n'est pas modifié et la banque perd de l'argent. Il est donc logique de stocker les messages dans la base de données pour qu'ils bénéficient des mêmes protections de fiabilité, redondance et disponibilité que les autres données. Si les messages de modification du solde sont perdus mais qu'ils résident dans la même base de données que les comptes, ils ne seront perdus que si ces comptes sont eux-mêmes perdus. Les fonctionnalités de sauvegarde, de sauvegarde de journaux et de stockage réseau utilisées pour garantir que les comptes bancaires ne sont pas perdus s'appliquent également aux messages de modification du solde, d'où un niveau de fiabilité du service extrêmement élevé. Si les exigences de fiabilité de votre messagerie sont importantes, Service Broker ou BizTalk proposent des avantages évidents dans ce domaine en stockant les messages dans une base de données.

L'une des nouvelles fonctionnalités de SQL Server 2005 capable d'améliorer la fiabilité du service est la mise en miroir de bases de données (DBM). Elle assure la fiabilité du service en créant une copie transactionnellement consistante de la base de données dont les transactions sont cohérentes avec celles de la base de données primaire. En effet, chaque transaction committée sur la base de données primaire est réappliquée sur la base de données secondaire avant que le contrôle ne soit rendu au service. Si la base de données primaire tombe en panne, la copie secondaire peut prendre le relais en quelques secondes.

Service Broker tire parti de la mise en miroir de bases de données pour améliorer la fiabilité de la messagerie. Si la base de données du compte est une paire de bases de données mises en miroir, la base de données Service Broker du distributeur automatique de billets ouvre des connexions réseau vers les bases de données primaire et secondaire et elle envoie les messages vers la base de données primaire. Si la base de données secondaire devient la base de données primaire, Service Broker en est immédiatement averti et les messages sont dirigés vers la nouvelle base de données primaire sans intervention utilisateur, ni interruption de service.

Les services comportant un niveau élevé de données peuvent tirer profit d'une autre fonctionnalité de SQL Server 2005 pour améliorer la fiabilité. Grâce à l'intégration de Common Language Runtime (CLR) dans le moteur de SQL Server, la logique du service peut également être exécutée sur la base de données. Ainsi, pour une application Service Broker, la logique, les messages, l'environnement d'exécution, le contexte de sécurité et les données pour un service peuvent se trouver dans la même base de données.

Ce stockage en un endroit unique comporte de nombreux atouts pour les systèmes exigeant une fiabilité élevée. Les serveurs de bases de données comportent généralement les fonctionnalités matérielles et logicielles nécessaires pour maintenir la forte fiabilité qu'exige une base de données. Cette fiabilité peut désormais s'appliquer à tous les aspects de la mise en œuvre du service. Dans l'improbable hypothèse d'une défaillance du service, tout l'environnement peut être restauré à un état transactionnel cohérent grâce aux fonctionnalités de récupération de la base de données. Non seulement celle-ci est sauvegardée, mais toutes les opérations en cours sont supprimées et redémarrées dans le même environnement d'exécution et de sécurité dans lequel elles se trouvaient au moment de la défaillance.

Les applications orientées service, qui sont par nature faiblement couplées et asynchrones, présentent des exigences de fiabilité particulières. Il convient donc que le niveau de fiabilité nécessaire au service soit analysé et pris en compte au cours de la conception de l'architecture. Microsoft offre diverses infrastructures d'implémentation de services avec des niveaux de fiabilité différents. Le choix de l'infrastructure adaptée consiste donc à mettre en adéquation les exigences de fiabilité avec les capacités de l'infrastructure. Les nouvelles fonctionnalités de SQL Server 2005 apportent une infrastructure d'hébergement de service avec des niveaux de fiabilité sans précédent pour des services extrêmement exigeants dans ce domaine. •

Ressources

Architecture orientée services
http://msdn.microsoft.com/architecture/soa/

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

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

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

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

À propos de l'auteur

Roger Wolter est architecte de solutions au sein de la Microsoft Architecture Strategy Team (Cellule stratégique en matière d'architecture). Au cours de ses 30 années d'expérience dans divers secteurs informatiques, il a assumé des fonctions chez Unisys, Infospan, Fourth Shift et a passé ces sept dernières années chez Microsoft en tant que Directeur de programme. Parmi les projets qu'il a mené à bien chez Microsoft, citons SQLXML, SOAP Toolkit, SQL Server Service Broker et SQL Server Express. Son intérêt pour Service Broker est né d'un système de fabrication basé sur messagerie sur lequel il a précédemment travaillé. Roger Wolter est également l'auteur de *The Rational Guide to SQL Server 2005 Service Broker Beta Preview* (Rational Press, 2005).

Un modèle flexible d'intégration des données

par Tim Ewald et Kimberly Wolk

Résumé

L'intégration de systèmes présente de nombreux défis pour les architectes et les développeurs : pour y répondre, l'industrie s'est focalisée sur XML, les services Web et SOA, en portant un intérêt particulier aux protocoles de communication, notamment pour l'ajout de fonctions avancées de prise en charge du flux de messages dans des topologies de réseau complexes. Toutefois, ce focus sur les protocoles de communication a détourné l'attention du problème de l'intégration des données. Des modèles suffisamment flexibles pour combiner les données de systèmes différents sont indispensables pour assurer le succès d'une intégration. Ils sont exprimés avec un schéma XML (XSD) dans les systèmes basés sur les services Web et le contenu XML des messages SOAP représente les instances de ces modèles. Dans notre travail sur l'architecture du système MTPS (MSDN TechNet Publishing System), nous avons traité trois problématiques. Nous verrons en quoi elles consistent ainsi que les solutions que nous leur avons apportées dans le contexte d'un problème d'ordre plus général, celui de l'intégration des informations « client ».

L'intégration de systèmes présente de nombreux défis pour les architectes et les développeurs. Ces dernières années, l'industrie s'est attachée à utiliser XML, services Web et architecture orientée services (SOA) pour résoudre les problèmes d'intégration. La plupart du travail accompli dans ce cadre a porté sur les protocoles de communication, notamment l'ajout de fonctions avancées conçues pour prendre en charge les flux de messages dans des topologies de réseau complexes. Bien que l'intérêt de cette approche soit indiscutable, tout ce travail sur les protocoles de communication a détourné l'attention du problème de l'intégration des données.

La réussite d'une intégration impose la mise à disposition de modèles suffisamment flexibles pour combiner les données de différents systèmes. Dans les systèmes basés sur les services Web, ces modèles sont exprimés dans le format XSD. Des exemples en sont représentés en tant que XML transmis entre les systèmes dans des messages SOAP. Certains systèmes mappent les données XML dans des bases de données relationnelles, d'autres pas. Du point de vue de l'intégration, la structure des modèles de ces bases de données relationnelles n'est pas importante. Seule compte la forme du modèle des données XML défini avec le format XSD.

Les projets d'intégration des données basés sur les services Web rencontrent généralement trois types d'écueils. Tous trois sont liés à la façon dont les schémas XML sont définis. Nous nous sommes confrontés à chacun d'eux dans notre travail sur le système MTPS (MSDN TechNet Publishing System), le système XML nouvelle génération qui constitue la base de MSDN2. Nous étudierons les solutions que nous proposons dans le contexte de l'intégration des informations sur les clients.

Principal problème lié à l'intégration des données

Supposons que vous travailliez dans une grande entreprise. Celle-ci est dotée de nombreux systèmes accessibles à l'extérieur de l'entreprise que les clients utilisent pour un grand nombre de tâches. Par exemple, un système propose des informations personnalisées sur les produits aux utilisateurs inscrits qui en ont exprimé le souhait. Un autre système fournit des outils de gestion des adhésions pour les clients d'un programme partenaire. Un troisième système effectue le suivi des clients qui se sont inscrits à de futurs événements. Malheureusement, les systèmes ont tous été développés séparément par des sociétés différentes que l'entreprise a rachetées il y a un an. Chacun d'eux stocke des informations relatives aux clients dans différents formats et emplacements.

Cette situation constitue un problème majeur pour l'entreprise : elle ne permet pas d'avoir une vue d'ensemble du client. Cela produit deux effets. D'abord, le client en pâtit, car l'entreprise avec laquelle il collabore le consi-

« LA RÉUSSITE D'UNE INTÉGRATION IMPOSE LA MISE À DISPOSITION DE MODÈLES SUFFISAMMENT FLEXIBLES POUR COMBINER LES DONNÉES DE DIFFÉRENTS SYSTÈMES. »

dère comme plusieurs entités dès lors qu'il utilise différents systèmes. Par exemple, un client ayant exprimé par e-mail le désir de recevoir des informations sur un produit dès sa disponibilité doit de nouveau exprimer son souhait de se voir informé lorsqu'il s'inscrit à des événements organisés par l'entreprise. Son expérience utilisateur serait facilitée si le système à l'aide duquel il s'est inscrit à un futur événement avait déjà connaissance de son intérêt pour un produit particulier.

En outre, l'entreprise en souffre, car elle n'a pas une vision unifiée de ses clients. Combien de clients membres d'un programme partenaire reçoivent également des informations sur les produits via e-mail ? Dans les deux cas, les séparations entre les systèmes que le client utilise limitent la capacité de l'entreprise à répondre correctement à ses besoins.

Que cette situation survienne parce que les systèmes ont été conçus et développés individuellement sans considération pour le contexte plus large dans lequel ils sont utilisés ou parce que différents groupes de systèmes intégrés ont été regroupés à l'occasion de fusions et d'acquisitions se révèle sans importance. Le problème demeure : l'entreprise doit intégrer les systèmes pour améliorer l'expérience du client, la connaissance qu'elle a de ce dernier et la façon de le servir.

L'approche la plus commune pour résoudre ce problème est de veiller à ce que tous les systèmes adoptent un seul modèle canonique par client. Un groupe d'architectes se constitue et travaille à la conception d'un format XML unique visant à représenter les données relatives aux clients. Celui-ci est défini à l'aide d'un schéma écrit en XSD. Pour permettre aux systèmes de partager les données dans le nouveau format, une équipe centrale construit un nouveau magasin qui le prend en charge. L'équipe

en charge du modèle de données XSD et celle en charge du magasin proposent leur solution à l'ensemble des équipes responsables des systèmes concernant, d'une manière ou d'une autre, les clients et requérant son adoption. Les principales modifications sont indiquées dans les figures 1 et 2.

Chaque système est modifié pour utiliser le magasin sous-jacent des données clients via son interface de services Web. Ils récupèrent et stockent les informations XML sur les clients en se conformant au schéma du client. Tous les systèmes partagent la même instance de service, le même modèle de données XSD et les mêmes informations XML.

Cette solution semble simple, élégante et adaptée mais de manière générale, une mise en œuvre naïve échouera pour l'une de trois raisons suivantes : demande de trop d'informations, absence de stratégie de gestion de versions efficace et absence de support pour une extension des schémas au niveau du système.

Demande d'un trop grand nombre d'informations

Un schéma et un magasin requérant trop d'informations constituent la première cause potentielle d'échec. Les personnes concevant un simple service Web pour une intégration point-à-point ont tendance à penser aux données dont leur service a besoin. Elles établissent un contrat dont les termes requièrent que ces données spécifiques soient fournies. Elles le sont de manière implicite lorsqu'un contrat est généré à partir du code source. La plupart des outils qui effectuent un mappage du code source au contrat de service Web traitent les champs de valeurs simples comme les éléments de données requis, imposant leur envoi par le client. Même lorsqu'un contrat est créé manuellement, la tendance qui consiste à traiter toutes les données comme requises demeure. Dès qu'il détermine (par code ou validation du schéma) que certaines des données requises sont manquantes, le service rejette une requête. Le client, lui, reçoit une erreur de service.

Cette approche pour définir les contrats de service Web est trop rigide et conduit à des systèmes fortement couplés. Toute modification des conditions d'utilisation du service entraîne une modification du contrat et des clients l'utilisant. Pour rendre ce couplage plus flexible, il convient de séparer la définition du format de données requis par un service de ses conditions d'exécution. De manière plus concrète, les formats de données définis par votre contrat doivent traiter toute donnée autre que des données d'identité comme facultative. L'implémentation de votre service doit s'assurer du respect des contraintes d'utilisation (à l'aide d'un code ou d'un schéma de validation dédié). Il doit être aussi indulgent que possible lorsque des données sont manquantes dans la requête d'un client et proposer une réponse adaptée.

Dans l'exemple relatif aux informations « client », il est facile de penser à des cas dans lesquels certains systèmes, qui voudraient travailler avec des données clients, ne disposeraient pas de toutes les informations nécessaires. Le système enregistrant la demande d'information d'un client pour un produit particulier peut, par exemple, ne mémoriser que le nom et l'adresse e-mail préférée de ce dernier. Le système d'inscription aux événements, en revanche, peut enregistrer également les informations relatives à l'adresse et à la carte de crédit. Si un modèle commun de données clients requiert que chaque enregistrement correct comprenne les informations sur le nom, l'e-mail, l'adresse et la carte de crédit, aucun système ne peut l'adopter sans recueillir plus de données que nécessaire ou fournir des données erronées. Rendre facultatives toutes les données autres que celles relatives à l'identité (numéro d'identification, adresse e-mail, etc.) facilite l'adoption du modèle de données, car il suffit aux systèmes de fournir les informations en leur possession.

La séparation du format de données et des conditions d'exécution permet de gérer plus aisément une modification dans l'implémentation d'un service. Elle est également critique lors de la définition d'un schéma XML commun qui doit être utilisé par plusieurs services et clients. Si trop d'informations sont obligatoires, certaines d'entre elles peuvent manquer aux systèmes désireux d'utiliser le modèle de données. Cela laisse donc le choix à ces derniers de ne pas adopter le modèle partagé et de stocker

Figure 1 Trois magasins de données séparés, un par système

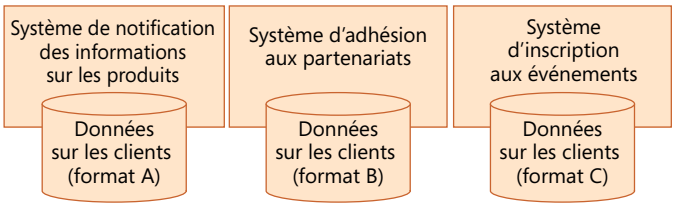
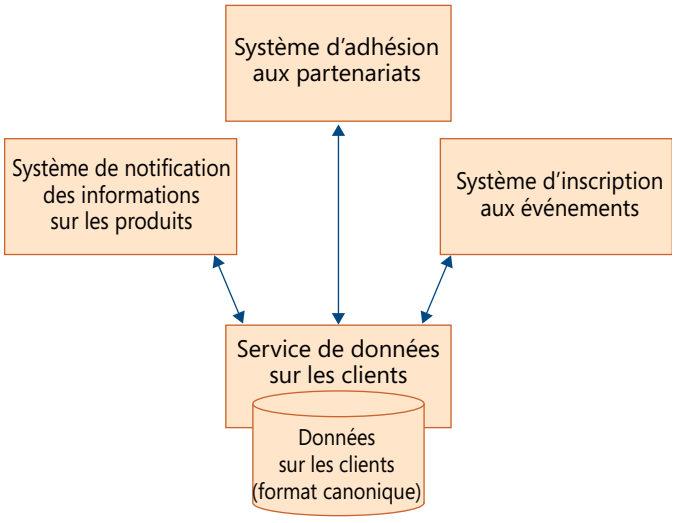


Figure 2 Un magasin de données et un format uniques



ou de fournir des données erronées (souvent la valeur par défaut d'un simple type de langage de programmation). Chacune de ces options peut être considérée comme une défaillance.

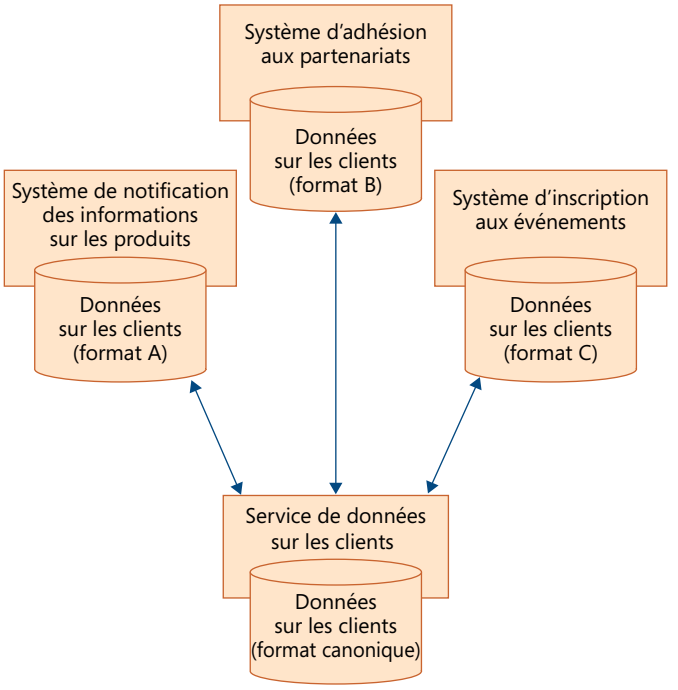
La suppression des contraintes sur le schéma offre un gain de flexibilité sur l'adoption du modèle par les systèmes. Chaque système peut contribuer à hauteur des données dont il dispose, ce qui facilite l'adoption d'un schéma XML commun. La seule obligation pour les systèmes recevant des données est de vérifier que celles dont ils ont réellement besoin sont disponibles. Si tel n'est pas le cas, ils doivent répondre de manière à obtenir plus de données de la part de l'utilisateur ou d'un autre magasin, en adaptant leur comportement ou, au pire des cas, en générant une erreur. Il s'agit, en fait, de déplacer certaines des contraintes que l'on placerait normalement dans un schéma XML dans son code, où elles seront vérifiées lors du fonctionnement. Ce déplacement permet de les modifier sans réviser le schéma partagé.

Absence de stratégie de gestion de versions efficace

L'absence de stratégie de gestion de versions constitue la seconde cause potentielle d'échec. Peu importe le temps et les efforts consacrés, initialement, à la définition d'un schéma XML, ce dernier devra être modifié avec le temps. Si les schémas, le magasin partagé qui les prend en charge et tous les systèmes qui les utilisent doivent passer simultanément à une nouvelle version tous à la fois, l'échec est assuré. Certains systèmes devront attendre les modifications nécessaires, car d'autres n'en sont pas au point de pouvoir adopter une révision. À l'inverse, certains systèmes seront forcés d'effectuer des tâches supplémentaires non prévues parce que d'autres doivent adopter une nouvelle révision. Cette approche est inapplicable.

Pour résoudre ce problème, il faut adopter une stratégie de gestion de versions qui permette au schéma et au magasin d'évoluer indépendamment du rythme auquel les autres systèmes adoptent leurs révisions. Cette solution apparaît simple et elle l'est, pour peu que l'on considère les schémas XML de la bonne façon.

Figure 3 Une combinaison de magasins (voir figures 1 et 2)



Les systèmes qui effectuent une intégration à l'aide d'un schéma XML commun le considèrent comme un contrat. Baisser la barre en matière de données requises en rendant certains éléments facultatifs facilite l'acceptation d'un contrat, car les systèmes ont moins d'obligations. En ce qui concerne la gestion des versions, les systèmes doivent aussi être autorisés à faire plus *sans modifier l'espace de nom du schéma*. Cela signifie, dans la pratique, qu'un système doit toujours produire des données XML en fonction de la version de schéma à partir de laquelle il a été développé. Il doit toujours utiliser des données basées sur cette même version *avec des informations supplémentaires*. Cette définition est une variante de la loi Postel : « Soyez libéral dans ce que vous acceptez, conservateur dans ce que vous envoyez ». On peut dire que cette idée est à la base de toutes les technologies performantes de systèmes distribués et, sans l'ombre d'un doute, de toutes celles des systèmes faiblement couplés. En adoptant cette approche, on peut étendre un schéma sans mettre à jour les clients.

Dans l'exemple relatif aux clients, une mise à jour du schéma et du magasin est susceptible de permettre la prise en charge d'un élément facultatif supplémentaire qui conserve le nom de jeune fille de la mère de l'utilisateur à des fins de sécurité. Si les systèmes fonctionnant avec l'ancienne version génèrent des enregistrements « client » sans cette information, cela n'a pas de conséquence, car l'élément est facultatif. S'ils envoient ces enregistrements à d'autres systèmes qui requièrent cette information, la requête peut ne pas aboutir et cela est également sans conséquence. Si les nouveaux systèmes envoient les données sur le client, avec le nom de jeune fille de la mère, aux anciens systèmes, cela ne pose, là non plus, pas de problèmes, car ils sont conçus pour l'ignorer.

Heureusement, de nombreux kits d'outils destinés aux services Web prennent en charge cette fonctionnalité directement dans leur conversion de paramètres basée sur les schémas. Il ne fait aucun doute que les mappeurs objet-XML .NET (la classe XmlSerializer, qui a fait ses preuves, et le nouveau XmlFormatter/DataContract) gèrent parfaitement les données supplémentaires. Certains kits d'outils Java le font et les frameworks qui prennent en charge les nouvelles spécifications du JAX-WS 2.0 et JAXB 2.0 également. Ceci étant, l'adoption de cette approche est relativement aisée.

Le seul problème réel avec ce modèle est qu'il introduit plusieurs définitions d'un schéma donné, chacune représentant une version différente. Lorsque, par exemple, l'on considère un fragment XML extrait d'un message envoyé sur la ligne, il est impossible de répondre à la question : « Ces données sont-elles valides ? », celle-ci n'ayant de réponse qu'en fonction d'une version particulière du schéma. L'incapacité de statuer de manière définitive si certaines données sont correctes constitue un problème en matière de débogage, voire éventuellement de sécurité. Avec des modèles de données décrits à l'aide d'un schéma XML, il est possible de répondre à une autre question plus intéressante : « Ces données sont-elles suffisamment valides ? »

Cette question constitue la préoccupation essentielle des systèmes et on peut y répondre à l'aide des résultats que la plupart des mécanismes de validation de schémas XML fournissent, mesure raisonnable à prendre dans le cas où cette validation est requise et pouvant être mise en œuvre avec les frameworks de validation disponibles aujourd'hui.

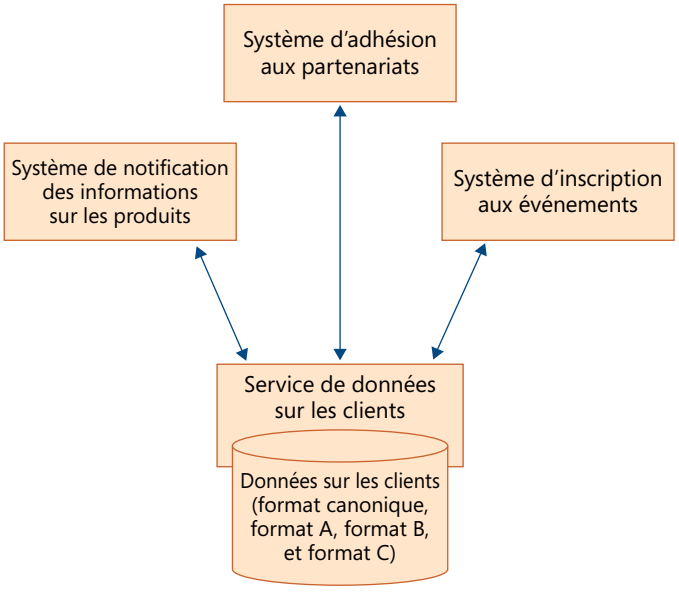
Absence de support pour une extension des schémas au niveau du système

L'absence de support pour une extension des schémas au niveau du système constitue la troisième cause potentielle d'échec. La stratégie de gestion de versions basée sur l'idée qu'une définition de schéma change avec le temps est nécessaire pour promouvoir l'adoption, mais pas suffisante. Alors qu'elle dispense les systèmes d'avoir à adopter immédiatement la dernière révision de schéma, elle n'aide en rien ceux qui attendent des mises à jour de schéma spécifiques. Des retards dans les révisions peuvent également rendre un schéma trop coûteux pour être adopté par un système. La solution à ce dernier problème consiste à permettre aux systèmes qui adoptent un schéma commun de l'étendre à l'aide de leurs propres informations supplémentaires. Les informations liées à ces extensions de schéma peuvent être stockées localement dans un magasin au niveau du système (voir figure 3).

Dans ce cas, chaque système est modifié pour écrire les données utilisateurs à la fois dans son magasin dédié à l'aide de son propre modèle de données et dans le magasin partagé à l'aide du schéma canonique. Il est également modifié pour lire les données clients à la fois à partir de son magasin dédié et depuis le magasin partagé. En fonction de ce qu'il trouve, il sait si un client est déjà connu de l'entreprise et du système. Le tableau 1 résume les trois possibilités.

Le système peut utiliser ces informations pour décider des données qu'il a besoin de recueillir sur un client. S'il s'agit d'un nouveau client de l'entreprise, le système ajoute le plus d'informations possible au magasin

Figure 4 Le même magasin (voir figure 3) avec les formats de données d'un magasin partagé



canonique. Celles-ci sont mises à la disposition des autres systèmes qui manipulent les données clients. Le système peut également stocker des données dans son magasin dédié afin de répondre à ses propres besoins. Ce modèle peut être ultérieurement étendu de manière à ce que les données spécifiques au système soient également stockées dans le magasin partagé (voir figure 4).

Cette solution permet à un système d'en intégrer un autre en manipulant des données d'extension absentes du schéma canonique. Pour fonctionner correctement, le magasin et les autres systèmes doivent avoir une visibilité sur ces données d'extension. En d'autres termes, elles ne peuvent pas être opaques. La façon la plus aisée de résoudre ce problème est de transformer les données d'extension en XML. Le système fournissant les données définit un schéma pour les données d'extension, de manière à ce que d'autres systèmes puissent le traiter de manière fiable. Le magasin partagé conserve la trace des schémas d'extension afin de pouvoir garantir que les données d'extension sont valides, même s'il ne sait pas de façon explicite ce que l'extension contient. Dans les cas les plus extrêmes, un système peut choisir de stocker toutes les données d'un client dans un format spécifique de données d'extension XML. D'autres systèmes qui comprennent ce format peuvent l'utiliser. Ceux qui ne le comprennent pas dépendent, au contraire, de la représentation canonique.

Lorsque les systèmes sont indépendants, chacun d'eux contrôle sa propre destinée. Ils peuvent recueillir et stocker toutes les informations dont ils ont besoin dans le format et l'emplacement de leur choix. Le passage à un schéma et un magasin communs change cette situation. Si l'adoption d'un format de données XML commun restreint la liberté d'un système, de telle sorte qu'il ne puisse plus offrir la fonctionnalité requise, l'échec est inévitable.

« L'INCAPACITÉ DE STATUER DE MANIÈRE DÉFINITIVE SI CERTAINES DONNÉES SONT CORRECTES CONSTITUE UN PROBLÈME EN MATIÈRE DE DÉBOGAGE, VOIRE ÉVENTUELLEMENT DE SÉCURITÉ. »

L'utilisation d'une combinaison de données d'extension XML à un niveau du système ou dans le magasin partagé ajoute en complexité, car les données doivent demeurer synchronisées. Mais elle fournit également une immense flexibilité. Il est possible d'aligner les systèmes sur toute combinaison du schéma canonique et de schémas alternatifs. On peut s'appuyer sur un format XML pendant longtemps, mais il est toujours possible de s'en détourner pour répondre à de nouvelles exigences. Cette liberté supplémentaire n'est pas des moindres dans le monde incertain de l'entreprise.

Un autre subtil avantage de ce modèle est qu'il permet à l'équipe définissant le schéma commun de ralentir son travail sur les révisions de ce schéma. Les systèmes peuvent utiliser des données d'extension pour répondre aux nouvelles exigences entre les révisions. L'équipe travaillant sur le modèle canonique peut extraire ces extensions pour les intégrer dans leur processus de révision. Cette approche permet de garantir que les modifications apportées au modèle sont vraiment justifiées par les exigences du système.

Ressources

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

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

Tableau 1 Les trois cas possibles de données sur les clients

Enregistrement dans le magasin partagé	Enregistrement dans le magasin système	Signification
Non	Non	Il s'agit d'un nouveau client de l'entreprise. Recueillir toutes les données communes et spécifiques au système..
Oui	Non	Il s'agit d'un client connu de l'entreprise mais nouveau dans le système. Recueillir les données spécifiques au système.
Oui	Oui	Il s'agit d'un client connu de l'entreprise et du système.

Réduction des risques

De nombreuses entreprises s'attachent à intégrer leurs systèmes en utilisant les données XML décrites à l'aide de schémas XML et échangées via services Web. Dans cet article, nous avons présenté trois causes communes d'échec de ces projets d'intégration centrés sur les données : la demande d'un trop grand nombre d'informations, l'absence de stratégie de gestion de versions efficace et l'absence de support pour des extensions schéma au niveau du système. Pour réduire ces risques :

- Rendez les éléments de schéma facultatifs et codez les conditions spécifiques propres au système dans l'implémentation de ces systèmes.
- Créez des systèmes qui produisent les données en fonction de leur version d'un schéma partagé, mais les utilisent de façon à pouvoir adopter des révisions de schéma à différents rythmes sans modifier l'espace de nom du schéma.
- Autorisez les systèmes à étendre les schémas partagés avec leurs propres données, pour répondre aux nouvelles exigences indépendamment des révisions de modèles de données.

L'ensemble de ces solutions repose sur une idée centrale : pour s'intégrer avec succès sans sacrifier leur flexibilité, les systèmes ont besoin de s'accorder sur aussi peu que possible tout en offrant le service souhaité. Tout cela est-il possible ? La réponse est oui. Ces techniques sont au cœur de la conception du système MTPS (MSDN/TechNet Publishing System), le moteur de publication de MSDN2. •

À propos des auteurs

Tim Ewald est architecte principal chez Foliage Software Systems où il aide les clients à concevoir des applications allant de l'informatique d'entreprise aux équipements médicaux. Avant de rejoindre Foliage, il a travaillé chez Mindreef, un des principaux fournisseurs d'outils de diagnostic de services Web. Avant cela, il était directeur de programme senior chez MSDN, où il a collaboré avec Kim Wolk sur le MTPS : le moteur de publication basé sur XML et les services Web de MSDN2. Tim est un conférencier et un auteur reconnu internationalement.

Kim Wolk est la responsable développement du MSDN et la force motrice du MTPS, le moteur de publication basé sur XML et les services Web de MSDN2. Elle a, au préalable, travaillé en tant que développeur et co-architecte MTPS. Avant de rejoindre MSDN, elle a été consultante pendant plusieurs années, en tant qu'indépendante et au sein de Microsoft Consulting Services, où elle a travaillé sur de nombreuses applications critiques d'entreprise.

Modèles de composition et de consommation des données à haute intégrité

par Dion Hinchcliffe

Résumé

Le défi consiste à consommer et à gérer des données de plusieurs sources sous-jacentes dans différents formats tout en participant à un écosystème d'informations fédéré et en conservant une intégrité et un couplage faible avec de bonnes performances. Voici quelques modèles émergents pour le monde en pleine expansion des « mashups » et des applications composites.

Aujourd'hui, le développement logiciel consiste autant à assembler et à composer des données et des services préexistants qu'à créer une toute nouvelle fonctionnalité. Les « mashups » sur le Web et les applications composites dans le domaine de l'architecture orientée services (SOA) requièrent le regroupement, l'interaction, puis la séparation de données aux formats très différents et de sources variées. Et ces approches nécessitent un bon fonctionnement de cette interaction, sans que la fidélité ou l'intégrité des données ne soit affectée.

La variété considérable de données existant à l'heure actuelle couvre toute une gamme de formats XML ainsi que des formats de données plus légers et de plus en plus répandus, tels que JSON (JavaScript Object Notation) et les microformats. Les applications aussi doivent de plus en plus souvent fonctionner avec des formats de données riches, tels que les images, l'audio et la vidéo, sans oublier les formats plus anciens utilisés au quotidien, tels que le texte, EDI ou les objets natifs. Tous ces formats sont de plus en plus mélangés les uns aux autres alors que les systèmes deviennent plus interconnectés et intégrés dans de grandes chaînes d'approvisionnement, dans des bus de services d'entreprise (ESB), dans des SOA et dans des « mashups » Web. Maintenir l'ordre dans ce chaos de données est plus important que jamais. Bonne nouvelle : des règles de premier ordre visant à gérer l'hétérogénéité de toutes ces données commencent à voir le jour.

Les services Web, la SOA et Internet, en particulier, sont constitués de normes ouvertes qui permettent aux systèmes de communiquer, la norme omniprésente et essentielle HTTP en étant un exemple édifiant. Toutefois, cette communication ne permet pas de connaître le type de données qu'utilisera votre ordinateur dans les applications de demain. Bien que l'on puisse tabler sur certains formats XML, il est désormais tout aussi probable que vous soyez confrontés à des formats de données simples à la popularité grandissante, tels que JSON. Toutefois, de plus en plus, il pourrait également s'agir d'un simple texte brut, d'une arborescence d'objets natifs, voire d'une pile multicouche de services Web de style WS-* fournie par la prochaine Windows Communication Foundation (WCF).

Les développeurs font donc face à de sérieux défis pour créer des techniques performantes d'intégration, d'architecture et de modélisation des données capables de répondre à cette diversité. L'hétérogénéité des formats de représentation des données peut être assez décourageante dans des environnements requérant un haut niveau d'intégration de systèmes. Peu importe que dans des systèmes très faiblement couplés et hautement fédérés, comme le deviennent de plus en plus d'applications, la probabilité de changements fréquents soit élevée. Il appartient à la communauté en charge du développement des applications de constituer un ensemble de connaissances sur les meilleures pratiques en matière de simple consommation de données de différents formats et sources, avec les techniques d'intégration, de fusion et de liaison correspondantes. Cette communauté doit également s'assurer que l'intégrité des données est conservée tout en veillant à ce qu'elles conservent leur résistance aux changements et ce, malgré l'inévitable évolution des formats de données sous-jacents.

Les modèles présentés ici, bien que ne faisant pas autorité et ayant été conçus à l'origine à titre d'aide, se basent sur le concept largement reconnu des contrats d'interface. Ces contrats d'interface sont désormais répandus dans le monde des services Web avec le WSDL ainsi que dans de nombreux langages de programmation mais, en particulier, lorsque la conception repose sur la définition d'un contrat. Dans un contrat d'interface, un opérateur et un fournisseur se réunissent pour décider d'un ensemble d'interactions, habituellement décrites sous les termes de méthodes ou de services. Ces interactions permettent de faire circuler des données ciblées dans les limites de l'interface.

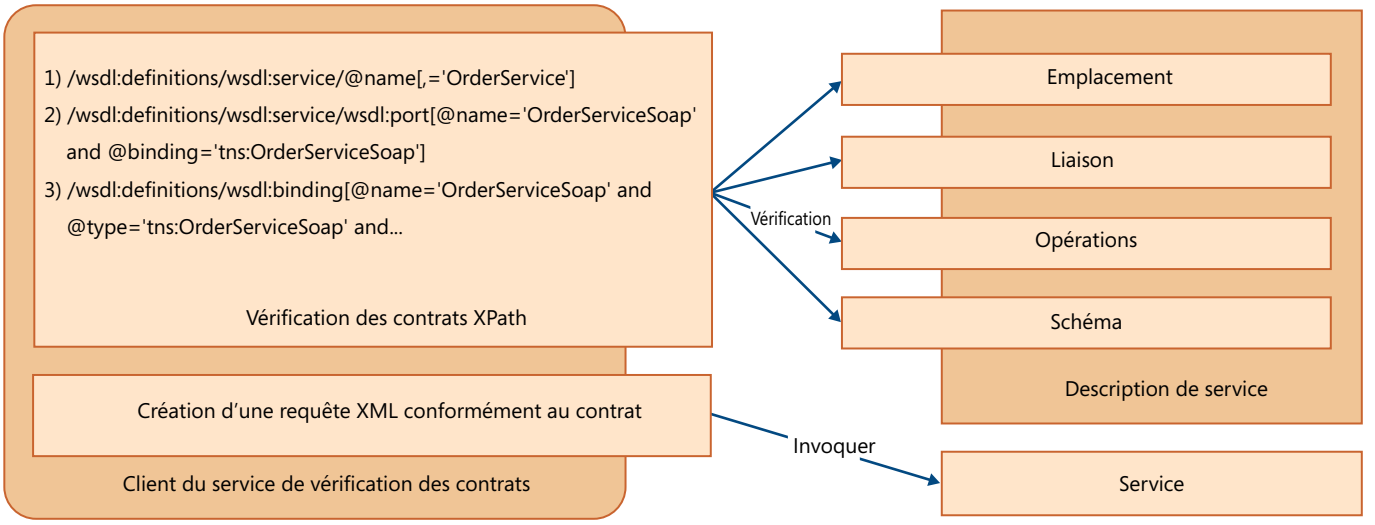
Le détail de l'interaction des données ainsi que leurs emplacements et protocoles sont définis avec précision dans le contrat qui, de préférence, est lisible par une machine. Les structures de données circulant en tant que paramètres, au cours de chaque interaction, entre le fournisseur et le client sont particulièrement intéressantes. Et ce sont ces structures de données qui sont au cœur du modèle de l'application composite et du « mashup ». Car c'est ici, au point de rencontre de ces structures de données très différentes les unes des autres, que nous devons appliquer au mieux les principes de recombinaison et d'intégration des données.

Forces et contraintes de la composition et de la consommation

En appliquant ces principes, nous pouvons obtenir une idée générale des forces qui définissent la composition et la consommation des données dans les logiciels distribués actuels. En voici le classement approximatif :

Le contrat d'interface en tant que pilote de la composition et de la consommation des données. Lorsque l'on utilise les structures de données d'une source ou d'un service externe, tel qu'un service Web, elles doivent être validées par rapport au contrat d'interface fourni par le service source. Cette validation peut souvent être effectuée au moment de la conception pour les sources de données reconnues comme hautement stables et fiables. Toutefois, dans les systèmes fédérés, notamment ceux qui ne sont pas sous le contrôle direct du consommateur, il convient de faire particulièrement attention à la vérification du contrat au moment de l'exécution. Cette vérification, dans des systèmes faiblement couplés, est une technique émergente et certaines options intéressantes sont à la disposition de l'utilisateur du service afin d'éviter, dès le départ, tout problème. Il en résulte que le contrat d'interface est le principal élément permettant d'orchestrer la composition et la consommation des données.

Figure 1 Les schémas intégrés dans un contrat d'interface constituent habituellement la plus grande dépendance du client.



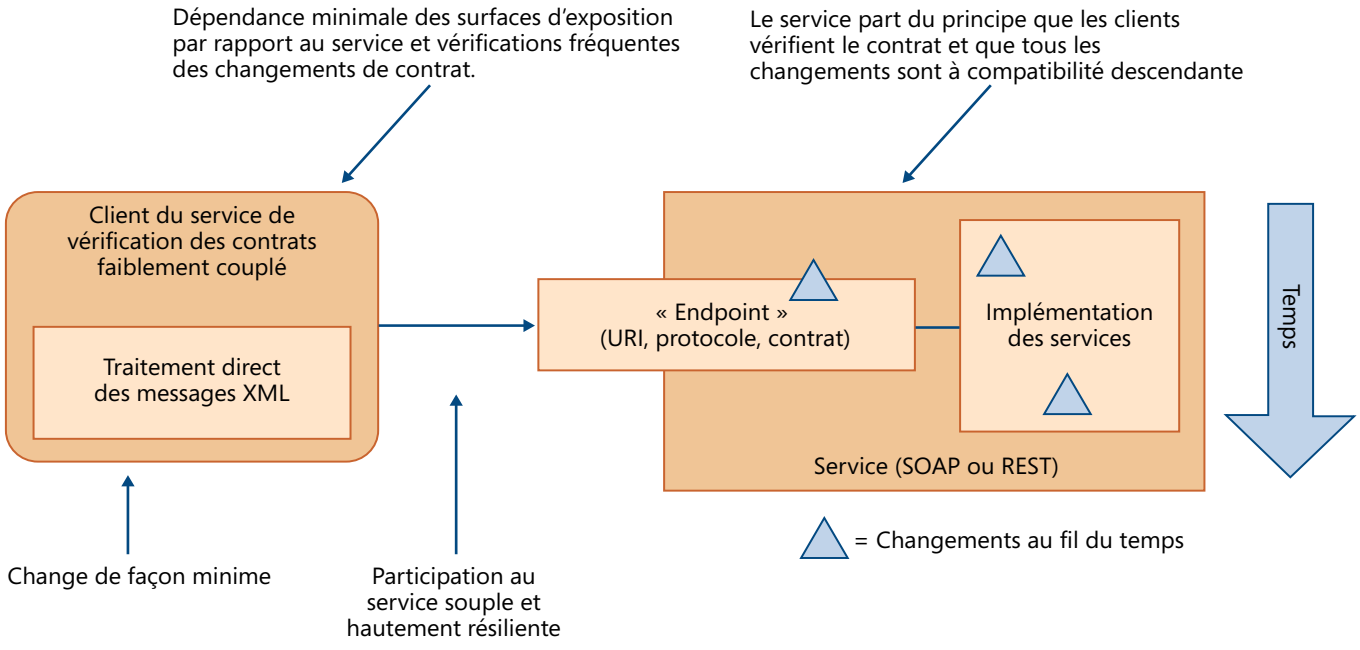
Le défaut d'adaptation des « impédances » pour la mise en place d'une abstraction des données interrompt la composition et la consommation des données. Le point physique auquel l'intégration des données a lieu est, de plus en plus, à l'extérieur du contrôle classique des bases de données et les bibliothèques d'accès de données convertissent tout en une abstraction de données unifiées. Les données récupérées des différents services externes et dans différents formats sont en conflit dans le navigateur, dans les clients « front-end » ou côté serveur, ou dans le code interne hors de la base de données. Les structures de données de ces services sous-jacents sont, selon l'application logicielle, des objets natifs, XML, JSON et documents ou fragments SOA, voire des données texte, binaires et multimédia. Bien que la problématique d'adaptation des impédances relative à l'abstraction des données (« impedance mismatch ») soit un problème connu depuis longtemps en ce qui concerne les logiciels, elle a été exacerbée par la prolifération de modèles destinés à la représentation des informations. Lors de la consommation et de la composition de ces structures de données dans des représentations agrégées utilisables, puis leur retour vers leurs services d'origine, plusieurs problèmes majeurs surviennent.

> « LORSQUE L'ON UTILISE LES STRUCTURES DE DONNÉES D'UNE SOURCE OU D'UN SERVICE EXTERNE, TEL QU'UN SERVICE WEB, ELLES DOIVENT ÊTRE VALIDÉES PAR RAPPORT AU CONTRAT D'INTERFACE FOURNI PAR LE SERVICE SOURCE. »

- *Variabilité des formats de données*. Les principaux formats des structures de données sous-jacentes sont souvent hautement incompatibles et de bons moyens pour traiter localement tous les formats potentiels ne sont habituellement pas disponibles. Les formats extrêmement complexes requérant des piles de protocoles sophistiqués pour être traités, tels que les services Web de haut niveau, mettant en œuvre des règles WS-POLICY, sont particulièrement problématiques.
- *Identité des données*. Les identificateurs d'objets/de données clés et uniques ne sont pas souvent dans des formats communs et ne font pas partie des moyens disponibles pour les vérifier et veiller à leur application homogène dans les différents modèles d'abstraction.
- *Relations des données*. Établir des relations et des associations entre les structures de données de formats très différents peut se révéler problématique, sur le plan de l'efficacité et des performances, pour un certain nombre de raisons. La conversion de toutes les données dans un format commun peut être coûteuse, tant en termes de maintenance que de coûts d'exécution. Ceci introduit également le problème du maintien de la provenance des données mélangées et de leur extraction une fois modifiées. La conservation des structures de données dans les formats d'origine suppose de bons mécanismes pour les composer, les lier et les manipuler.
- *Provenance des données*. Le maintien du service source d'origine d'un segment de données est essentiel pour conserver la validité du contexte des conversations en cours et pour les changements des données en particulier, tels que les mises à jour, les ajouts et les suppressions.
- *Intégrité des données*. Il est important de modifier les données source en veillant à ce que les changements soient valides conformément au contrat d'interface du service sous-jacent et sans enfreindre les définitions de schémas. Avec des documents XML bien définis et reposant sur des schémas XML riches, ces validations sont une tâche aisée. Toutefois, des modèles de programmation plus légers, tels que JSON, ont rarement des schémas lisibles par une machine. Ces types de formats de données, bien que souvent uniquement disponibles à partir d'un service fédéré, constituent le risque le plus élevé de problèmes d'intégrité, car leurs règles de changement et de manipulation ne sont pas aussi bien définies que pour les autres formats.
- *Conversion des données*. Souvent, il n'existe pas de conversion clairement spécifiée entre les formats de données ou, pire, les multiples choix proposés modifient les données de façons subtiles. Les données les plus fréquemment affectées sont des représentations numériques, mais ces conversions peuvent également affecter, presque tout aussi souvent une source de données dans plusieurs jeux de caractères, des données audio/vidéo et des données GIS.

Surfaces d'exposition des contrats d'interface de grande envergure. Des structures d'interface et des schémas extrêmement complexes conduisent à l'un des principaux problèmes dans le domaine de l'architecture des données de systèmes distribués. L'expérience a démontré à maintes reprises que la simplicité est garante de plus de fiabilité et d'une qualité supérieure. Pourtant, les contrats d'interface des services distribués contiennent souvent des schémas XML et des définitions WSDL complexes. La probabilité de changements, opérés ou non par inadvertance, s'ac-

Figure 2 Vérification de l'exécution du contrat d'interface des services Web



croît avec la complexité d'un contrat d'interface et de ses schémas intégrés. Alors que les approches XML et REST des services tendent à encourager la simplicité souhaitée, certains domaines d'applications ne sont parfois pas « simples » et ces modèles de services sont sujets à des problèmes supplémentaires en raison de l'absence de normes relatives au contrat d'interface. Résultat : la probabilité que les changements effectués sur le contrat sans qu'un fournisseur distant n'en soit avisé augmente suivant la taille du contrat, car les changements opérés sur les grands schémas sont plus susceptibles de ne pas être détectés que ceux effectués sur un schéma plus petit.

Changements de version implicites. Même dans les environnements bien contrôlés, les changements effectués sur les services et leurs contrats d'interface peuvent être effectués sans en avertir toutes les parties concernées. En apportant des changements au fonctionnement actuel des services de données, ce modèle n'en sera que plus répandu dans les environnements hautement fédérés d'aujourd'hui et de demain. Dans le cas contraire, les services dont vous dépendez changeront sans que vous n'en ayez connaissance ou que vous n'ayez donné votre avis au préalable et vous devrez vous y résigner. Par conséquent, le développement de stratégies conscientes visant à détecter les changements de versions et à les gérer efficacement est essentiel pour conserver la qualité des fonctionnalités et des données d'un « mashup » ou d'une application composite.

Modèles de composition et de consommation de données

Les « patterns » décrits dans cet article visent à présenter une position simple sur la modélisation des données et l'architecture des applications. Ces remarques résultent de l'observation du monde dynamique des « mashups » et des applications composites qui se multiplient sur le Web depuis déjà plusieurs années. Les approches souples et minimalistes utilisées, notamment, par les « mashups » constituent une source d'inspiration mais ne suffisent souvent pas à créer un logiciel de qualité. Ces modèles visent à épouser l'esprit des « mashups », à mettre ces derniers en contexte avec de bonnes pratiques d'ingénierie logicielle et à ouvrir un débat avec les gens de la profession sur ces méthodes minimales de connexion des services et des données.

« Pattern 1 » : dépendance minimale des surfaces d'exposition sur les contrats d'interface. Ce modèle est l'aspect visible de la maxime bien connue dans la conception logicielle : « Soyez libéral dans ce que vous acceptez, conservateur dans ce que vous envoyez ». La plupart des boîtes à outils de services Web, des frameworks de base de données relationnelles et des bibliothèques multimédia encouragent les dépendances sur trop de points du contrat d'interface (souvent sur l'ensemble du contrat), indépendamment de ce dont dépend le logiciel. Dans un très grand nombre de scénarios de composition et de consommation, une petite dépendance des surfaces d'exposition est tout à fait suffisante. Une dépendance de tout le contrat induit une grande fragilité, car les changements qui n'ont aucune relation avec les données de la structure dont elles dépendent, seront responsables d'une rupture sur le client. Alors que certaines boîtes à outils de consommation des données ne tiennent déjà plus compte de cette dépendance, il y a souvent peu de contrôle sur les parties qu'elles prennent en charge.

Pour de nombreuses applications, seules sont appropriées les dépendances directes des éléments de données internes requis. Toutes les autres dépendances doivent être activement éliminées de la relation de dépendance au sein des données sources. Le point crucial dans ce cas est que les changements effectués dans le contrat d'interface, qui ne sont pas importants pour le consommateur des données, ne doivent pas en empêcher l'utilisation. Le contraire doit également être vrai : les changements effectués dans le contrat qui sont importants doivent immédiatement devenir apparents pour empêcher un comportement et une utilisation incorrects des données. Les exemples de dépendances minimales comprennent : XML (les chemins d'accès clés des schémas aux éléments de données), les données relationnelles (uniquement les tableaux, les types, les colonnes et les index utilisés par le client) et les données multimédia (uniquement les parties de la structure de média requises, telles que les canaux spécifiques, la bande passante, la portion d'image ou le segment vidéo).

« Pattern 2 » : vérification des changements du contrat en cours d'exécution. Les services dont dépendent les « mashups » et les applications pour les données sont sujets, à tout moment, à des changements inattendus. Ces derniers peuvent être dus au déplacement du « endpoint » dans un nouvel emplacement, à l'arrêt de l'utilisation d'un protocole précédemment supporté, à des changements effectués sur le schéma sous-jacent, voire même simplement à des changements délibérés opérés en bloc sur l'interface ou à des défaillances des services internes. La raison conduit à la nécessité de vérifier le contrat afin d'en détecter les changements. Elle requiert également de pouvoir traiter correctement de nombreux changements de contrats, car ils peuvent très bien ne pas affecter la partie du contrat qui vous importe.

Il existe, en fait, deux types de vérifications de contrat en cours d'exécution. La première consiste à *vérifier la spécification du contrat*, s'il y en a une. Il s'agit souvent du WSDL ou d'autres métadonnées officiellement publiées en conjonction avec le service lui-même. Elle peut être vérifiée facilement et rapidement à l'aide d'une variété de techniques de programmation, incluant les requêtes XPath pour les schémas XML ou d'autres techniques relativement légères. Les vérifications de contrats codés en dur dans les langages de programmation traditionnels ne sont pas si faciles à implémenter ou à modifier et doivent constituer un choix secondaire.

La seconde vérification de contrat consiste à valider le contrat par rapport aux instances de données fournies par le service. Les contrats d'interface ont souvent leur propre impédance relative à l'abstraction des données avec leurs mécanismes de livraison, et la fréquence des incohérences entre les données fournies et le contrat d'interface, même avec des boîtes à outils de haute qualité, peut se révéler surprenante.

La fréquence à laquelle vérifier la validation de contrat constitue le plus grand dilemme présenté par ce modèle. La vérification du contrat et des données d'instance à l'aide de chaque donnée récupérée à partir de sa source peut prendre du temps et se révèle habituellement inutile. À terme, déterminer la fréquence des vérifications de contrat dépend largement des exigences de l'application ainsi que de la nature des données et de leur tolérance à l'inexactitude et à un comportement incorrect. Pour de nombreuses applications, une vérification synchrone peut même ne pas être une option et il peut se révéler judicieux de mettre en place un processus d'arrière-plan périodique et approprié pour identifier les changements de contrat qui se produiront inévitablement.

« Pattern 3 » : alignement des structures de données sur un format commun d'abstraction. Dans les faits, l'hétérogénéité des structures et des formats de données avec lesquels les « mashups » et applications composites doivent fonctionner ne cesse de s'accroître. Les logiciels peuvent les manipuler dans leurs formats de données d'origine, ce qui signifie perdre des opportunités de maintenir des relations entre elles et d'appliquer les règles métier ou ils peuvent tous les convertir dans une abstraction commune. Cette conversion est l'approche que les bibliothèques de données comme ADO.NET utilisent avec leurs « Datasets » et que XML utilise avec d'autres sources de données non-XML. Englober les différences de données sources, par la conversion dans et hors d'une abstraction de données commune fournissant un modèle unifié unique peut sembler approprié à plusieurs titres. Premièrement, les relations entre les différentes sources de données sous-jacentes peuvent être vérifiées et appliquées lorsque les données sont manipulées et modifiées. Deuxièmement, les vues sur les données peuvent tirer parti du mécanisme d'abstraction en rendant secondaire la création de mécanismes MVC (Model-View-Controller) spécifiques et l'utilisation des bibliothèques et des frameworks existants pouvant offrir cette abstraction.

Ce modèle n'est pas adapté à tous les ensembles de structures de données formatées de façon hétérogène et peut se révéler inutile pour certains formats de données dont les niveaux d'impédance relative à l'abstraction des données sont très élevés, les données de type image avec des données d'un autre type, par exemple. De plus, la transformation et la conversion des données dans et hors du format commun ont un coût potentiellement élevé. Pour de nombreuses applications, toutefois, ce coût est tout à fait acceptable.

Dans certains environnements de consommation de données, tels que le navigateur Web, les possibilités de conserver un format de données commun sont limitées et JSON et le XML DOM tendent à être assez populaires pour les solutions basées sur le navigateur. Côté serveur, les choix sont beaucoup plus nombreux mais dépendent souvent de la plateforme. Les structures XML, les bibliothèques d'Object Relational Mapping comme Hibernate, les bases de données relationnelles, voire même les graphes d'objets natifs font souvent, en fonction de l'application, d'excellents modèles d'abstraction commune. Mais la nature très différente des données hiérarchiques, telles que XML et les graphes d'objets, est l'un des problèmes d'adaptation d'impédance classique en informatique et une attention particulière doit être apportée lors de l'utilisation de ce modèle.

Résultat : si les performances ne sont pas indispensables et que les schémas et formats de données sous-jacents sont souples, ce modèle peut se révéler très puissant lors de l'utilisation de sources de données fédérées. L'inconvénient est que l'approche d'abstraction commune peut certainement impliquer plus de maintenance et générer de la fragilité car le mappage et les métadonnées doivent être maintenues.

« Pattern 4 » : structures d'origine modifiées avec MVC (Model-View-Controller). La conversion de toutes les données sources dans un format commun ne se révèle pas une option dans de nombreuses situations car 1) le surcoût du traitement est excessif dans la mesure où un grand nombre de données est susceptible de ne pas être utilisé ou 2) leur duplication dans l'environnement local peut être prohibitif en termes de ressources. Cela peut être aussi parce qu'il n'existe pas de format commun adapté à tous les types de données sous-jacents. Dans ce cas, la création d'un MVC qui modifie l'accès, la traduction et l'intégrité des données avec les structures d'origine sous-jacentes peut fournir les meilleures options pour les performances et le stockage des données.

> « INTERNET DEVIENT LE FOURNISSEUR LE PLUS IMPORTANT DE DONNÉES HAUTEMENT FÉDÉRÉES, UNE SOURCE QUI CONTINUERA À S'ACCROÎTRE DANS LES PROCHAINES ANNÉES »

MVC est un modèle de conception puissant à part entière qui s'est souvent révélé une excellente stratégie de séparation des problèmes dans les logiciels d'applications. Son utilisation est particulièrement indiquée lorsqu'il existe plusieurs modèles de données sous-jacents dans une application donnée. Une bonne conception logicielle implique qu'une vue unifiée des données sources offre une interface unique, propre et cohérente qui permet facilement de visualiser les données sous-jacentes, d'interagir avec elles et de les modifier. L'accès à ces dernières à l'aide de l'approche MVC est également relativement efficace, car seules les données requises doivent être traitées pour répondre à la plupart des requêtes.

Bien qu'il existe de nombreux avantages à utiliser le MVC, les inconvénients sont toutefois similaires à ceux du modèle 3, à savoir que la maintenance du code MVC peut être énorme. Sans l'ombre d'un doute, il existe un nombre croissant de bibliothèques prêtes à utiliser qui peuvent aider les concepteurs de logiciels à créer un modèle MVC sur le client et sur le serveur. Soyez attentif, toutefois : le code de mappage est fragile et fastidieux.

« Pattern 5 » : accès direct aux structures de données natives. Pour de nombreuses applications, notamment celles qui sont simples et basées sur un navigateur, la conversion des données sources en des formats communs ou la création d'architectures MVC sophistiquées n'est pas une bonne option. L'accès direct aux données est plus logique et la décision tombe souvent sous le sens, lorsque l'on dispose des bibliothèques. De plus, comme mentionné précédemment, la *simplicité* est souvent plus indiquée et garante d'une qualité supérieure, car il y a moins de remise en cause de l'existant ou de maintenance.

Dans ce modèle, qui fonctionne au mieux avec moins de données hautement structurées, les structures de données natives sont utilisées directement sans intermédiaire ou conversion de données, ce qui signifie que les magasins de données aux formats texte, XML et JSON, entre autres, sont manipulés en mode natif. L'inconvénient, bien sûr, est que vous ne pouvez pas avoir recours aux moyens que les bibliothèques d'abstraction de données peuvent vous donner pour appliquer les changements relatifs à l'intégrité ou au suivi. Toutefois, ce modèle est celui qui requiert souvent le moins de traitement ou d'étude des bibliothèques tiers. Il peut être facilement développé et se révèle également plutôt rapide puisqu'il n'y a pas de conversion ou de nécessité de traverser des couches d'accès aux données.

«Pattern 6 » : la modification des données est atomique. Un nombre plus important de vues très élaborées sur les données et de services sophistiqués rend obligatoires des propriétés connues sous le sigle ACID (pour atomicité, concurrence, isolation et durabilité). Habituellement attribuée aux systèmes de bases de données, ACID constitue, en règle générale, une

Tableau 1 : abstractions de données

Abstraction	Norme de contrat	Avantages	Inconvénients
Texte, JSON, binaire	Informel, textuel	Relativement efficace	Pas autodescriptif, pas réellement efficace
Objets natifs	Définition des classes	Comportement et données unifiés, encapsulation, abstraction de haut niveau et composition	Requiert la conversion de la plupart des données en objets, nécessitant une technique de mappage
XML	Schémas XML (XSD), Relax NG, WSDL et de nombreux autres	Autodescriptif, description de schéma riche et compatibilité descendante extensible sans fragmentation	Très inefficace en termes de taille, pas de possibilité de distribuer le comportement et les descriptions de schémas sont limitées même avec XSD
Images	Spécifications pour JPEG, TIFF, GIF, BMP, PNG et de nombreux autres formats	N/A	N/A
Audio	Spécifications pour WAV, WMA, MP3 et AAC	N/A	N/A
Vidéo	Spécifications pour AVI, QuickTime, MPEG et WMF	N/A	N/A

bonne pratique pour presque tous les systèmes d'accès aux données concurrents. Malheureusement, presque personne encore dans le monde des services Web et des « mashups » n'intègre la notion de transaction dans leurs protocoles, ce qui conférerait nombre des propriétés ACID à la modification des données. Loin d'ignorer le problème, les développeurs de « mashups » et d'applications composites doivent être tout à fait conscients qu'ils marchent sur une corde raide lorsqu'ils utilisent leurs données.

Cette prise de conscience est à l'origine de problèmes significatifs avec les scénarios de modification de données complexes, à savoir que les récupérations et stockages de données dépendant d'une conversation étendue avec des services sous-jacents ne sont pas protégés par la même frontière de transaction et les mêmes propriétés ACID correspondantes. Le code logiciel doit s'attendre à ce que des problèmes d'intégrité des données aient lieu, en particulier dans une conversation étendue qui risque d'échouer ou de ne jamais arriver à terme. Éviter des conversations de longue durée est une option. Rendre, autant que possible, une opération logicielle dépendante de services sous-jacents dans l'interaction atomique individuelle est une autre solution. Chaque étape de l'interaction constitue une validation unitaire visible qui permet au logiciel d'offrir à ses utilisateurs des options claires lorsqu'une conversation avec un service de données du fournisseur rencontre une défaillance. Ces deux options permettent aux développeurs de logiciels de respecter, en règle générale, les propriétés de transaction ACID.

Retour à la simplicité

Internet devient le fournisseur le plus important de données hautement fédérées, une source qui continuera à s'accroître dans les prochaines années. Alors que XML et les formats de données simples seront probablement la structure de données utilisée par la plupart des logiciels dans un futur proche, d'autres formats au parcours plus hasardeux sont en train de faire leur apparition. Il s'agit des optimisations particulièrement nécessaires dans XML, telles que le XML binaire, les avancées dans les microformats, de nouveaux codecs multimédia qui transformeront tout un paysage audiovisuel ou encore les protocoles de transport entièrement nouveaux, comme Bittorrent, qui rendront, c'est le moins que l'on puisse dire, nombre des modèles évoqués ici problématiques.

Un retour à la simplicité en matière de conception de données est à nouveau en vogue avec, pour exemple, le regain d'intérêt pour des formats tels que JSON, les microformats et les langages dynamiques comme PHP et Ruby. Cette simplicité de la conception peut rendre les données plus malléables et plus faciles à connecter entre elles. Elle présente également moins de difficultés pour écrire des logiciels visant à les gérer. Bien que les modèles présentés ici soient ceux qui aboutissent à une composition et une consommation des données à la fois moins fragiles, plus faiblement couplées et à haute intégrité, l'histoire ne fait que commencer... Internet portant de moins en moins sur les pages Web visuelles et de plus en plus sur les services ainsi que sur les données et contenus purs, devenir un consommateur et un fournisseur souple de l'écosystème des informations sera un facteur de succès toujours plus important. •

Ressources

microformats
http://microformats.org

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

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

Hewlett-Packard Development Company
Rapports techniques
« Rethinking the Java SOAP Stack », Steve Loughran and Edmund Smith
www.hpl.hp.com/techreports/2005/HPL-2005-83.html

À propos de l'auteur

Dion Hinchcliffe est le responsable technologique de Sphere of Influence Inc.

Réplication de données : un « antipattern » d'une SOA d'entreprise

par Tom Fuller et Shawn Morgan

Résumé

La réplication des données est souvent utilisée comme solution d'intégration rapide, dans le cadre des applications transverses à l'entreprise. Ce choix en faveur de la facilité engendre redondance et incohérence. En abusant de cet antipattern, on dilue les objectifs proposés sur le long terme par une architecture orientée services (SOA). Toutefois, selon le contexte dans lequel elle apparaît, la réplication des données peut être soit positive, soit négative. Pour fournir une SOA performante, les architectes d'entreprises doivent s'inspirer des succès et des échecs d'autres architectures. Les patterns d'architecture, lorsqu'ils sont découverts et documentés, peuvent fournir la meilleure solution pour accomplir la tâche difficile qui consiste à influencer les changements dans l'entreprise. Dans cet article, nous utiliserons un pattern ainsi qu'un antipattern pour expliquer les conséquences de la réplication de données sur votre architecture d'entreprise.

Rappelons en premier lieu que la mise en place des architectures orientée services n'en est qu'à ses débuts. Aussi, pour que la SOA atteigne les objectifs ambitieux envisagés par l'industrie informatique, les architectes devront s'appuyer sur les meilleures pratiques, qui plus est documentées. C'est là que les patterns et les antipatterns vont nous aider à combler l'écart entre la conceptualisation et la réalité. Les patterns constituent une stratégie répétable et éprouvée pour atteindre cet objectif. Les patterns et antipatterns d'architecture logicielle offrent des exemples documentés de succès et d'échecs dans l'application de ces stratégies d'architectures logicielles à l'informatique. En identifiant avec succès des abstractions architecturales, la SOA pourra commencer à trouver des stratégies d'implémentation garantes de résultats positifs.

Nous expliquerons pourquoi la réplication des données, lorsqu'elle est utilisée de façon récurrente dans votre SOA d'entreprise, peut aboutir à une architecture compliquée et coûteuse et vous apporter une stratégie de refactorisation pour passer de la réplication des données (antipattern) aux services directs (pattern). Ensuite, nous parcourrons les domaines dans lesquels la réplication des données apporte de vraies réponses (la réplication comme pattern cette fois-ci) sans compromettre d'autres initiatives d'architecture. Les architectes retireront de cette discussion une stratégie qui les aidera à éviter des fautes architecturales.

L'industrie logicielle a, dans un souci de cohérence, convenu d'une matrice afin de documenter les patterns de conception. Mais il est apparu que cette matrice est insuffisante pour décrire les antipatterns d'architecture et leurs possibles issues en termes de refactorisation. Un ensemble de sections des matrices de patterns a été consolidé afin de définir une matrice pour l'antipattern Replication ainsi que le pattern décrit dans cet article (voir Ressources). Le tableau 1 présente l'ensemble de sections de haut niveau et si elles peuvent être appliquées à un pattern, un antipattern ou aux deux.

Réplication de données : un antipattern d'architecture SOA

Commençons par une description détaillée des raisons qui font de la réplication de données un antipattern lorsqu'elle est appliquée dans une SOA. Nous utiliserons, pour cela, la matrice de l'antipattern afin d'expliquer les particularités et les contraintes du contexte qui justifient cette architecture. Un exemple illustrera les problèmes liés à l'utilisation de cet antipattern. Pour conclure, nous présenterons la solution refactorisée et les avantages liés à l'utilisation du pattern plutôt que de l'antipattern.

Contexte. Cet antipattern décrit un scénario couramment rencontré par les architectes qui essaient de construire des solutions SOA d'entreprise en environnement distribué. Le passage d'un environnement grand système à un environnement distribué est marqué par une nécessaire modification dans la gestion des données de référence (ces données critiques qui font vivre nos activités métiers au quotidien). Nous pouvons être tentés de créer ces nouvelles applications distribuées selon un modèle en silo, dans lequel l'application est définie comme un ensemble de fonctionnalités métiers,

« L'INDUSTRIE LOGICIELLE A, DANS UN SOUCI DE COHÉRENCE, CONVENU D'UNE MATRICE AFIN DE DOCUMENTER LES DESIGN PATTERNS. MAIS CETTE MATRICE EST APPARUE INSUFFISANTE POUR DÉCRIRE LES ANTIPATTERNS D'ARCHITECTURE AINSI QUE LEURS POSSIBLES ISSUES EN TERMES DE REFACTORISATION. »

exposées au travers d'une interface utilisateur et chaque fonctionnalité est mise en œuvre dans un environnement qui lui est propre. Cette décision va dans le sens des architectures qui favorisent un isolement fort voire extrême ainsi qu'une inutile agrégation d'entités métiers.

Forces. On retrouve cet antipattern dans une SOA, parce qu'il rassure architectes et concepteurs. Même lorsqu'il est présenté comme antipattern dans un contexte SOA, il continue à séduire pour les raisons suivantes :

- *L'architecte connaît bien la stratégie de réplication des données*. Les techniques utilisées dans la réplication sont bien rodées par rapport au tout début des développements batch, ce qui crée un confort certain pour l'architecte qui conçoit, depuis des années, des systèmes avec réplication des données à l'aide de transferts de fichiers ou d'outils d'extraction, de transformation et de chargement de données (outils ETL). Cela semble suffisamment simple pour continuer à utiliser cette stratégie pour d'autres usages.

Tableau 1 Tableau 1 Corrélation entre les sections des patterns et antipatterns

Section	Description	Pattern	Antipattern
Nom	Un nom court utile pour décrire rapidement le pattern ou l'antipattern.	X	X
Contexte	Les informations de fond utiles, qui décrivent le domaine d'application du pattern ou de l'antipattern.	X	X
Forces	Il existe plusieurs raisons pour lesquelles cette solution est utilisée. Très souvent, dans le cas d'un antipattern, ces raisons sont basées sur une erreur d'appréhension du domaine ou un manque de connaissances.	X	X
Solution	Cette section décrit la solution proposée en fonction du contexte, des contraintes et enjeux documentés. Elle représente la stratégie ou l'intelligence du pattern.	X	
Solution médiocre (antipattern)	Cette section décrira l'antipattern ou la solution à un problème qui semble avoir des avantages, mais qui, en fait, aura des implications négatives lors de son utilisation.		X
Conséquences	Les conséquences sont les effets de bord sur la solution implémentée. Dans le cas d'un antipattern, les conséquences seront toujours composées d'effets négatifs qui l'emportent sur l'avantage du pattern souhaité.	X	X
Solution refactorisée	Chaque antipattern devrait avoir un contraire qui serait, en fait, une solution ou un pattern, et qui serait documenté comme c'est le cas des patterns.		X
Avantages	Les avantages seront les mêmes que pour un pattern documenté et la solution refactorisée d'un antipattern.	X	X
Exemple	Un exemple réel où le pattern/l'antipattern est appliqué pour mettre en lumière/résoudre un problème.	X	X

- *L'architecte se soucie des performances des services exposés par le système de gestion des données de référence.* Par exemple, accéder à distance à un entrepôt de données au travers de services peut ralentir la nouvelle solution par rapport à une solution qui ferait appel à une base de données locale. Nous qualifierons ce scénario de One-To-Many (dans le sens d'un référentiel de données à plusieurs).
- *Des services distincts peuvent présenter des vues ou aspects différents d'une entité métier.* Le regroupement de ces différentes vues en une entité agrégée ainsi que le fait de s'assurer de l'autonomie de chacun des services sont considérés comme trop difficiles. Aussi, les données exposées par ces services sont répliquées dans un seul référentiel. Nous qualifierons ce scénario de Many-To-One (dans le sens de plusieurs vues sur ces données, à un nouveau référentiel de données).
- *L'architecte cherche à éviter que la disponibilité du nouveau système puisse être compromise.* Par exemple, si le nouveau système doit accéder aux données de référence via un service, les défaillances du réseau ou du système de gestion de données de référence peuvent provoquer la non-disponibilité du nouveau système. Disposer des données en local est séduisant pour réduire ce risque.
- *Les applications sont construites selon un modèle en silo.* Une application est définie comme un ensemble de fonctionnalités métiers exposées au travers d'une interface utilisateur ; chaque fonctionnalité métier est développée dans son propre contexte sans qu'un architecte d'entreprise n'invite les développeurs de l'application à la réutilisation de services disponibles et répondant à leurs besoins.
- *Les ressources pour créer une solution à l'aide de la réplication des données ne manquent pas.* Les compétences requises pour implémenter ce type de solution sont généralement faciles à trouver. Par contre, créer des applications qui tirent parti de services nouveaux ou existants exige des compétences plus rares : de bonnes connaissances en systèmes distribués, technologies liées aux services Web, programmation orientée objet et autres technologies de nouvelle génération.

Des lignes de contrôle imprécises

Solution médiocre (antipattern). Pour réaliser cet antipattern, des données de référence sont répliquées depuis la base de données de référence vers une nouvelle base de données créée pour les besoins des fonctionnalités de la nouvelle application (voir figure 1). Cette réplication peut être réalisée par différents mécanismes, tels qu'un outil ETL, ou par des mécanismes plus élémentaires, tels qu'un transfert de données par fichiers. Les données de référence peuvent enfin être enrichies par les nouvelles applications, avec autant d'attributs supplémentaires par rapport à l'entité d'origine.

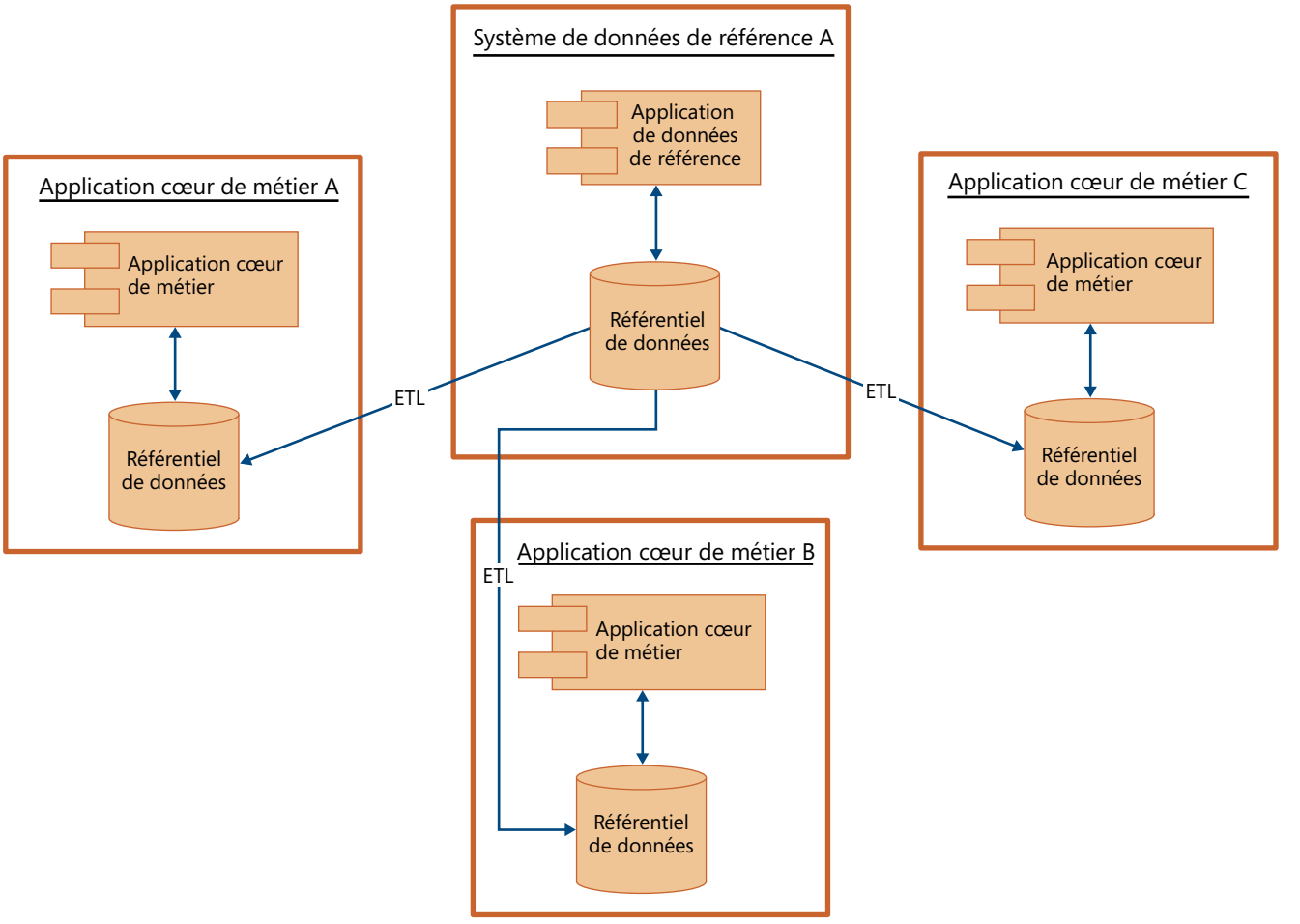
Conséquences. Les conséquences de ce pattern appliqué de façon erronée à une architecture SOA ne sont pas immédiatement visibles. Lorsque les données sont répliquées de façon éclatée, depuis le système qui les possède vers les multiples bases liées aux différentes applications, votre architecture orientée service devient très difficile à gérer. Les données répliquées deviennent une pratique répandue à travers l'entreprise. Une définition imprécise des responsabilités quant aux données complique alors leur contrôle. Plusieurs services manipulent les mêmes données (ou les données en partie enrichies). La vue entreprise des données métiers devient disparate.

Ce flou au niveau des responsabilités sur les données et la perte de contrôle sur les données de référence est un phénomène déconcertant pour tout architecte. La complexité augmente à mesure que les données prolifèrent et d'autant plus que les applications viennent les modifier et les compléter. Les nouveaux champs ajoutés aux données dans une base locale à une application peuvent être importants à un niveau entreprise aussi, mais risquent de ne pas être réintégrés dans la base de données de référence. Bien entendu, compte tenu des délais et des budgets serrés de la plupart des projets d'entreprises, une équipe projet prendra rarement le temps de remonter les nouveaux champs de données métiers vers le système de données de référence, pas plus que d'exposer ces nouveaux champs vers des services qui pourront alors être utilisés par la base de données de référence, ou encore de créer une couche d'agrégation de services (qui prendrait alors le relais du système de gestion des données de référence initial).

D'ailleurs, ces équipes pourraient aller au-delà du champ de leur application en exposant ces nouvelles données de référence. Ou encore, elles pourraient créer de nouveaux services qui imiteraient une partie des fonctionnalités du système de gestion des données de référence afin d'exposer les données qui ont été répliquées vers leur propre système. Cette exposition est à l'origine d'un véritable casse-tête et défi à relever pour les architectes d'entreprises. Ils doivent, à présent, éviter que de nouveaux systèmes ne consomment des services exposant des données répliquées, ce qui créerait une confusion, au sein de l'architecture d'entreprise, quant à la véritable propriété de l'objet métier.

La réplication s'accompagne par ailleurs d'une responsabilité majeure pour le système de réplication, chargé de dupliquer aussi la logique métier du système de données de référence. En effet, les données font très souvent l'objet de manipulation par le système de gestion des données de référence, avant d'être exposées sous forme de service aux autres applications. Dans ces cas, la méthode *éclair* qui consiste à répliquer des données sur le nouveau système nécessite aussi de répliquer la logique métier utilisée pour manipuler les données de référence. À moins qu'une analyse approfondie ne soit effectuée sur le système de gestion des données de référence, la logique peut ne jamais être implémentée, ou l'être de façon incorrecte. On trouve un dernier scénario, beaucoup plus dommageable, dans lequel ne figure, au départ, aucune logique métier déroulée au moment de la récupération des données de référence ; la logique métier est dans ce cas ajoutée plus tard à l'occasion d'un projet dédié à la mise à niveau

Figure 1 La prolifération des données principales aboutit à une propriété distribuée des diverses permutations.



du système de gestion des données de référence. Ce scénario peut être douloureux pour une entreprise qui a répliqué des données plusieurs fois. Chaque système doit alors être analysé afin de détecter les modifications à effectuer pour que le système maître réponde aux nouvelles exigences des données d'entreprise.

« LES ENTREPRISES NE TIRERONT PLEINEMENT PARTI D'UNE SOA QUE LORSQU'ELLES AURONT PRIS CONSCIENCE QU'UN CHANGEMENT D'ÉTAT D'ESPRIT EST NÉCESSAIRE POUR CRÉER DES SOLUTIONS BASÉES SUR DES NOUVEAUX MODÈLES D'ARCHITECTURE. »

Généralisation des redondances

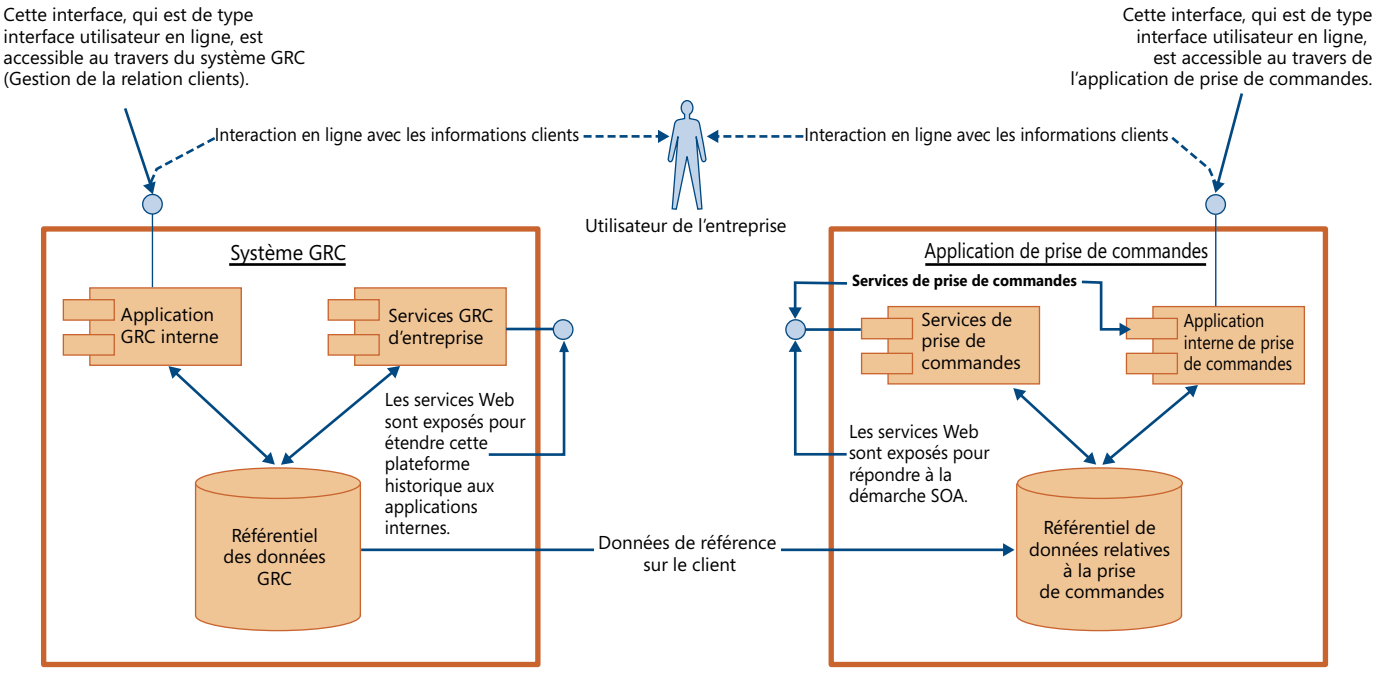
Il est tentant de négliger les conséquences de l'utilisation de la réplication des données parce que, à ne pas y regarder de près, ces conséquences ne semblent pas lourdes. Les problématiques telles que l'espace disque, les éléments réutilisables et le gain du travail semblent gérables, que l'on se tourne ou non vers une stratégie basée sur l'orientation services. La conséquence, qui éclipse toutes les précédentes, est cependant la capacité à minimiser la complexité et à fournir des solutions qui peuvent rapidement s'adapter au changement. La réplication de données est synonyme de complexité, de fragilisation et de non-flexibilité en raison des redondances généralisées qu'elle crée dans votre architecture d'entreprise.

Prenons l'exemple d'une nouvelle application ayant pour objectif de gérer les prises de commande. L'architecture exige que des services métiers soient créés pour le nouveau système et que l'interface utilisateur, qu'il s'agisse d'un portail ou d'une implémentation Web, consomme ces services métier.

Les besoins métiers requièrent du système qu'il implémente les fonctions de création, de modification et d'affichage des commandes. Une partie de la commande correspond à des informations client et est stockée dans un système de gestion de la relation clients (GRC). Ce dernier expose des services permettant de récupérer les informations associées à un client, mais l'architecte décide, à partir des motivations décrites précédemment, de répliquer les données client de référence vers le système de prise de commandes.

Un besoin du nouveau système de prise de commandes est de créer un numéro de compte pour chaque client. Ce champ additionnel est joint aux données client répliquées dans la base de données locale du système de prise de commandes. L'interface utilisateur affiche les informations du client via un nouveau service Web créé pour le système de prise de commandes fortement verticalisé. D'autres services métiers sont développés pour la création, la modification et l'affichage des commandes. Ces services sont consommés par l'interface utilisateur du système de prise de commandes. Toute manipulation ou modification des données effectuée par le système de gestion des données de référence lorsqu'on appelle les services d'accès aux données de référence, doit être répliquée dans le nouveau système. À partir de ce moment, toute modification apportée à ces données doit également être répliquée sur le nouveau système, de même que toute modification de logique métier pour accéder à ces données.

Figure 2 Redondance de données dans le contexte antipattern.



Le diagramme présenté à la figure 2 illustre clairement que la réplication des données aboutit à la duplication des services. Chaque application possède désormais une représentation propre de l'entité client, avec son propre moyen d'accès aux données au travers d'un service créé dans ce but. Le système de prise de commandes copie et étend les données client via son propre ensemble de services, ce qui crée une redondance et une confusion pour n'importe quel nouveau système qui cherchera à consommer ces services.

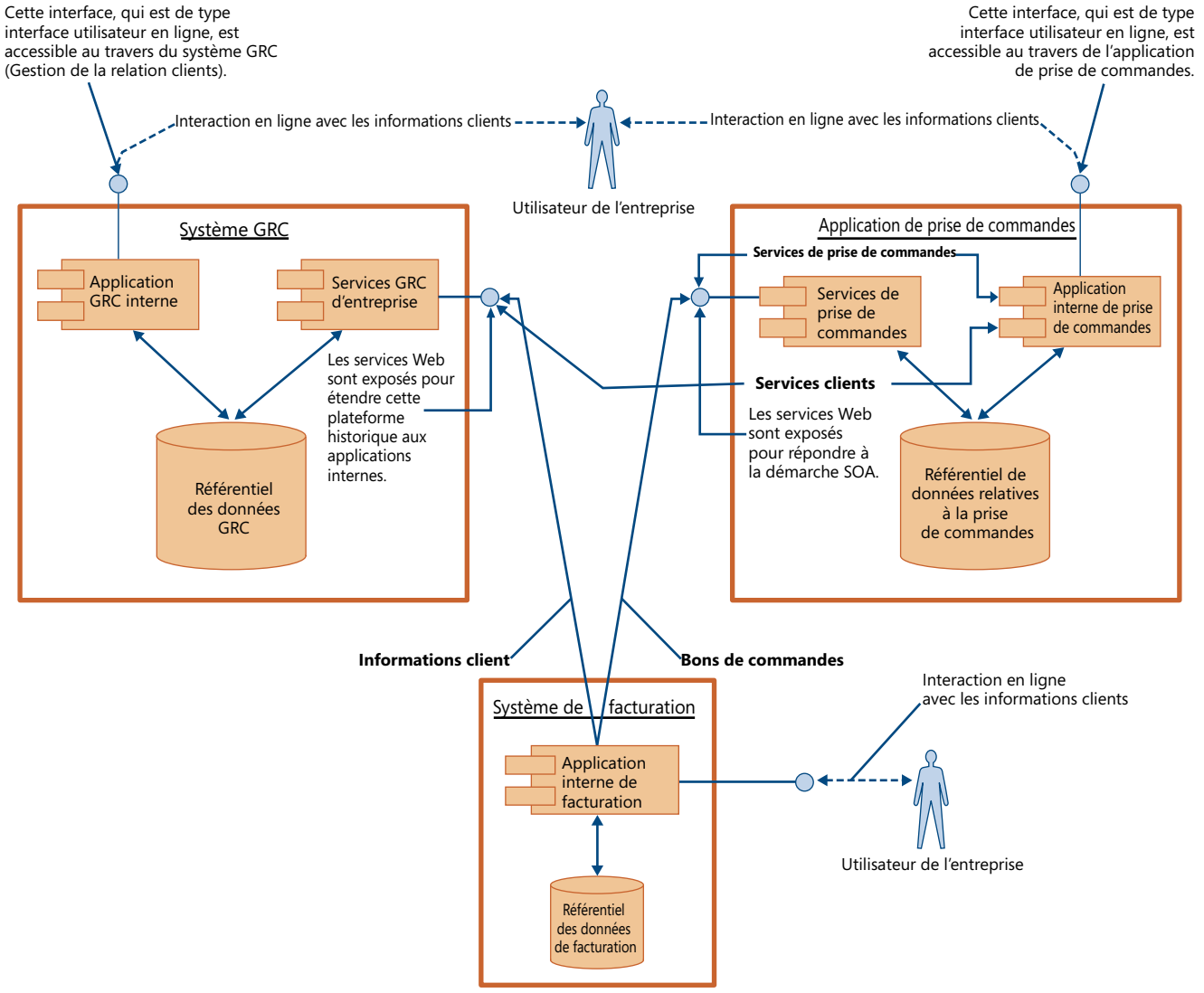
L'exemple suivant concerne un système de gestion de la facturation/comptabilité. Ce système a également besoin des données client, y compris du numéro de compte. Plusieurs possibilités se présentent pour les concepteurs de ce système. Vont-ils récupérer les données de référence du client au sein du système de GRC, tandis que le numéro de compte client sera accédé au travers du système de prise de commandes ? Ou bien vont-ils préférer aller chercher toutes les informations client dans le système de prise de commandes (où résident toutes les données) ? Dans ce cas, la problématique de disponibilité et de pérennisation du service étant une préoccupation majeure pour les architectes (une force dans cet antipattern), la décision est prise de répliquer les données client depuis le système de prise de commandes vers le système de facturation. Trois systèmes disposent désormais d'une représentation, plus ou moins complète, des données client. Ces trois systèmes sont forcés de créer une logique métier autour de l'accès à ces données. Chacun d'eux expose les données client via des services (car nous construisons une « architecture orientée services ») et la capacité de l'entreprise à gérer les données client de façon souple continue tranquillement de se détériorer.

La reconnaissance de la nécessité d'un changement

Solution refactorisée. Les entreprises ne tireront pleinement parti d'une SOA que lorsqu'elles auront pris conscience du changement d'état d'esprit nécessaire pour créer ces solutions en mettant en œuvre des patterns d'architecture. La SOA ne pourra fournir tous les bénéfices attendus tant que les architectes ne comprendront pas que c'est seulement en fournissant un point de contrôle unique des données et services métiers que les entreprises pourront devenir effectivement réactives, flexibles et auront la capacité à grossir. Pour refactoriser votre système avec succès, vous devez trouver une réponse aux motivations qui sont à l'origine de l'antipattern :

- *L'architecte est habitué à utiliser une stratégie de réplication des données.* Aux indicateurs clés de performance qui concernent une implémentation SOA, ajouter un indicateur qui permette de ne pas répliquer des données inutilement. Les formations aux implémentations SOA qui réduisent les besoins en réplication de données peuvent aussi limiter cette tendance.
- *L'architecte se préoccupe des performances d'un service exposé par le système de gestion des données de référence.* Tester et valider les performances d'un service exposé. Grâce aux tests et à une bonne compréhension du niveau de qualité de service (SLA) requis pour la nouvelle application, de nombreux architectes constateront que l'utilisation d'un service nouveau ou existant qui expose les données de référence fonctionnera conformément aux exigences du SLA.
- *Des services distincts peuvent posséder des représentations quelque peu différentes d'une entité.* Les compétences et technologies pour regrouper les différentes représentations dans une entité agrégée et pour maintenir l'autonomie des services sont déjà bien identifiées. Ce travail peut généralement être réalisé au moment de l'exécution.
- *L'architecte cherche à éviter que la disponibilité du nouveau système ne soit compromise.* Traiter la question de la disponibilité des services métiers exposés au travers des solutions disponibles au niveau de l'infrastructure, comme le clustering par exemple.
- *Les applications sont créées selon un modèle en silo.* Dans de nombreux projets, la participation d'un architecte se révèle nécessaire. Il s'intéressera particulièrement aux services nouveaux et existants et aux fonctionnalités qu'ils doivent fournir. D'ailleurs, les applications d'aujourd'hui devront être revues pour passer du stade de boîte noire à un assemblage de services existants dans l'entreprise et nouveaux dans le cadre de l'application.
- *Les ressources pour construire des solutions mettant en œuvre la réplication des données sont nombreuses dans l'entreprise.* Dans ce cas, mettre à disposition des concepteurs et développeurs de niveau « intermédiaire » les patterns, les matrices, les normes et outils actuellement disponibles permet d'implémenter de véritables SOA.

Figure 3 Le système refactorisé



Une analyse détaillée de ces problématiques vous permettra d'élaborer votre stratégie de mise en place d'architecture. Face aux décisions à prendre lors du passage à une architecture orientée services, nous devons reconsidérer certaines questions fondamentales. Plutôt que de se focaliser sur l'architecture de leur « application », les architectes devraient s'intéresser à l'architecture des services de l'entreprise. Les services devraient être créés sur le système qui maintient et gère les données ou bien des services d'agrégation devraient être créés pour fournir les données, éliminant ainsi le besoin de les répliquer vers une autre application chargée d'assurer un service.

La vision qu'ont les architectes d'une application, telle qu'elle s'applique à l'entreprise, doit fondamentalement changer. Pour qu'une SOA soit un succès, nous devons créer non plus des applications autonomes mais plutôt élaborer des produits, à savoir un éventail de fonctionnalités à livrer à l'utilisateur. Nous devrions créer des services qui fournissent ces fonctionnalités métiers. Le produit n'est rien de plus qu'une composition ou une orchestration d'un ou plusieurs services, qui fournissent une ou plusieurs parties d'une fonctionnalité métier. Il s'agit de l'agrégation de cas d'utilisation déjà implémentés.

Supprimer l'ambiguïté

Dès lors que cette stratégie est clairement définie et que les tendances de fond à la réplication ont été combattues, l'implémentation de services cœur de métier peut se concrétiser. Les interfaces utilisateur des nouveaux systèmes peuvent appeler des services existants lorsque nécessaire, sans aucune indirection de données.

La refactorisation de l'architecture selon une stratégie d'accès à des services supprime l'ambiguïté quant à la propriété des données. Les nouvelles applications, telles que le système de facturation (voir figure 3), sont capables de récupérer des données depuis les systèmes de stockage maîtres.

Avantages. Les architectes sont constamment appelés à évaluer les inconvénients liés à l'utilisation d'un certain pattern au sein d'une solution. Lorsque l'entreprise tente de passer à une architecture SOA, pour toutes les raisons qui en font une architecture de poids pour l'entreprise, les inconvénients générés par la réplication des données dans le but d'améliorer les performances de l'application ou sa disponibilité doivent être confrontés aux avantages en termes de clarté et de simplicité qu'apporte une architec-

ture de services. Ces avantages sont possibles grâce à la clarté des responsabilités de propriétés des données de référence ainsi qu'à la gouvernance rendue possible au niveau global de l'entreprise.

Les indicateurs clés de performances, qui peuvent être la source de motivation de la mise en place d'une architecture SOA, constituent un intérêt supplémentaire de choisir un modèle d'accès direct via des services. L'orientation services est une nouvelle tentative visant à disposer d'applications dont les livrables puissent facilement être mis à jour. La réutilisation de services peut produire d'importantes économies, à la fois en termes d'espace de stockage et en coûts de main-d'œuvre. Certes, la réplication des données permet de répondre aux exigences fonctionnelles, mais les objectifs d'une dimension entreprise de l'architecture sont alors loin d'être atteints.

Un autre avantage à noter réside dans l'efficacité obtenue en ne récupérant que les données dont les applications ont besoin. Dans cette optique, les architectures de type EDA qui accèdent aux données au travers d'événements, seront sans aucun doute plus efficaces que les architectures traditionnelles qui font appel à des processus de type batchs. Répliquer l'ensemble des données au cours d'un traitement par lot dont l'exécution est limitée dans le temps entraîne d'importants coûts additionnels. Sans un mécanisme permettant de récupérer les données adéquates au bon moment à l'aide de services, vous consommez des ressources pour déplacer des données qui peuvent avoir peu ou aucun intérêt pour les fonctionnalités de l'application. Le pattern d'accès aux données via des services garantit que vous n'exécutez pas de traitements inutiles.

À terme, l'objectif de tout service qui expose des données d'entreprise est de présenter une vue aussi complète que possible d'une entité sans sacrifier les avantages des initiatives d'architecture d'entreprise. Lorsque des données cruciales de l'entité à exposer sont fragmentées sur de multiples systèmes, la conception du service peut exiger un certain niveau d'agrégation. Mais cela n'écarte pas la possibilité d'utiliser le pattern de services d'accès aux données pour récupérer ces différents constituants de l'entité.

« UN AUTRE AVANTAGE RÉSIDE DANS LES NIVEAUX D'EFFICACITÉ QUI SONT OBTENUS EN NE RÉCUPÉRANT QUE LES DONNÉES DONT VOUS AVEZ BESOIN. LES ARCHITECTURES QUI TABLENT SUR LA RÉCUPÉRATION DE DONNÉES BASÉES SUR LES ÉVÉNEMENTS SERONT SANS AUCUN DOUTE PLUS EFFICACES QUE CELLES QUI FONT APPEL AUX PROCESSUS TRADITIONNELS DE TYPE BATCH. »

Une architecture qui implémente et consomme de façon efficace des services d'accès aux données minimise la fragilité du portefeuille des solutions de l'entreprise en supprimant la redondance. Grâce à la centralisation et à la réutilisation, les services créés à l'aide de cette stratégie offrent le meilleur moyen de rester agile dans un environnement en changement.

Réplication des données comme pattern

Toutes les entreprises disposent de systèmes et de contraintes complexes qui obligent les architectes à rester pragmatiques face aux modèles qu'ils soutiennent. Dans ces scénarios, un architecte doit, au travers d'une bonne gouvernance d'entreprise, contrôler la généralisation de l'adoption des stratégies de réplication. Dans le cas contraire, l'entreprise risque d'abuser de l'utilisation de ce pattern. Malgré le fait que les développeurs, les concepteurs et les architectes la considèrent simple à utiliser, la stratégie de réplication des données ne devrait être considérée comme un pattern que dans de rares occasions.

Figure 4 Composant de déplacement de données

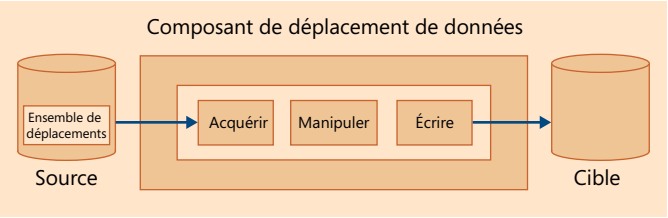
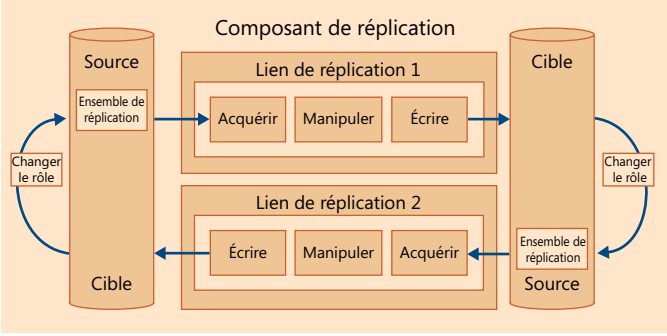


Figure 5 Un scénario de réplication de type maître-à-maître



Contexte. Le contexte dans lequel la réplication de données est un véritable pattern diffère quelque peu de celui où elle est un antipattern. Il s'agit bien là encore d'un environnement distribué mais pour lequel vous souhaiteriez qu'une copie des données soit disponible dans un système qui est externe au système de gestion des données de référence. Dans certains cas, vous ne pourrez pas éviter la réplication. Par exemple : lorsque la latence du réseau est un problème, comme dans un WAN ; lorsque le système de gestion des données de référence n'est pas fiable ou s'il comporte des fenêtres de maintenance incompatibles avec la disponibilité de vos applications ; lorsque la capacité à fonctionner dans un scénario déconnecté est une exigence forte pour l'application ou bien encore lorsque les risques encourus par le fait de ne pas utiliser des services sont compensés par le cadre d'utilisation de l'application.

Forces. Les motivations qui influencent la décision d'effectuer une copie des données de référence et qui peuvent alors justifier la complexité induite par la duplication des données sont les suivantes :

- *La disponibilité des données sur le système de gestion des données de référence ne correspond pas aux exigences du nouveau système.* Par exemple, le système de gestion des données de référence doit être interrompu pour des raisons de maintenance entre certaines heures pendant lesquelles le nouveau système doit être disponible (et les données concernées sont indispensables pour l'exécution des cas d'utilisation du nouveau système).
- *Le réseau n'est pas fiable ou est trop lent*. Exemple : sur un WAN où le réseau déconnecterait de façon inopportune et incontrôlée le système de gestion des données de référence, copier les données serait un remède à ce problème. Un autre scénario serait celui d'un réseau trop lent pour supporter une consommation des données au fil de l'eau, lorsque demandé par l'application distribuée.
- *Le nouveau système doit avoir la capacité à travailler en mode déconnecté*. Cette contrainte est souvent une exigence propre aux systèmes qui auront très peu ou pas de connectivité aux services d'entreprise qui fournissent les données de référence.

- *Les données seront copiées à des fins d'analyse uniquement, sans aucune logique métier réutilisable.* Il est des situations dans lesquelles les données devront être copiées dans un entrepôt de données à des fins de comptes rendus analytiques. Dans ce cas, le transfert de données est souvent assuré par un outil ETL, qui simplifie la copie des données dès qu'elles sont considérées prêtes pour l'archivage.

Solution. La solution pour remédier à ces contraintes est la réplication des données. Celle-ci peut être réalisée à l'aide d'un composant d'architecture appelé *composant de déplacement de données.* Ce composant est constitué d'une source, d'un lien de type déplacement et d'une destination.

Le diagramme de la figure 4 présente le composant de base de nombreux patterns de déplacement de données (voir Ressources). Selon la problématique, vous devrez peut-être utiliser plusieurs composants de déplacement de données pour gérer des scénarios de type maître-à-maître où les données peuvent être mises à jour à la fois au niveau de la source et de la destination (voir figure 5).

Avantages. Une grande partie du travail décrit par les patterns précédents peut être réalisé avec des outils ou bien par des ressources qui ne relèvent pas du domaine du développement. Cet aspect est l'un des principaux avantages de cette approche, car vous constaterez souvent, au départ, un plus faible coût de possession. Cet avantage ne doit pas pour autant éclipser d'autres problématiques. La maintenance et la gestion de versions de ce type d'outils et de stratégies sont souvent très onéreuses sur le long terme. Si la priorité est le délai de commercialisation et que le temps de réalisation semble très court, ce pattern peut être viable et apporter de nombreux avantages. Gardez cependant à l'esprit que vous fragiliserez votre architecture d'entreprise, et ne négligez pas les coûts de refactorisation à venir. Une telle refactorisation peut parfois être exécutée en encapsulant un système historique existant, et ne répondant pas aux principes SOA, par une nouvelle interface de type SOA. Ainsi, l'architecture d'entreprise peut passer au SOA, dans sa globalité, sans avoir à refactoriser les applications concernées.

Documenter les meilleures pratiques

L'innovation technologique comporte un aspect inéluctable : le changement. Les architectes d'applications doivent s'attacher à réduire le risque de changement. Dès les premiers patterns orientés objet, l'utilisation de patterns et d'antipatterns pour documenter les meilleures pratiques a été couronnée de succès. Appliqués de façon opportune, les patterns d'architecture sont l'arme la plus fiable pour gérer le changement.

Le pattern et l'antipattern présentés dans cet article s'intéressaient à la réplication des données, et sa place au sein de l'initiative SOA d'une entreprise. En utilisant une matrice pour décrire le pattern, vous avez pu constater pourquoi cette stratégie peut avoir un effet positif ou négatif selon le contexte dans lequel il est appliqué. Les patterns utilisés pour résoudre ce décalage illustrent exactement combien l'adoption d'une méthodologie axée sur l'architecture peut constituer un facteur de réussite.

L'évolution perpétuelle de l'industrie logicielle ne peut qu'amplifier le rôle de l'architecture et des patterns. Les patterns, comme la réplication des données, qui sont communs dans les architectures traditionnelles peuvent désormais être préjudiciables à l'évolutivité de votre architecture d'entreprise. Les sociétés qui anticiperont le passage des applications aux services seront à même d'effectuer cette transition plus rapidement et de réaliser tout le potentiel du SOA.

« LA MAINTENANCE ET LA GESTION DE VERSIONS DE CE TYPE D'OUTILS ET DE STRATÉGIES SONT TRÈS ONÉREUSES À LONG TERME. SI LA PRIORITÉ EST LE DÉLAI DE COMMERCIALISATION ET QUE LE TEMPS DE RÉALISATION SEMBLE TRÈS COURT, CE PATTERN PEUT ÊTRE VIABLE ET APPORTER DE NOMBREUX AVANTAGES. »

Les objectifs de l'industrialisation logicielle, qui semblent pourtant très ambitieux et sans fin, s'imposent de plus en plus à nous. Les patterns sont les fondations de cohérence et d'agilité d'une architecture d'entreprise. Les traiter comme des artéfacts de première nécessité permettra d'accélérer leur découverte et leur adoption. Une entreprise qui constitue à terme sa bibliothèque réutilisable de patterns est plus réactive face à ses équipes métiers et possède un temps d'avance sur ses concurrents. •

Ressources

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 et Carsten Zeitz (Microsoft Corporation, 2003)

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

« SOA Antipatterns » (IBM, November, 2005)

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

À propos des auteurs

Tom Fuller est directeur technique et consultant senior en SOA chez Blue Arch Solutions Inc. (www.BlueArchSolutions.com), un cabinet de conseils en architecture et formation, ainsi que fournisseur de solutions basé à Tampa, en Floride. Tom a récemment reçu le prix MVP (Most Valuable Professional) de Microsoft dans la catégorie Visual Développeur – Architecte Solution. Il préside également la section de Tampa Bay de l'Association internationale des architectes logiciels (International Association of Software Architects ou IASA), détient une certification MCSD.NET et gère par ailleurs un site dédié à SOA, aux services Web et au framework Windows Communication Foundation (anciennement « Indigo »). Tom est l'auteur d'une série d'articles publiés récemment par l'alliance Active Software Professional et le magazine SQL Server Standard. Tom est invité comme orateur auprès de nombreux groupes d'utilisateurs dans le sud-est des États-Unis. Visitez le site www.SOApitstop.com pour plus d'informations et vous pouvez contacter Tom à tom.fuller@soapitstop.com.

Shawn Morgan est PDG et architecte senior chez Blue Arch Solutions, Inc. (www.BlueArchSolutions.com). Shawn a conçu l'architecture de solutions de nombreuses sociétés parmi les Fortune 500, dont Federal Express, Beverly Enterprises et Publix Super Markets. Shawn aide les entreprises à fournir des solutions informatiques au moyen de l'architecture. Contactez Shawn à shawn.morgan@bluearchsolutions.com.

Modèles de composition et de consommation des données à haute intégrité

par Dion Hinchcliffe

Résumé

Le défi consiste à consommer et à gérer des données de plusieurs sources sous-jacentes dans différents formats tout en participant à un écosystème d'informations fédéré et en conservant une intégrité et un couplage faible avec de bonnes performances. Voici quelques modèles émergents pour le monde en pleine expansion des « mashups » et des applications composites.

Aujourd'hui, le développement logiciel consiste autant à assembler et à composer des données et des services préexistants qu'à créer une toute nouvelle fonctionnalité. Les « mashups » sur le Web et les applications composites dans le domaine de l'architecture orientée services (SOA) requièrent le regroupement, l'interaction, puis la séparation de données aux formats très différents et de sources variées. Et ces approches nécessitent un bon fonctionnement de cette interaction, sans que la fidélité ou l'intégrité des données ne soit affectée.

La variété considérable de données existant à l'heure actuelle couvre toute une gamme de formats XML ainsi que des formats de données plus légers et de plus en plus répandus, tels que JSON (JavaScript Object Notation) et les microformats. Les applications aussi doivent de plus en plus souvent fonctionner avec des formats de données riches, tels que les images, l'audio et la vidéo, sans oublier les formats plus anciens utilisés au quotidien, tels que le texte, EDI ou les objets natifs. Tous ces formats sont de plus en plus mélangés les uns aux autres alors que les systèmes deviennent plus interconnectés et intégrés dans de grandes chaînes d'approvisionnement, dans des bus de services d'entreprise (ESB), dans des SOA et dans des « mashups » Web. Maintenir l'ordre dans ce chaos de données est plus important que jamais. Bonne nouvelle : des règles de premier ordre visant à gérer l'hétérogénéité de toutes ces données commencent à voir le jour.

Les services Web, la SOA et Internet, en particulier, sont constitués de normes ouvertes qui permettent aux systèmes de communiquer, la norme omniprésente et essentielle HTTP en étant un exemple édifiant. Toutefois, cette communication ne permet pas de connaître le type de données qu'utilisera votre ordinateur dans les applications de demain. Bien que l'on puisse tabler sur certains formats XML, il est désormais tout aussi probable que vous soyez confrontés à des formats de données simples à la popularité grandissante, tels que JSON. Toutefois, de plus en plus, il pourrait également s'agir d'un simple texte brut, d'une arborescence d'objets natifs, voire d'une pile multicouche de services Web de style WS-* fournie par la prochaine Windows Communication Foundation (WCF).

Les développeurs font donc face à de sérieux défis pour créer des techniques performantes d'intégration, d'architecture et de modélisation des données capables de répondre à cette diversité. L'hétérogénéité des formats de représentation des données peut être assez décourageante dans des environnements requérant un haut niveau d'intégration de systèmes. Peu importe que dans des systèmes très faiblement couplés et hautement fédérés, comme le deviennent de plus en plus d'applications, la probabilité de changements fréquents soit élevée. Il appartient à la communauté en charge du développement des applications de constituer un ensemble de connaissances sur les meilleures pratiques en matière de simple consommation de données de différents formats et sources, avec les techniques d'intégration, de fusion et de liaison correspondantes. Cette communauté doit également s'assurer que l'intégrité des données est conservée tout en veillant à ce qu'elles conservent leur résistance aux changements et ce, malgré l'inévitable évolution des formats de données sous-jacents.

Les modèles présentés ici, bien que ne faisant pas autorité et ayant été conçus à l'origine à titre d'aide, se basent sur le concept largement reconnu des contrats d'interface. Ces contrats d'interface sont désormais répandus dans le monde des services Web avec le WSDL ainsi que dans de nombreux langages de programmation mais, en particulier, lorsque la conception repose sur la définition d'un contrat. Dans un contrat d'interface, un opérateur et un fournisseur se réunissent pour décider d'un ensemble d'interactions, habituellement décrites sous les termes de méthodes ou de services. Ces interactions permettent de faire circuler des données ciblées dans les limites de l'interface.

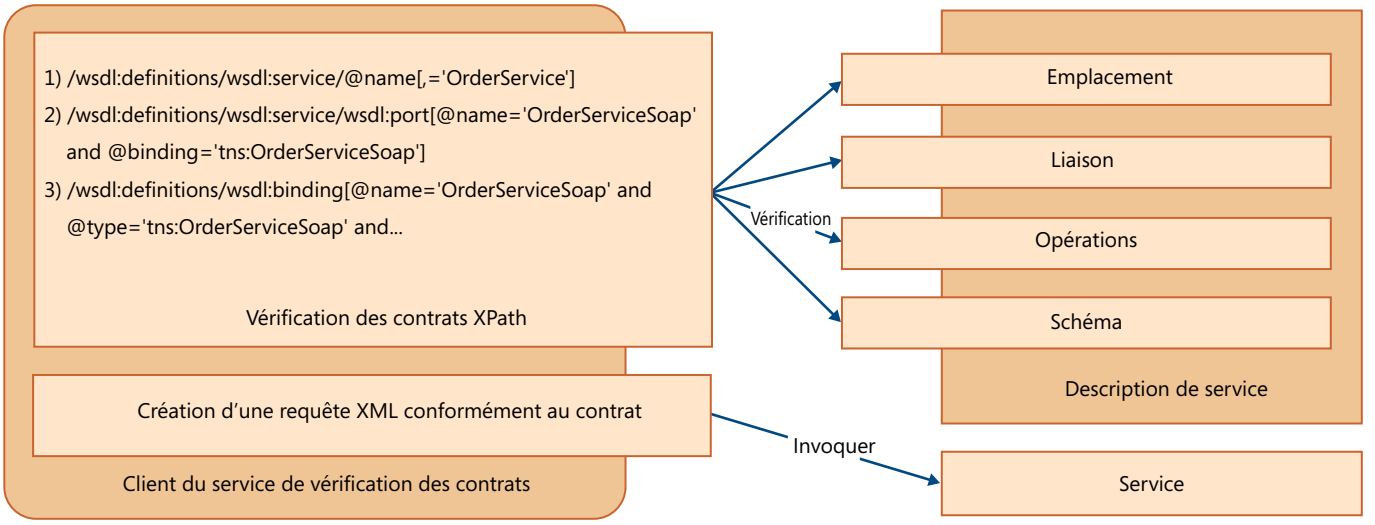
Le détail de l'interaction des données ainsi que leurs emplacements et protocoles sont définis avec précision dans le contrat qui, de préférence, est lisible par une machine. Les structures de données circulant en tant que paramètres, au cours de chaque interaction, entre le fournisseur et le client sont particulièrement intéressantes. Et ce sont ces structures de données qui sont au cœur du modèle de l'application composite et du « mashup ». Car c'est ici, au point de rencontre de ces structures de données très différentes les unes des autres, que nous devons appliquer au mieux les principes de recombinaison et d'intégration des données.

Forces et contraintes de la composition et de la consommation

En appliquant ces principes, nous pouvons obtenir une idée générale des forces qui définissent la composition et la consommation des données dans les logiciels distribués actuels. En voici le classement approximatif :

Le contrat d'interface en tant que pilote de la composition et de la consommation des données. Lorsque l'on utilise les structures de données d'une source ou d'un service externe, tel qu'un service Web, elles doivent être validées par rapport au contrat d'interface fourni par le service source. Cette validation peut souvent être effectuée au moment de la conception pour les sources de données reconnues comme hautement stables et fiables. Toutefois, dans les systèmes fédérés, notamment ceux qui ne sont pas sous le contrôle direct du consommateur, il convient de faire particulièrement attention à la vérification du contrat au moment de l'exécution. Cette vérification, dans des systèmes faiblement couplés, est une technique émergente et certaines options intéressantes sont à la disposition de l'utilisateur du service afin d'éviter, dès le départ, tout problème. Il en résulte que le contrat d'interface est le principal élément permettant d'orchestrer la composition et la consommation des données.

Figure 1 Les schémas intégrés dans un contrat d'interface constituent habituellement la plus grande dépendance du client.



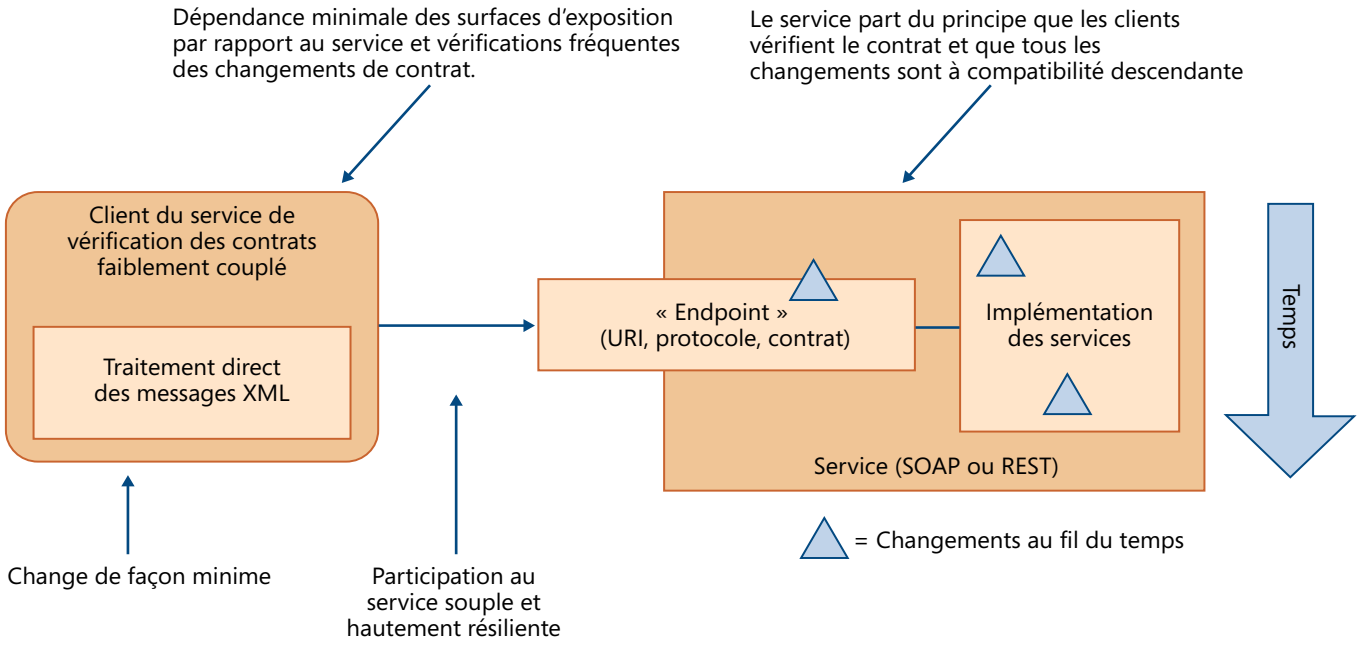
Le défaut d'adaptation des « impédances » pour la mise en place d'une abstraction des données interrompt la composition et la consommation des données. Le point physique auquel l'intégration des données a lieu est, de plus en plus, à l'extérieur du contrôle classique des bases de données et les bibliothèques d'accès de données convertissent tout en une abstraction de données unifiées. Les données récupérées des différents services externes et dans différents formats sont en conflit dans le navigateur, dans les clients « front-end » ou côté serveur, ou dans le code interne hors de la base de données. Les structures de données de ces services sous-jacents sont, selon l'application logicielle, des objets natifs, XML, JSON et documents ou fragments SOA, voire des données texte, binaires et multimédia. Bien que la problématique d'adaptation des impédances relative à l'abstraction des données (« impedance mismatch ») soit un problème connu depuis longtemps en ce qui concerne les logiciels, elle a été exacerbée par la prolifération de modèles destinés à la représentation des informations. Lors de la consommation et de la composition de ces structures de données dans des représentations agrégées utilisables, puis leur retour vers leurs services d'origine, plusieurs problèmes majeurs surviennent.

> « LORSQUE L'ON UTILISE LES STRUCTURES DE DONNÉES D'UNE SOURCE OU D'UN SERVICE EXTERNE, TEL QU'UN SERVICE WEB, ELLES DOIVENT ÊTRE VALIDÉES PAR RAPPORT AU CONTRAT D'INTERFACE FOURNI PAR LE SERVICE SOURCE. »

- *Variabilité des formats de données*. Les principaux formats des structures de données sous-jacentes sont souvent hautement incompatibles et de bons moyens pour traiter localement tous les formats potentiels ne sont habituellement pas disponibles. Les formats extrêmement complexes requérant des piles de protocoles sophistiqués pour être traités, tels que les services Web de haut niveau, mettant en œuvre des règles WS-POLICY, sont particulièrement problématiques.
- *Identité des données*. Les identificateurs d'objets/de données clés et uniques ne sont pas souvent dans des formats communs et ne font pas partie des moyens disponibles pour les vérifier et veiller à leur application homogène dans les différents modèles d'abstraction.
- *Relations des données*. Établir des relations et des associations entre les structures de données de formats très différents peut se révéler problématique, sur le plan de l'efficacité et des performances, pour un certain nombre de raisons. La conversion de toutes les données dans un format commun peut être coûteuse, tant en termes de maintenance que de coûts d'exécution. Ceci introduit également le problème du maintien de la provenance des données mélangées et de leur extraction une fois modifiées. La conservation des structures de données dans les formats d'origine suppose de bons mécanismes pour les composer, les lier et les manipuler.
- *Provenance des données*. Le maintien du service source d'origine d'un segment de données est essentiel pour conserver la validité du contexte des conversations en cours et pour les changements des données en particulier, tels que les mises à jour, les ajouts et les suppressions.
- *Intégrité des données*. Il est important de modifier les données source en veillant à ce que les changements soient valides conformément au contrat d'interface du service sous-jacent et sans enfreindre les définitions de schémas. Avec des documents XML bien définis et reposant sur des schémas XML riches, ces validations sont une tâche aisée. Toutefois, des modèles de programmation plus légers, tels que JSON, ont rarement des schémas lisibles par une machine. Ces types de formats de données, bien que souvent uniquement disponibles à partir d'un service fédéré, constituent le risque le plus élevé de problèmes d'intégrité, car leurs règles de changement et de manipulation ne sont pas aussi bien définies que pour les autres formats.
- *Conversion des données*. Souvent, il n'existe pas de conversion clairement spécifiée entre les formats de données ou, pire, les multiples choix proposés modifient les données de façons subtiles. Les données les plus fréquemment affectées sont des représentations numériques, mais ces conversions peuvent également affecter, presque tout aussi souvent une source de données dans plusieurs jeux de caractères, des données audio/vidéo et des données GIS.

Surfaces d'exposition des contrats d'interface de grande envergure. Des structures d'interface et des schémas extrêmement complexes conduisent à l'un des principaux problèmes dans le domaine de l'architecture des données de systèmes distribués. L'expérience a démontré à maintes reprises que la simplicité est garante de plus de fiabilité et d'une qualité supérieure. Pourtant, les contrats d'interface des services distribués contiennent souvent des schémas XML et des définitions WSDL complexes. La probabilité de changements, opérés ou non par inadvertance, s'ac-

Figure 2 Vérification de l'exécution du contrat d'interface des services Web



croît avec la complexité d'un contrat d'interface et de ses schémas intégrés. Alors que les approches XML et REST des services tendent à encourager la simplicité souhaitée, certains domaines d'applications ne sont parfois pas « simples » et ces modèles de services sont sujets à des problèmes supplémentaires en raison de l'absence de normes relatives au contrat d'interface. Résultat : la probabilité que les changements effectués sur le contrat sans qu'un fournisseur distant n'en soit avisé augmente suivant la taille du contrat, car les changements opérés sur les grands schémas sont plus susceptibles de ne pas être détectés que ceux effectués sur un schéma plus petit.

Changements de version implicites. Même dans les environnements bien contrôlés, les changements effectués sur les services et leurs contrats d'interface peuvent être effectués sans en avertir toutes les parties concernées. En apportant des changements au fonctionnement actuel des services de données, ce modèle n'en sera que plus répandu dans les environnements hautement fédérés d'aujourd'hui et de demain. Dans le cas contraire, les services dont vous dépendez changeront sans que vous n'en ayez connaissance ou que vous n'ayez donné votre avis au préalable et vous devrez vous y résigner. Par conséquent, le développement de stratégies conscientes visant à détecter les changements de versions et à les gérer efficacement est essentiel pour conserver la qualité des fonctionnalités et des données d'un « mashup » ou d'une application composite.

Modèles de composition et de consommation de données

Les « patterns » décrits dans cet article visent à présenter une position simple sur la modélisation des données et l'architecture des applications. Ces remarques résultent de l'observation du monde dynamique des « mashups » et des applications composites qui se multiplient sur le Web depuis déjà plusieurs années. Les approches souples et minimalistes utilisées, notamment, par les « mashups » constituent une source d'inspiration mais ne suffisent souvent pas à créer un logiciel de qualité. Ces modèles visent à épouser l'esprit des « mashups », à mettre ces derniers en contexte avec de bonnes pratiques d'ingénierie logicielle et à ouvrir un débat avec les gens de la profession sur ces méthodes minimales de connexion des services et des données.

« Pattern 1 » : dépendance minimale des surfaces d'exposition sur les contrats d'interface. Ce modèle est l'aspect visible de la maxime bien connue dans la conception logicielle : « Soyez libéral dans ce que vous acceptez, conservateur dans ce que vous envoyez ». La plupart des boîtes à outils de services Web, des frameworks de base de données relationnelles et des bibliothèques multimédia encouragent les dépendances sur trop de points du contrat d'interface (souvent sur l'ensemble du contrat), indépendamment de ce dont dépend le logiciel. Dans un très grand nombre de scénarios de composition et de consommation, une petite dépendance des surfaces d'exposition est tout à fait suffisante. Une dépendance de tout le contrat induit une grande fragilité, car les changements qui n'ont aucune relation avec les données de la structure dont elles dépendent, seront responsables d'une rupture sur le client. Alors que certaines boîtes à outils de consommation des données ne tiennent déjà plus compte de cette dépendance, il y a souvent peu de contrôle sur les parties qu'elles prennent en charge.

Pour de nombreuses applications, seules sont appropriées les dépendances directes des éléments de données internes requis. Toutes les autres dépendances doivent être activement éliminées de la relation de dépendance au sein des données sources. Le point crucial dans ce cas est que les changements effectués dans le contrat d'interface, qui ne sont pas importants pour le consommateur des données, ne doivent pas en empêcher l'utilisation. Le contraire doit également être vrai : les changements effectués dans le contrat qui sont importants doivent immédiatement devenir apparents pour empêcher un comportement et une utilisation incorrects des données. Les exemples de dépendances minimales comprennent : XML (les chemins d'accès clés des schémas aux éléments de données), les données relationnelles (uniquement les tableaux, les types, les colonnes et les index utilisés par le client) et les données multimédia (uniquement les parties de la structure de média requises, telles que les canaux spécifiques, la bande passante, la portion d'image ou le segment vidéo).

« Pattern 2 » : vérification des changements du contrat en cours d'exécution. Les services dont dépendent les « mashups » et les applications pour les données sont sujets, à tout moment, à des changements inattendus. Ces derniers peuvent être dus au déplacement du « endpoint » dans un nouvel emplacement, à l'arrêt de l'utilisation d'un protocole précédemment supporté, à des changements effectués sur le schéma sous-jacent, voire même simplement à des changements délibérés opérés en bloc sur l'interface ou à des défaillances des services internes. La raison conduit à la nécessité de vérifier le contrat afin d'en détecter les changements. Elle requiert également de pouvoir traiter correctement de nombreux changements de contrats, car ils peuvent très bien ne pas affecter la partie du contrat qui vous importe.

Il existe, en fait, deux types de vérifications de contrat en cours d'exécution. La première consiste à *vérifier la spécification du contrat*, s'il y en a une. Il s'agit souvent du WSDL ou d'autres métadonnées officiellement publiées en conjonction avec le service lui-même. Elle peut être vérifiée facilement et rapidement à l'aide d'une variété de techniques de programmation, incluant les requêtes XPath pour les schémas XML ou d'autres techniques relativement légères. Les vérifications de contrats codés en dur dans les langages de programmation traditionnels ne sont pas si faciles à implémenter ou à modifier et doivent constituer un choix secondaire.

La seconde vérification de contrat consiste à valider le contrat par rapport aux instances de données fournies par le service. Les contrats d'interface ont souvent leur propre impédance relative à l'abstraction des données avec leurs mécanismes de livraison, et la fréquence des incohérences entre les données fournies et le contrat d'interface, même avec des boîtes à outils de haute qualité, peut se révéler surprenante.

La fréquence à laquelle vérifier la validation de contrat constitue le plus grand dilemme présenté par ce modèle. La vérification du contrat et des données d'instance à l'aide de chaque donnée récupérée à partir de sa source peut prendre du temps et se révèle habituellement inutile. À terme, déterminer la fréquence des vérifications de contrat dépend largement des exigences de l'application ainsi que de la nature des données et de leur tolérance à l'inexactitude et à un comportement incorrect. Pour de nombreuses applications, une vérification synchrone peut même ne pas être une option et il peut se révéler judicieux de mettre en place un processus d'arrière-plan périodique et approprié pour identifier les changements de contrat qui se produiront inévitablement.

« Pattern 3 » : alignement des structures de données sur un format commun d'abstraction. Dans les faits, l'hétérogénéité des structures et des formats de données avec lesquels les « mashups » et applications composites doivent fonctionner ne cesse de s'accroître. Les logiciels peuvent les manipuler dans leurs formats de données d'origine, ce qui signifie perdre des opportunités de maintenir des relations entre elles et d'appliquer les règles métier ou ils peuvent tous les convertir dans une abstraction commune. Cette conversion est l'approche que les bibliothèques de données comme ADO.NET utilisent avec leurs « Datasets » et que XML utilise avec d'autres sources de données non-XML. Englober les différences de données sources, par la conversion dans et hors d'une abstraction de données commune fournissant un modèle unifié unique peut sembler approprié à plusieurs titres. Premièrement, les relations entre les différentes sources de données sous-jacentes peuvent être vérifiées et appliquées lorsque les données sont manipulées et modifiées. Deuxièmement, les vues sur les données peuvent tirer parti du mécanisme d'abstraction en rendant secondaire la création de mécanismes MVC (Model-View-Controller) spécifiques et l'utilisation des bibliothèques et des frameworks existants pouvant offrir cette abstraction.

Ce modèle n'est pas adapté à tous les ensembles de structures de données formatées de façon hétérogène et peut se révéler inutile pour certains formats de données dont les niveaux d'impédance relative à l'abstraction des données sont très élevés, les données de type image avec des données d'un autre type, par exemple. De plus, la transformation et la conversion des données dans et hors du format commun ont un coût potentiellement élevé. Pour de nombreuses applications, toutefois, ce coût est tout à fait acceptable.

Dans certains environnements de consommation de données, tels que le navigateur Web, les possibilités de conserver un format de données commun sont limitées et JSON et le XML DOM tendent à être assez populaires pour les solutions basées sur le navigateur. Côté serveur, les choix sont beaucoup plus nombreux mais dépendent souvent de la plateforme. Les structures XML, les bibliothèques d'Object Relational Mapping comme Hibernate, les bases de données relationnelles, voire même les graphes d'objets natifs font souvent, en fonction de l'application, d'excellents modèles d'abstraction commune. Mais la nature très différente des données hiérarchiques, telles que XML et les graphes d'objets, est l'un des problèmes d'adaptation d'impédance classique en informatique et une attention particulière doit être apportée lors de l'utilisation de ce modèle.

Résultat : si les performances ne sont pas indispensables et que les schémas et formats de données sous-jacents sont souples, ce modèle peut se révéler très puissant lors de l'utilisation de sources de données fédérées. L'inconvénient est que l'approche d'abstraction commune peut certainement impliquer plus de maintenance et générer de la fragilité car le mappage et les métadonnées doivent être maintenues.

« Pattern 4 » : structures d'origine modifiées avec MVC (Model-View-Controller). La conversion de toutes les données sources dans un format commun ne se révèle pas une option dans de nombreuses situations car 1) le surcoût du traitement est excessif dans la mesure où un grand nombre de données est susceptible de ne pas être utilisé ou 2) leur duplication dans l'environnement local peut être prohibitif en termes de ressources. Cela peut être aussi parce qu'il n'existe pas de format commun adapté à tous les types de données sous-jacents. Dans ce cas, la création d'un MVC qui modifie l'accès, la traduction et l'intégrité des données avec les structures d'origine sous-jacentes peut fournir les meilleures options pour les performances et le stockage des données.

> « INTERNET DEVIENT LE FOURNISSEUR LE PLUS IMPORTANT DE DONNÉES HAUTEMENT FÉDÉRÉES, UNE SOURCE QUI CONTINUERA À S'ACCROÎTRE DANS LES PROCHAINES ANNÉES »

MVC est un modèle de conception puissant à part entière qui s'est souvent révélé une excellente stratégie de séparation des problèmes dans les logiciels d'applications. Son utilisation est particulièrement indiquée lorsqu'il existe plusieurs modèles de données sous-jacents dans une application donnée. Une bonne conception logicielle implique qu'une vue unifiée des données sources offre une interface unique, propre et cohérente qui permet facilement de visualiser les données sous-jacentes, d'interagir avec elles et de les modifier. L'accès à ces dernières à l'aide de l'approche MVC est également relativement efficace, car seules les données requises doivent être traitées pour répondre à la plupart des requêtes.

Bien qu'il existe de nombreux avantages à utiliser le MVC, les inconvénients sont toutefois similaires à ceux du modèle 3, à savoir que la maintenance du code MVC peut être énorme. Sans l'ombre d'un doute, il existe un nombre croissant de bibliothèques prêtes à utiliser qui peuvent aider les concepteurs de logiciels à créer un modèle MVC sur le client et sur le serveur. Soyez attentif, toutefois : le code de mappage est fragile et fastidieux.

« Pattern 5 » : accès direct aux structures de données natives. Pour de nombreuses applications, notamment celles qui sont simples et basées sur un navigateur, la conversion des données sources en des formats communs ou la création d'architectures MVC sophistiquées n'est pas une bonne option. L'accès direct aux données est plus logique et la décision tombe souvent sous le sens, lorsque l'on dispose des bibliothèques. De plus, comme mentionné précédemment, la *simplicité* est souvent plus indiquée et garante d'une qualité supérieure, car il y a moins de remise en cause de l'existant ou de maintenance.

Dans ce modèle, qui fonctionne au mieux avec moins de données hautement structurées, les structures de données natives sont utilisées directement sans intermédiaire ou conversion de données, ce qui signifie que les magasins de données aux formats texte, XML et JSON, entre autres, sont manipulés en mode natif. L'inconvénient, bien sûr, est que vous ne pouvez pas avoir recours aux moyens que les bibliothèques d'abstraction de données peuvent vous donner pour appliquer les changements relatifs à l'intégrité ou au suivi. Toutefois, ce modèle est celui qui requiert souvent le moins de traitement ou d'étude des bibliothèques tiers. Il peut être facilement développé et se révèle également plutôt rapide puisqu'il n'y a pas de conversion ou de nécessité de traverser des couches d'accès aux données.

«Pattern 6 » : la modification des données est atomique. Un nombre plus important de vues très élaborées sur les données et de services sophistiqués rend obligatoires des propriétés connues sous le sigle ACID (pour atomicité, concurrence, isolation et durabilité). Habituellement attribuée aux systèmes de bases de données, ACID constitue, en règle générale, une

Tableau 1 : abstractions de données

Abstraction	Norme de contrat	Avantages	Inconvénients
Texte, JSON, binaire	Informel, textuel	Relativement efficace	Pas autodescriptif, pas réellement efficace
Objets natifs	Définition des classes	Comportement et données unifiés, encapsulation, abstraction de haut niveau et composition	Requiert la conversion de la plupart des données en objets, nécessitant une technique de mappage
XML	Schémas XML (XSD), Relax NG, WSDL et de nombreux autres	Autodescriptif, description de schéma riche et compatibilité descendante extensible sans fragmentation	Très inefficace en termes de taille, pas de possibilité de distribuer le comportement et les descriptions de schémas sont limitées même avec XSD
Images	Spécifications pour JPEG, TIFF, GIF, BMP, PNG et de nombreux autres formats	N/A	N/A
Audio	Spécifications pour WAV, WMA, MP3 et AAC	N/A	N/A
Vidéo	Spécifications pour AVI, QuickTime, MPEG et WMF	N/A	N/A

bonne pratique pour presque tous les systèmes d'accès aux données concurrents. Malheureusement, presque personne encore dans le monde des services Web et des « mashups » n'intègre la notion de transaction dans leurs protocoles, ce qui conférerait nombre des propriétés ACID à la modification des données. Loin d'ignorer le problème, les développeurs de « mashups » et d'applications composites doivent être tout à fait conscients qu'ils marchent sur une corde raide lorsqu'ils utilisent leurs données.

Cette prise de conscience est à l'origine de problèmes significatifs avec les scénarios de modification de données complexes, à savoir que les récupérations et stockages de données dépendant d'une conversation étendue avec des services sous-jacents ne sont pas protégés par la même frontière de transaction et les mêmes propriétés ACID correspondantes. Le code logiciel doit s'attendre à ce que des problèmes d'intégrité des données aient lieu, en particulier dans une conversation étendue qui risque d'échouer ou de ne jamais arriver à terme. Éviter des conversations de longue durée est une option. Rendre, autant que possible, une opération logicielle dépendante de services sous-jacents dans l'interaction atomique individuelle est une autre solution. Chaque étape de l'interaction constitue une validation unitaire visible qui permet au logiciel d'offrir à ses utilisateurs des options claires lorsqu'une conversation avec un service de données du fournisseur rencontre une défaillance. Ces deux options permettent aux développeurs de logiciels de respecter, en règle générale, les propriétés de transaction ACID.

Retour à la simplicité

Internet devient le fournisseur le plus important de données hautement fédérées, une source qui continuera à s'accroître dans les prochaines années. Alors que XML et les formats de données simples seront probablement la structure de données utilisée par la plupart des logiciels dans un futur proche, d'autres formats au parcours plus hasardeux sont en train de faire leur apparition. Il s'agit des optimisations particulièrement nécessaires dans XML, telles que le XML binaire, les avancées dans les microformats, de nouveaux codecs multimédia qui transformeront tout un paysage audiovisuel ou encore les protocoles de transport entièrement nouveaux, comme Bittorrent, qui rendront, c'est le moins que l'on puisse dire, nombre des modèles évoqués ici problématiques.

Un retour à la simplicité en matière de conception de données est à nouveau en vogue avec, pour exemple, le regain d'intérêt pour des formats tels que JSON, les microformats et les langages dynamiques comme PHP et Ruby. Cette simplicité de la conception peut rendre les données plus malléables et plus faciles à connecter entre elles. Elle présente également moins de difficultés pour écrire des logiciels visant à les gérer. Bien que les modèles présentés ici soient ceux qui aboutissent à une composition et une consommation des données à la fois moins fragiles, plus faiblement couplées et à haute intégrité, l'histoire ne fait que commencer... Internet portant de moins en moins sur les pages Web visuelles et de plus en plus sur les services ainsi que sur les données et contenus purs, devenir un consommateur et un fournisseur souple de l'écosystème des informations sera un facteur de succès toujours plus important. •

Ressources

microformats
http://microformats.org

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

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

Hewlett-Packard Development Company
Rapports techniques
« Rethinking the Java SOAP Stack », Steve Loughran and Edmund Smith
www.hpl.hp.com/techreports/2005/HPL-2005-83.html

À propos de l'auteur

Dion Hinchcliffe est le responsable technologique de Sphere of Influence Inc.

Le modèle « Nordic » de bases de données objets/relationnelles

par Paul Nielsen

Résumé

Le nouveau modèle de bases de données objets/relationnelles surnommé « Nordic » n'est pas, comme de nombreux outils, une solution adaptée à tous les types de problèmes que rencontrent les bases de données. Ce modèle hybride peut toutefois offrir des performances élevées en termes de puissance, de flexibilité, d'efficacité, voire d'intégrité des données par rapport aux modèles relationnels traditionnels, notamment pour les bases de données qui comportent un héritage, des explorations de données créatives, des interactions souples entre classes et des contraintes de workflow. Découvrez quelques-unes des innovations possibles lorsque la technologie orientée objet est modélisée à l'aide des bases de données relationnelles aujourd'hui matures.

Les différences entre le développement orienté objet et le modèle de base de données relationnelle créent une tension communément appelée *décalage d'impédance objet/relationnel*, l'héritage ne se traduisant pas bien dans un schéma relationnel. Ce décalage technique d'impédance est aggravé par la déconnexion culturelle entre les codeurs d'applications et les administrateurs de bases de données (DBA). Souvent, aucune des parties ne comprend clairement, ni ne respecte le langage de l'autre. Pour piquer au vif un DBA, il faut se référer à la base de données comme l'« utilitaire d'objets persistants ». Cette relation est regrettable, car chaque partie apporte ses avantages au problème de l'architecture des données.

Le modèle orienté objet est, à bien des égards, supérieur au modèle relationnel. On pourra, par exemple, utiliser la structure supertype/sous-type relationnelle pour concevoir l'inhérence entre classes, mais le modèle orienté objet est une solution plus élégante. D'autre part, la majeure partie du code d'application est orienté objet et une base de données orientée objet entrera en interface avec l'application plus facilement qu'une base de données relationnelle.

Même si la technologie orientée objet est à un stade avancé en termes de modélisation de la réalité, il ne faut pas négliger les avantages notables qu'apporte la technologie relationnelle. Les moteurs de bases de données relationnelles offrent des qualités de performance et d'évolutivité ainsi que des options de haute disponibilité. Ils ont aussi l'appui financier qui assure à cette plateforme un bel avenir. Nous maîtrisons la technologie relationnelle et les bases de données de ce type offrent des outils de requête et de création de rapports plus puissants que les bases de données objets. Les rares entreprises qui proposent des bases de données purement objets n'ont tout simplement pas les ressources pour concurrencer Microsoft, Oracle ou IBM.

Le problème du décalage d'impédance entre le modèle objet et le modèle relationnel consiste donc à trouver comment adopter l'élégance des technologies orientées objet tout en conservant la puissance, la flexibilité et la stabilité à long terme d'un moteur de base de données relationnelle mature. Puisque les programmeurs d'applications sont généralement les plus intéressés par la résolution de cette question et que par nature, on ait tendance à se tourner vers les solutions les plus familières pour résoudre ses problèmes, on comprend aisément pourquoi la plupart des solutions sont mises en œuvre dans une couche de mappage entre la base de données et le code d'application, qui traduit les objets en tables relationnelles pour les objets persistants.

La proposition « Nordic »

Je suis d'avis qu'un modèle relationnel, conçu pour émuler des fonctionnalités orientées objet, peut présenter d'excellentes performances dans les moteurs de bases de données relationnelles actuels et que la manipulation de l'héritage des classes et des associations complexes directement dans la base de données, c'est-à-dire proche des données, est, en fait, très efficace. Cette efficacité n'a pas toujours eu cours. J'ai été recruté pour optimiser un modèle de base de données objets/relationnelle (O/R) intensive Transact-SQL (T-SQL) mis en œuvre avec SQL Server 6.5 et j'ai échoué. Le développement du modèle « Nordic » de bases de données objets/relationnelles a impliqué une année d'itérations, un schéma de métadonnées simplifié et le T-SQL mature de SQL Server.

« MÊME SI LA TECHNOLOGIE ORIENTÉE OBJET EST À UN STADE AVANCÉ DE LA MODÉLISATION DE LA RÉALITÉ, IL NE FAUT PAS NÉGLIGER LES AVANTAGES NOTABLES QU'APPORTE LA TECHNOLOGIE RELATIONNELLE. »

Comme pour tout projet de base de données, une couche d'abstraction de données strictement appliquée, qui encapsule la base de données, est indispensable pour l'extensibilité à long terme. Pour une base de données hybride O/R, la couche d'abstraction des données offre également la façade pour les fonctionnalités orientées objet. Derrière le code de façade se trouvent le schéma des métadonnées et la génération du code pour les classes, objets et associations (voir figure 1). Plusieurs décisions importantes de conception doivent être prises lors de l'implémentation de cette solution.

Gestion des classes. Dans le schéma relationnel, les métadonnées de classe et d'attribut sont facilement modélisées à l'aide d'une relation un-à-plusieurs commune. La relation superclasse/sous-classe est modélisée en tant qu'arborescence hiérarchique à l'aide du modèle de listes adjacentes, plus courant, ou bien du modèle du chemin matérialisé, plus efficace.

En naviguant vers le haut et le bas dans la hiérarchie des classes avec des fonctions définies par l'utilisateur, les requêtes SQL peuvent se joindre facilement aux classes descendantes ou ancêtres d'une classe. Ces fonctions définies par l'utilisateur sont exploitées à travers la couche de façade. Par exemple, lorsque l'on sélectionne toutes les propriétés d'une classe, joindre la fonction définie par l'utilisateur superclasses() renvoie toutes les superclasses

et la requête peut alors sélectionner l'ensemble des propriétés d'une classe donnée, notamment celles héritées des superclasses.

Dans le cadre d'un projet impliquant des objets persistants, le terme *polymorphisme* se rapporte à la capacité du mode de sélection à récupérer non seulement les objets de la classe active, mais aussi tous les objets des sous-classes. À titre d'exemple, sélectionner tous les contacts devrait permettre de sélectionner les objets de la classe contacts, mais aussi ceux des sous-classes du client et du client principal. Une fonction définie par l'utilisateur, qui renvoie une variable de table de toutes les sous-classes d'une classe donnée, rend l'écriture de cette requête et de cette procédure stockée à la fois efficace et réutilisable.

Gestion des objets. La modélisation des objets sera optimisée par le recours à une table d'objets unique qui stocke les données courantes de l'objet, telles que l'ID d'objet unique, la classe de l'objet, les données d'audit, et quelques attributs de recherche communs à la quasi-totalité des classes, comme le nom ou la date. D'autres attributs sont stockés dans des tables de classes personnalisées qui utilisent une clé étrangère d'ID d'objet pour relier les attributs personnalisés à la table d'objets. La procédure createclass et les autres procédures de gestion des classes stockées dans la façade exécutent le code DDL (langage de définition de données) pour créer ou modifier les tables de classes personnalisées et générer le code de façade personnalisé afin de sélectionner, d'insérer et de mettre à jour les objets.

La principale décision de conception pour la modélisation du stockage des objets réside dans la représentation des données d'attributs personnalisés. Trois méthodes sont possibles : le *modèle valeur-paire*, les tables de classes personnalisées concrètes et les tables de classes personnalisées en cascade. Le modèle valeur-paire, également appelé *modèle générique*, utilise une structure en losange constituée d'une classe, d'une propriété, d'un objet et d'une valeur. La table des valeurs comporte une seule colonne pour stocker toutes les valeurs. Cette longue table étroite compte une ligne par attribut. Dix millions d'objets avec 15 attributs utiliseraient 150 millions de lignes dans la table des valeurs. SQL Server a tout à fait les capacités de travailler avec de vastes tables, là n'est pas le problème. Ce modèle semble offrir la meilleure flexibilité, car des attributs peuvent être ajoutés sans modifier le schéma relationnel. Mais il ne permet pas (ou, au mieux, d'une façon peu pratique) le typage des données et les requêtes sont difficiles à effectuer avec SQL.

Une table pour chaque classe

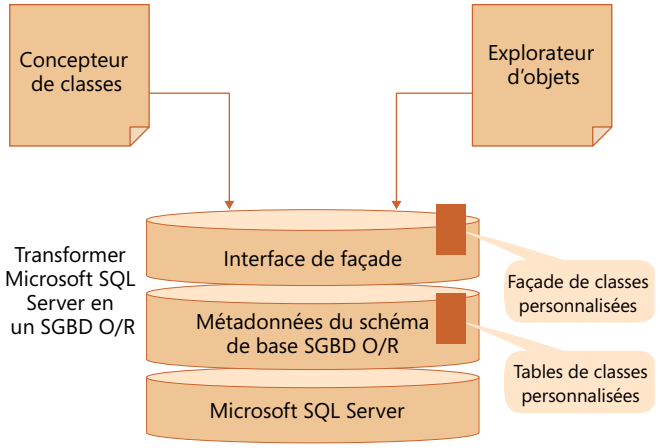
Le modèle de *classes personnalisées concrètes* utilise une table pour chaque classe avec des colonnes pour chacun des attributs personnalisés, y compris les attributs hérités non abstraits. Un objet existe dans deux tables uniquement (la table d'objets et la table de classes personnalisées concrètes) tandis que les attributs sont répliqués dans les tables de classes personnalisées concrètes de chaque sous-classe. Par conséquent, si la classe des animaux comporte un attribut de date de naissance et si la sous-classe des mammifères comporte un attribut de sexe (puisque certains animaux n'ont pas de sexe), alors la table de classes personnalisées des mammifères inclut les colonnes ID d'objet, date de naissance et sexe.

Ce modèle a un avantage : la sélection de tous les attributs pour une classe donnée exige uniquement de joindre la table des métadonnées objets à une seule table de classes personnalisées. L'inconvénient est la mise en œuvre du polymorphisme ; la sélection de tous les animaux exige d'exécuter une union de chaque sous-classe, et d'éliminer les attributs des sous-classes ou d'ajouter les attributs de substitution des superclasses de sorte que toutes les sélections dans l'union ont des colonnes compatibles.

Les tables de classes personnalisées concrètes, qui sont une amélioration du modèle valeur-paire, utilisent une colonne relationnelle pour chaque attribut afin que le typage des données d'attributs puisse facilement implémenter les types de données natives de la base de données relationnelle hôte.

La troisième option implémentée dans le modèle « Nordic » de bases de données objets/relationnelles, les tables de *classes personnalisées en cascade*, utilise une table pour chaque classe, comme la solution de

Figure 1 Le modèle hybride O/R utilise une façade pour encapsuler la fonctionnalité orientée objet, qui est exécutée dans un schéma de base de données relationnelle.



classes concrètes. Toutefois, plutôt que de répliquer les attributs, chacun d'entre eux n'est représenté qu'une seule fois dans sa propre classe et chaque objet est représenté une seule fois dans chaque classe en cascade. Si l'on se base sur l'exemple des mammifères et des animaux, la table des animaux contient l'ID d'objet et la date de naissance et la table des mammifères contient l'ID d'objet et le sexe. Une instance de l'objet mammifère est stockée dans les métadonnées objets, dans la table des animaux et dans la table des mammifères. Dans cette option, le polymorphisme est très simple, mais un plus grand nombre de jointures est nécessaire pour sélectionner tous les attributs des sous-classes. Puisque SQL Server est optimisé pour les jointures, les tables de classes personnalisées en cascade fonctionnent très bien. Comme pour les tables de classes personnalisées concrètes, le typage des données fort est pris en charge par la base de données hôte.

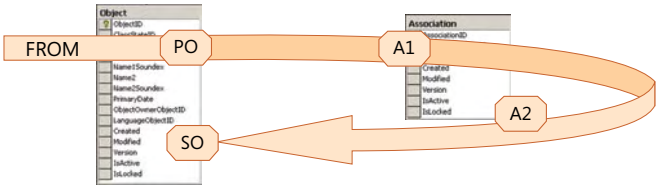
Gestion des associations. Dans la technologie orientée objet, les associations sont très similaires aux contraintes de clés étrangères que l'on trouve dans la technologie des bases de données relationnelles. Il est certainement possible de définir des associations au sein d'un modèle hybride O/R en ajoutant des attributs de clé étrangère et en affectant des contraintes d'intégrité référentielle déclarative relationnelle aux tables de classes personnalisées.

Toutefois, stocker chaque objet dans une table unique offre plusieurs alternatives intéressantes pour la modélisation d'associations. Tandis qu'une base de données relationnelle normalisée peut contenir des dizaines de clés étrangères, chacune avec une table de clés étrangères et une colonne distinctes et chacune référençant une clé primaire différente, une table d'associations hybride O/R ne doit référencer qu'une seule table. Chaque relation de clé étrangère dans la base de données peut être généralisée en une clé étrangère association.objectid to object.objectid.

La table d'associations peut être conçue en tant que liste par paires (id_ObjetA, id_ObjetB), mais cette conception est trop restrictive pour les collections complexes et les requêtes doivent identifier objetA et objetB. L'alternative la plus souple fait appel à une liste d'objets-associations constituée d'un seul ID d'objet pour référencer l'objet et d'un ID d'association pour regrouper les objets associés. Une colonne ID de type d'association (associationtypeid) peut référencer les métadonnées descriptives de l'association et peut éventuellement fournir les contraintes de l'association.

Une table unique pour chaque objet et une autre table pour chaque association ne ressemblent en rien à un schéma normalisé, ce qui est le cas. La structure physique n'est pourtant pas un problème. SQL Server excelle dans le domaine des tables longues et étroites et ce modèle, qui se prête bien au paramétrage d'index cluster et non cluster, produit de hautes performances.

Figure 2 Une requête générique à trois jointures localise tous les objets associés, quelle que soit la 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 SO.ObjectID = A2.ObjectID
```

Trouver toutes les associations

Pour l'architecte de bases de données, le modèle de la liste d'objets-associations offre d'étonnantes possibilités. Tout d'abord, la jointure d'un objet avec un nombre n d'autres objets de toute classe fait appel à la même requête (voir figure 2). L'ajout de tables supplémentaires à une requête relationnelle ajoute des jointures n-1, mais la liste d'objets-associations utilise toujours les trois mêmes jointures, indépendamment du nombre de classes concernées. Selon l'application, ce style d'objets liés peut être beaucoup plus performant qu'un modèle relationnel modélisé. À mesure que de nouvelles classes sont ajoutées au modèle de données ou à l'association, elles sont automatiquement incluses dans les requêtes « trouver toutes les associations » sans modifier le code existant.

Trouver toutes les associations entre classes ou tous les objets qui ne participent pas à une quelconque association avec une autre classe, ou d'autres requêtes d'exploration des données, créatives et pourtant puissantes, sont toutes des requêtes reposant sur un jeu, qui sont triviales et réutilisables. Les fonctions SQL Server définies par l'utilisateur peuvent encapsuler le travail avec des associations ainsi que les nombreuses combinaisons logiques d'objets qui participent ou non à des associations.

Il est également possible de créer des collections complexes avec une liste d'objets-associations. Par exemple, une collection de salles de classes peut inclure une salle, un ou plusieurs enseignants, un ou plusieurs bureaux, le programme et un ou plusieurs élèves tels que définis dans les métadonnées de l'association.

Les associations généralisées ouvrent de plus larges possibilités. Les pages Web sont elles aussi liées à l'aide d'une méthode généralisée ; chaque lien hypertexte utilise une balise d'ancrage et une URL. Il s'agit en fait d'une liste d'objets-associations incorporée au code HTML. Le mappage graphique d'une page Web et l'affichage de la navigation entre les pages Web sont triviaux, quel que soit le contenu de la page Web. De la même manière, il est banal de basculer entre les tables d'objets et d'associations et de localiser immédiatement des objets associés par plusieurs degrés de séparation, quelle que soit la classe.

Je suis actuellement en train de développer une base de données pour la gestion de parrainages d'enfants afin d'aider des organisations caritatives qui luttent contre la pauvreté. À l'aide de la liste d'objets-associations, une seule fonction définie par l'utilisateur, qui trouve des associations pour n'importe quel objet, est capable de trouver que Joe parraine un enfant au Pérou, qu'il doit assister à une conférence sur la pauvreté, qu'il a envoyé trois lettres à l'enfant, qu'il lui a envoyé un cadeau l'année dernière et qu'il s'est renseigné à propos d'un enfant en Russie.

En étendant la liste d'associations-objets pour obtenir des degrés de séparation multiples, on constate que Greg doit se rendre dans la ville péruvienne où vit l'enfant que parraine Joe, que 14 autres personnes ont participé à la conférence sur la pauvreté à laquelle assistait Joe et que trois de ces personnes parrainent des enfants au Pérou. Cette flexibilité de requête pour tout objet est impossible avec un modèle relationnel.

État du workflow d'objets

La généralisation de la liste d'objets-associations se prête bien à une autre innovation dans le domaine des bases de donnes : l'intégration de l'état du workflow d'objets dans la base de données. Certes, l'état du workflow ne s'applique pas à toutes les classes, mais pour certaines d'entre elles, le workflow est une dimension de l'intégrité des données qui manque au modèle de base de données relationnelle.

Pour une commande, un workflow typique peut être : un panier, une confirmation de commande, une confirmation de paiement, un inventaire alloué, en cours de traitement, prête pour l'expédition ou expédiée. Une clé étrangère relationnelle astreint uniquement la table secondaire à référencer une valeur de clé primaire valide. Avec une base de données relationnelle, une ligne descriptive peut être créée pour référencer la *commande*, indépendamment de l'état de son workflow. Un code personnalisé doit valider que la commande a exécuté certaines étapes avant l'expédition.

Avec des états de workflow pouvant être hérités et définis comme partie intégrante des métadonnées de la classe, les métadonnées de l'association peuvent restreindre les objets à la classe et à l'état du workflow, intégrant ainsi le workflow dans les données d'objet.

> « AVEC DES ÉTATS DE WORKFLOW POUVANT ÊTRE HÉRITÉS ET DÉFINIS COMME PARTIE INTÉGRANTE DES MÉTADONNÉES DE LA CLASSE, LES MÉTADONNÉES DE L'ASSOCIATION PEUVENT RESTREINDRE LES OBJETS À LA CLASSE ET L'ÉTAT DU WORKFLOW, INTÉGRANT AINSI LE WORKFLOW DANS LES DONNÉES D'OBJET. »

J'ai présenté quelques-unes des innovations possibles lorsque la technologie orientée objet est modélisée à l'aide des bases de données relationnelles aujourd'hui matures. Comme pour tout outil, le modèle « Nordic » de bases de données objets/relationnelles n'est pas une solution indiquée pour toutes les bases de données. Il peut, néanmoins, offrir des performances supérieures en termes de puissance, de flexibilité, d'efficacité, voire d'intégrité des données par rapport aux modèles relationnels traditionnels pour les bases de données qui bénéficient d'un héritage, d'explorations de données créatives, d'interactions souples entre classes et de contraintes de workflow. •

À propos de l'auteur

Paul Nielsen qui est MVP SQL Server, est l'auteur de la série *SQL Server Bible* (Wiley, 2002). Il travaille actuellement à la rédaction de *Nordic Object/Relational Design In Action* (Manning), dont la publication est prévue pour la fin de l'année. Il propose des ateliers sur la conception et l'optimisation de bases de données. Paul Nielsen peut être contacté sur son site Web à www.SQLServerBible.com.

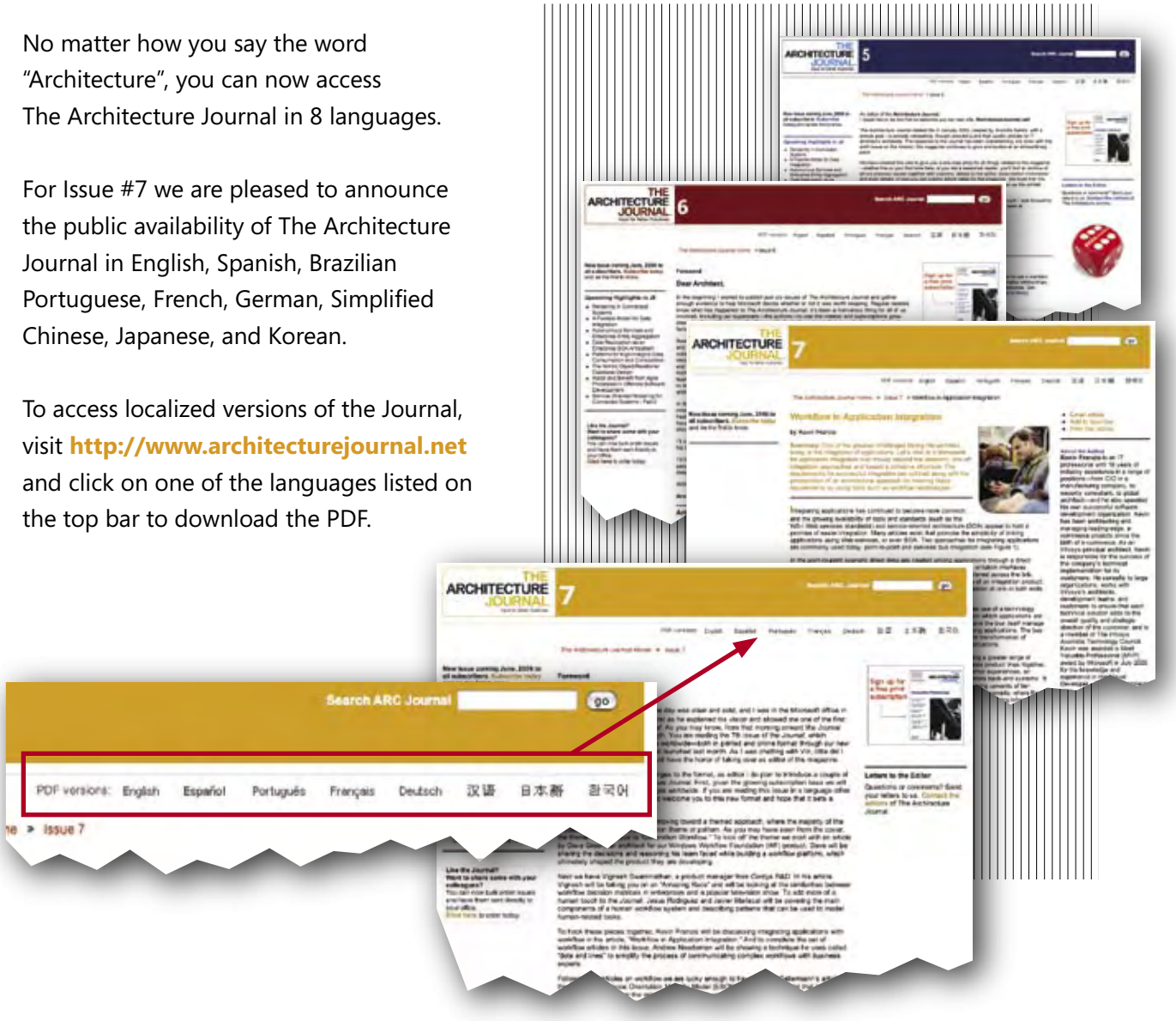
THE ARCHITECTURE JOURNAL™

Input for Better Outcomes

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

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

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



Now live at www.ArchitectureJournal.net!

Microsoft®

ARC

L'adoption des méthodologies agiles dans un projet Offshore

par Andrew Filev

Résumé

Dans le développement logiciel actuel, deux tendances permettent de faire plus avec moins : les méthodologies agiles et l'externalisation délocalisée (développement offshore). Voyons comment et quand combiner avec succès les deux pour accroître la compétitivité de votre entreprise.

Après l'éclatement de la bulle internet, les budgets informatiques ont été réduits plus que les besoins, ce qui a conduit les DSI à rechercher des solutions moins coûteuses, et accentué la tendance à délocaliser le développement dans des pays émergents (développement offshore). La pression économique n'a pas été le seul facteur dans cette évolution : la croissance exponentielle des infrastructures de communications a également été déterminante.

La collaboration d'équipes virtuelles, réparties géographiquement, a largement reposé sur les solutions de voix sur IP (VoIP), les messageries instantanées, l'e-mail et les wikis. Ces outils font également leurs preuves dans des équipes plus traditionnelles, et remplacent avantageusement les techniques classiques (et trop souvent orales) de collaboration ; en effet, ils structurent l'information, permettent de la conserver tout au long de la vie d'un projet, et bien sûr de la diffuser simplement à chaque personne concernée.

L'explosion de la bande passante a aussi permis à des outils plus classiques de prendre une dimension nouvelle. Les outils de modélisation permettent de rendre la documentation plus explicite pour une équipe répartie. Le suivi de bugs, les contrôleurs de sources, les portails web et les outils de collaboration en ligne aident à coordonner les projets répartis. Les machines virtuelles et les protocoles de prise de contrôle à distance jouent un rôle important pour les tests et l'administration.

L'Internet a également placé les marchés émergents sur la voie des hautes technologies. Comme il ignore les frontières géographiques, des milliers de jeunes gens de pays émergents, tels que la Russie et la Chine, l'utilisent pour apprendre des technologies de pointe et améliorer leurs connaissances en anglais. Cette nouvelle vague d'ingénieurs en logiciel formés sur le Web est arrivée juste à temps pour renforcer la tendance à la délocalisation.

Le récent engouement pour le développement offshore a été suffisamment important pour susciter des débats politiques. Dans cet article, nous considérons que le développement offshore est une réalité de fait et nous nous attachons à maximiser le retour sur investissement de tels engagements. Nous ne tiendrons pas compte des points de vue politiques, mais nous référerons à la liste de ressources établie par le service de recherche du McKinsey Global Institute, liste qui mesure les avantages de la délocalisation pour l'économie américaine et réduit à néant certains mythes.

La tendance agile

Revenons un instant sur les projets traditionnels. De nombreux responsables et ingénieurs sont conquis par une autre tendance actuelle : le développement agile. Les méthodologies *classiques* de développement n'ont pas fait leurs preuves dans l'environnement dynamique actuel. L'amaigrissement des budgets et l'augmentation des exigences ne font pas bon ménage avec des approches bureaucratiques peu tournées vers le retour sur investissement (ROI). La force des méthodologies agiles repose sur la collaboration, la flexibilité et le respect de la valeur métier des logiciels, comme le reflètent les principes clés du Agile Manifesto : *personnes et interactions* pour les processus et outils, *logiciels de travail* pour une documentation complète, *collaboration avec les clients* pour la négociation d'un contrat et *réactivité au changement* pour le suivi d'un plan (voir Ressources).

Les méthodologies agiles conviennent parfaitement à la nouvelle vague de start-ups Internet (souvent appelées Web 2.0), leur permettant de faire plus avec moins et de mener à terme d'importants projets avec de petites équipes et des budgets serrés. Leur fonctionnement (des itérations courtes produisant systématiquement un logiciel testable bien qu'incomplet) est reflété dans une pratique appelée *béta perpétuelle* depuis que plusieurs produits Google ont intégré le terme « bêta » dans leur logo.

Ces méthodologies ne sont pas *adaptées à tous les cas*. Elles fonctionnent bien pour de petites équipes mono-site confrontées à des exigences évoluant rapidement. Même s'il existe des cas où l'applicabilité du développement agile est sujette à question, comme dans le développement offshore, mes cinq années d'expérience concluante dans l'application de ces principes au sein d'équipes virtuelles démontrent que c'est possible et profitable lorsqu'on les utilise à bon escient.

> « LES MÉTHODES AGILES NE SONT PAS ADAPTÉES À TOUS LES CAS. ELLES FONCTIONNENT BIEN POUR DE PETITES ÉQUIPES MONO-SITE CONFRONTÉES À DES EXIGENCES ÉVOLUANT RAPIDEMENT. »

Il existe d'autres scénarios où les méthodologies agiles peuvent être remises en question. Les grandes équipes de développement (plus de 20 personnes travaillant sur un projet indivisible), les systèmes où la prévisibilité est primordiale (applications critiques) et les environnements bureaucratiques en sont quelques des exemples. Nous nous limiterons dans cet article à des entreprises dont la culture est compatible avec le développement agile, et à des équipes de moins de 20 personnes (par sous-projet indépendant, et non pour l'ensemble de l'équipe de développement). Nous déclinerons ensuite deux scénarios : les équipes réparties et les projets offshore.

Combinaison des tendances

Les contrats de développements offshore vont de l'embauche d'un développeur sur rentacoder.com jusqu'à des opérations de plusieurs milliards de

Figure 1 Intégration point-à-point et par bus de serviceson

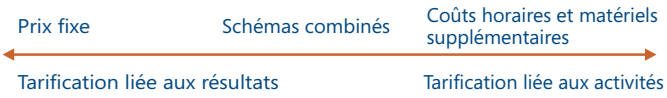


Figure 2 Trois couches d'intégration

dollars avec des entreprises américaines créant des filiales à l'étranger. Certains de ces contrats sont tels qu'ils empêchent les entreprises d'utiliser un processus de développement agile, même si l'une des parties le souhaite.

Pour mettre en œuvre un processus agile, le modèle d'externalisation sélectionné doit encourager les communications et la collaboration, permettre la flexibilité et justifier de fréquentes livraisons (itérations). Parmi les dizaines de critères pouvant être appliqués aux opérations d'externalisation, le modèle de tarification est sans doute le plus important pour la suite de notre exposé. Reportez-vous à la figure 1 pour le mappage des schémas de tarification.

Des résultats prévisibles impliquent des processus prévisibles. La figure 2 indique les groupes des processus de développement alignés sur l'échelle prévisionnelle-spéculative. Si nous utilisons un critère prévisionnel-spéculatif, les schémas prévisionnels, qui sont plus les proches de l'extrémité gauche de l'échelle (voir figure 1), requièrent plus de prévisibilité de la part du processus de développement logiciel tandis que les processus de développement agile se trouvent à l'opposé du continuum des processus de développement logiciel.

Le problème de la prévisibilité est tout particulièrement lié aux engagements de type *objectif fixe-prix fixe*. Lorsque ce type de contrat s'applique à une opération d'externalisation, le client et le fournisseur sont naturellement dépendants des processus de développement logiciel prévisionnels et de leurs conséquences. Ce contrat n'est, par conséquent, pas adapté au développement logiciel agile. Lors de la conception d'un engagement en matière d'externalisation, rappelez-vous que la réactivité aux changements encourage l'utilisation de modèles de tarification tels ceux relatifs à l'heure et au matériel, qui apportent flexibilité au client et au fournisseur et qui sont mieux adaptés au développement agile.

Pour élaborer un projet offshore, considérez l'impact qu'aura le schéma sélectionné sur les communications et la collaboration. Un processus de développement agile requiert un environnement ouvert, une étroite intégration au sein de l'équipe, des objectifs partagés, une compréhension de la valeur métier et une communication fréquente. Plus les barrières entre les ingénieurs, les clients, les utilisateurs, les responsables et d'autres parties prenantes sont nombreuses, plus il est difficile d'avoir une base pour le développement logiciel agile, qui signifie une réduction des intermédiaires pour permettre une transparence et une intégration maximales entre les équipes.

Les grandes multinationales ont résolu cette question en établissant leurs propres filiales dans d'autres pays, ce qui est justifié économiquement si une entreprise souhaite avoir plus de 100 ingénieurs dans son centre de développement à l'étranger. Le nombre exact dépend fortement de l'autre pays et de facteurs tels que la facilité d'embauche de salariés qualifiés.

Si vous considérez cette alternative, ne commettez pas l'erreur de sous-estimer les dépenses cachées liées, par exemple, au temps consacré par les cadres supérieurs, aux déplacements ou aux frais d'avocats. De plus, la croissance rapide des économies des pays en développement entraîne des pénuries de talents, ce qui ne facilite pas la vie des nouveaux entrants. Les entreprises locales connaissent bien leur marché et peuvent recruter efficacement ; les multinationales peuvent quant à elles jouer sur leur renommée, des salaires plus élevés et des avantages sociaux.

Un tel luxe est souvent hors de portée des petites entreprises, qui souhaitent avoir une quinzaine de développeurs à l'étranger. Les petites et moyennes entreprises performantes maîtrisent les coûts de gestion inhérents à la création de bureaux distants en faisant appel, entre autres, à des équipes de développement dédiées, à des centres de développement délocalisé, à des modèles BOT (build-operate-transfer), à des bureaux virtuels et à des équipes virtuelles. Indépendamment du nom et des détails, un point est commun à ces modèles : le fournisseur délocalisé s'attache plus à fournir une infrastructure physique, légale, informatique et RH au client qu'à participer au processus de développement. La responsabilité du projet dans de telles opérations est partagée entre le client et le fournisseur.

Pour tirer parti de tous les avantages d'un développement offshore, les ingénieurs doivent toujours être affectés à un même client et les taux de maintien des équipes doivent être bons dans les deux entreprises. Le fournisseur doit proposer des communications transparentes au client à tous les niveaux, incluant les communications de développeur à développeur. Les gens doivent pouvoir s'exprimer dans la même langue pour éviter de recourir à des traducteurs.

Dans les engagements porteurs de résultats, on constate parfois que même si les salariés des bureaux distants sont payés par le fournisseur, ils s'associent plus au client et partagent les valeurs et la culture des deux sociétés.

> « LE CHOIX DU BON MODÈLE EST IMPORTANT, MAIS IL NE GARANTIT PAS LE SUCCÈS. »

Utilisation des bonnes pratiques et des outils

Les équipes de développement performantes utilisent des pratiques connues : des normes de codage communes, un serveur de contrôle de source, des scripts automatisés de compilation et de déploiement, une intégration continue, des tests unitaires, un suivi des anomalies, des design patterns (modèles de conception) et des briques applicatives standardisées (par exemple les Application Blocks de Microsoft). Ces pratiques doivent être appliquées plus strictement aux équipes réparties qu'aux équipes locales.

Considérons, par exemple, l'intégration continue. Il peut s'avérer extrêmement frustrant d'arriver au travail et de constater que le contrôleur de sources contient une version du projet qui ne se compile pas, alors que la personne qui en est responsable se trouve à des milliers de kilomètres, au beau milieu de la nuit. Ce problème est secondaire si le collègue du bureau voisin en est à l'origine, mais peut devenir majeur dans un scénario réparti où il nuirait à la productivité et aux communications. Vous pouvez réduire de tels risques en adoptant des pratiques d'intégration continue au sein de toute l'équipe et en installant le serveur correspondant (tel que Microsoft Team Foundation Server, CruiseControl.NET et CruiseControl).

Les équipes travaillant sur la plateforme Microsoft .NET bénéficient directement des fonctionnalités fournies par Microsoft Visual Studio Team System. Elles ont accès à une structure normative pour le développement agile, ainsi qu'à des outils standardisés et intégrés. Ce produit est d'une grande utilité pour les équipes qui requièrent plus d'aide en matière de développement agile dans des environnements répartis. Pour les équipes expérimentées, il s'agit d'une solution intégrée qui fournit un retour sur investissement important.

Windows SharePoint Services (WSS) est un autre produit Microsoft qui apporte une grande valeur ajoutée aux équipes distribuées. Les wikis (sites Web modifiables) favorisent le développement agile au sein des équipes distribuées et la prochaine version de WSS prévoit de les compter parmi ses améliorations. WSS est aussi étroitement intégré à Visual Studio Team

System, ce qui en fait le meilleur choix pour le portail Web de l'équipe.

Pour ce qui est de l'infrastructure informatique, je recommande l'utilisation d'un réseau privé virtuel (VPN), qui donne aux équipes le même accès aux ressources partagées. L'environnement VPN, moins strict qu'un réseau public, permet d'utiliser des fonctionnalités telles que le partage d'applications de Windows Live Messenger, les appels voix et vidéo, l'assistance à distance et le tableau blanc.

Communication omniprésente

En travaillant à distance, de petits malentendus se transforment vite en gros problèmes. Les responsables des équipes de développement distribuées doivent être attentifs aux pratiques de communication, dont ils font parfois abstraction pour un développement local. Ceci inclut des comptes rendus réguliers (quotidiens/hebdomadaires) et des réunions sur l'état en cours, qui permettent aux membres de l'équipe d'être synchrones, de discuter des résultats obtenus et de mettre en évidence les problèmes. Les responsables doivent également s'attacher à créer des relations au sein de leurs équipes, à l'aide de réunions de présentation, de visites sur site, d'activités promouvant l'esprit d'équipe et d'autres méthodes.

Lors d'un projet offshore, il faut tenir compte des obstacles linguistiques, culturels et temporels, et trouver des façons de les surmonter. La mondialisation efface lentement mais sûrement les différences culturelles dans l'environnement professionnel, mais il demeure des cas où ces dernières sont source de confusion. Les problèmes relatifs à ce domaine, mais propres à un pays en particulier, ne sont pas abordés ici. Les problèmes linguistiques sont plus faciles à détecter, bien que tout aussi difficiles à surmonter. Lorsqu'une entreprise est confrontée à des obstacles d'ordre linguistique, il est habituel et fortement recommandé qu'elle propose des formations en langues à ces salariés. Dans la plupart des pays où le développement est délocalisé, les informaticiens sont désireux d'apprendre l'anglais : ce sont donc généralement les gens de ces pays qui bénéficient d'une formation en langue.

Les différences de fuseaux horaires compliquent également le processus. Il apparaît toutefois que dans les pays où l'industrie de l'externalisation est développée, les équipes sont habituellement disposées à adapter leurs horaires de travail pour collaborer avec leurs homologues de l'étranger. Il existe deux stratégies pour gérer les décalages horaires. La première consiste à répartir les équipes par activité, par exemple, avoir les responsables produits et assurance qualité sur site et les développeurs à l'étranger. Une telle organisation permet de mettre en place un cycle, durant lequel les développeurs implémentent les corrections et les nouvelles exigences pendant que leurs homologues dorment, et vice versa. Il doit bien sûr y avoir des heures où tout le monde travaille (au début/à la fin d'une journée ouvrée). La seconde approche consiste à répartir les projets par bloc et à affecter chaque bloc à un site, en déléguant autant de fonctions que possible à ce dernier. La seconde approche induit une meilleure communication et facilite, ce faisant, un développement agile, mais les deux approches sont envisageables (le choix est parfois dicté par des impératifs externes).

Le choix du bon modèle est important, mais il ne garantit pas le succès. Il est fortement recommandé qu'au moins une partie de l'équipe possède déjà une expérience de développement agile, de préférence dans un environnement réparti. L'absence de communication directe ainsi que les différences linguistiques, culturelles et temporelles requièrent l'attention et l'investissement d'efforts supplémentaires pour obtenir les résultats escomptés. Les avantages d'avoir un partenaire délocalisé fiable, à savoir les économies, la souplesse d'embauche et de débauche, et la délégation des tâches d'infrastructure compensent largement l'investissement initial de création de relations productives. Ce résultat positif serait impossible sans les outils modernes d'une infrastructure de communication performante désormais disponible partout dans le monde. •

« UN PROCESSUS DE DÉVELOPPEMENT AGILE REQUIERT UN ENVIRONNEMENT OUVERT, UNE ÉTROITE INTÉGRATION AU SEIN DE L'ÉQUIPE, DES OBJECTIFS PARTAGÉS, UNE COMPRÉHENSION DE LA VALEUR MÉTIER ET UNE COMMUNICATION FRÉQUENTE. »

Ressources

« Exploding the Myths of Offshoring », Martin N. Baily et Diana Farrell, *The McKinsey Quarterly* (McKinsey & Company, 2004)
www.mckinseyquarterly.com (Note : inscription requise.)

Manifesto for Agile Development
http://agilemanifesto.org

À propos de l'auteurr

Andrew Filev (MCA, MVP) est vice-président et responsable des opérations délocalisées chez Murano Software. Il crée des centres de développement offshore et dirige des équipes. Excellent communiquant, Andrew Filev comble le vide entre les différentes cultures et crée des partenariats durables avec les clients.

Modélisation orientée services pour systèmes connectés – Partie 2

par Arvindra Sehmi et Beat Schwegler

Résumé

En tant qu'architectes, nous pouvons adopter un nouveau mode de réflexion, consistant principalement à nous imposer la prise en compte explicite des artéfacts de modèles de services lors de nos processus de conception, ce qui nous aidera à identifier correctement les artéfacts, avec le niveau d'abstraction approprié, afin de satisfaire et de nous aligner aux besoins métier de nos entreprises. Dans la première partie de cet article, nous avons proposé une approche en trois parties de la modélisation des systèmes orientés services connectés, en prônant un alignement entre la solution informatique et les besoins de l'entreprise. Nous avons analysé la perspective actuelle en matière de réflexion orientée services et expliqué comment notre mode de réflexion actuel et une mauvaise conceptualisation de l'orientation des services ont conduit à de multiples échecs ainsi qu'à des taux de rentabilité globalement faibles. Nous avons également passé en revue les avantages liés à l'insertion d'un modèle de service entre les modèles conventionnels métiers et technologiques, que la plupart des architectes connaissent, avant d'analyser la méthodologie et le mappage des capacités de Microsoft Motion afin d'identifier les capacités métier pouvant être mappées sur les services. Dans cette seconde partie de l'analyse, nous vous montrerons comment implémenter ces services mappés.

Nous avons achevé la première partie de cet article sur un processus pragmatique d'analyse et de conception orienté services (SOAD, Service Oriented Analysis and Design) utilisé pour extraire l'ensemble des éléments nécessaires à la création de votre modèle de service. Cette extraction concerne les contrats de service, les contrats de niveau de service (SLA, service-level agreements) qui sont déduits des attentes de niveau de service (SLE, service-level expectation) définies pour chaque capacité métier et les critères d'orchestration des services. Après avoir créé un modèle de service détaillé, adapté au mieux au modèle métier dont il est tiré, vous devez désormais être en mesure de le mapper sur un modèle technologique qui identifie la manière d'implémenter, d'héberger et de déployer chaque service.

En utilisant l'approche précédente et en créant le modèle de service, vous pouvez transmettre à votre département informatique les schémas de données, les contrats de services et les conditions du SLA. Toutefois, avant que le service ne soit créé, il convient de supporter son autonomie au niveau technologique en séparant clairement les interfaces de l'implémentation et des mécanismes de transport sous-jacents. L'élaboration de stratégies d'implémentation du service qui découple son point terminal de son implémentation vous permet de vous préparer à des évolutions. La sélection d'options appropriées de gestion des services et des hôtes se conformant aux SLA définis requiert également une attention particulière. Vos choix dans ces domaines sont conservés et définis par le modèle technologique.

Création d'un modèle technologique

Le modèle technologique consiste en plusieurs artéfacts : l'interface des services, l'implémentation des services, l'hôte des services, la gestion des services et le moteur d'orchestration. Nous considérons ci-après chacun d'eux en détail.

L'interface du service spécifie la façon dont un document ou un message peut être reçu. Elle vous permet d'indiquer les transports fournis, indépendamment de l'implémentation. Plusieurs interfaces de services peuvent implémenter un contrat de service, mais chacune d'elle implémente une liaison spécifique telle que SOAP via HTTP.

Si nécessaire, vous pouvez fournir plusieurs interfaces de services utilisant différents transports. Vous pouvez, par exemple, lier une interface à un transport de services Web et également à un transport Message Queuing de Windows. Chaque transport fournit différentes capacités, telles que l'interopérabilité et la prise en charge des transactions ainsi que différentes restrictions. Par exemple, Message Queuing ne prend pas directement en charge le modèle requête/réponse. Au moins une interface doit être fournie pour l'interopérabilité, en s'assurant qu'elle est conforme à la version 1.x de WS-I BP (Web Services Interoperability Basic Profile).

L'implémentation du service consiste en l'implémentation d'une capacité métier indépendante de l'hôte sous-jacent. Elle ne doit pas non plus dépendre de ses interfaces. L'implémentation des services peut appeler d'autres services en utilisant les proxys dépendant d'une liaison ; pour avoir des proxys ne dépendant pas d'une liaison, ils doivent être créés à l'aide de fabriques.

L'hôte du service fournit un point terminal aux interfaces de services. Son choix doit s'effectuer en fonction des exigences spécifiées dans le SLA. Par exemple, s'il est prévu que l'activité fonctionne 24 heures sur 24, 7 jours sur 7, cette caractéristique doit avoir une influence déterminante sur votre choix d'hôte et, dans ce cas, des technologies de file d'attente, telles que Microsoft Message Queue ou SQL Server Service Broker, sont souvent requises. Dans le modèle technologique, les hôtes représentent les options de coût. Dans le modèle métier, le SLE représente la valeur. En parvenant à mapper le choix de l'hôte à un SLA décrit dans le modèle de service et, finalement, à une SLE métier, le coût d'un hôte peut être justifié et quantifié.

Le fait de pouvoir migrer un service vers un hôte qui fournit les performances et la redondance nécessaires lorsque l'on a une capacité métier spécifique associée nécessitant de fonctionner 24 heures sur 24, 7 jours sur 7 constitue un avantage supplémentaire du passage aux services et de l'abandon des applications silo. Vous pouvez utiliser des hôtes moins coûteux pour d'autres capacités. Avec l'approche en silo, vous êtes obligé de sélectionner un hôte pour l'ensemble de l'application.

Dans le cas de l'artéfact *Gestion du service*, une action appropriée doit être effectuée si un SLA ne peut pas être rempli ; pour prendre cette action en charge, vous devez pouvoir surveiller et gérer le service. Les interfaces,

implémentations et hôtes de services doivent être instrumentés, par exemple, à l'aide de Windows Management Instrumentation (WMI). Microsoft Operations Manager (MOM) peut, dans ce cas, être utilisé pour surveiller et gérer le service, alors que Microsoft Systems Management Server (SMS) permet de prendre en charge la gestion de la configuration et des changements dans l'écosystème « périphérique » du service. Le moteur d'orchestration que vous avez choisi doit également proposer un support de supervision, telles les fonctionnalités BAM (Business Activity Monitoring) fournies par BizTalk Server. À l'avenir, WS-Management jouera un rôle central pour gérer les ressources indépendamment de la plateforme de gestion et d'hébergement.

En ce qui concerne l'artéfact *Moteur d'orchestration,* les exigences d'orchestration doivent être définies de façon indépendante de la plateforme par le modèle de service mais, à terme, il convient d'utiliser un moteur d'orchestration spécifique. Le modèle technologique identifie la plateforme cible, telle que Microsoft BizTalk Server.

En créant le modèle technologique, vous pouvez explicitement extraire les artéfacts précédents, mais comment avoir une approche du développement de service ? Comment mapper ces artéfacts à une implémentation de service et comment mettre en œuvre le contrat spécifié par le modèle de service ? Nous décrirons plus loin une approche permettant de créer des services satisfaisant aux principes traités plus haut.

Création de services à l'heure actuelle

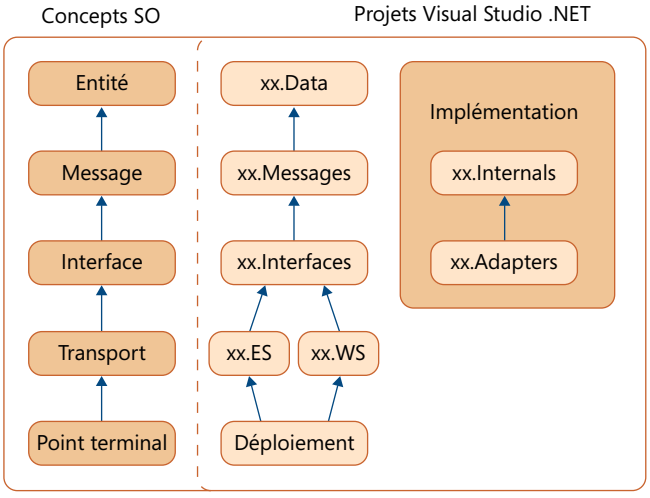
Comment définir un ensemble d'artéfacts de services conceptuels sous la forme de codes et organiser ces derniers dans Visual Studio 2005 ? L'important est de garder à l'esprit la nécessité de séparer les données, les messages, les interfaces, les aspects internes d'une implémentation et les liaisons de transport. La figure 1 décrit une approche de mappage des artéfacts de services aux projets et solutions Visual Studio 2005.

Les emplacements réservés « xx » indiqués dans la figure 1 représentent un espace de nom approprié basé sur le nom de service, par exemple, OrderService. L'utilisation d'un fichier de solution Visual Studio 2005 unique pour un service donné et comme conteneur des différents projets indiqués dans la figure 1 est recommandée. Considérons ces projets.

> « CHAQUE TRANSPORT FOURNIT DIFFÉRENTES CAPACITÉS, TELLES QUE L'INTEROPÉRABILITÉ ET LA PRISE EN CHARGE DES TRANSACTIONS AINSI QUE DIFFÉRENTES RESTRICTIONS »

- Le projet *xx.Data* sert à conserver les définitions de schémas de données obtenues à partir du modèle de service ; il contient également la représentation CLR (Common Language Runtime) de ce schéma.
- Le projet *xx.Messages* sert à conserver les définitions des messages requis pour communiquer avec le service, à savoir les messages entrants et sortants. Un message est représenté en tant qu'élément de schéma contenant des éléments du contrat de données.
- Le projet *xx.Interfaces* contient les définitions des interfaces.
- Les projets *xx.ES* et *xx.WS* contiennent des points terminaux de transport pour les interfaces. Dans l'exemple de la figure 1, xx.ES fournit un transport de services d'entreprise et xx.WS un transport de services Web.
- Les projets *xx.Adapters* et *xx.Internals* contiennent l'implémentation du service. Un adaptateur contenu dans le projet xx.Adapters fournit un niveau d'indirection entre l'interface et l'implémentation interne. L'adaptateur analyse le message entrant et transmet les données pertinentes au code d'implémentation interne. Il intègre également les données provenant de l'implémentation dans le message de réponse.

Figure 1 Mappage des artéfacts de services aux projets Visual Studio 2005



Notez qu'avec cette approche, l'implémentation ignore tout des messages transmis via l'interface, ce qui vous permet de modifier l'infrastructure de messagerie sans que cela ait un impact sur l'implémentation de vos services internes.

Le point terminal indiqué dans la figure 1 est un artéfact de déploiement qui dépend de la liaison de transport choisie. Par exemple, avec un transport de services Web, vous déployez votre service dans un serveur Web IIS (Internet Information Services) exécutant ASP.NET.

Adaptateurs. L'adaptateur est un artéfact clé qui vous permet de découpler votre implémentation de l'échange de messages. C'est également l'endroit idéal pour transformer des messages, si nécessaire. Exemple : dans l'adaptateur, vous pouvez transformer une représentation externe d'un numéro d'identification client (par exemple, 10 caractères) en votre représentation interne (par exemple, 12 chiffres). En créant un adaptateur pour chaque version de votre service, vous avez également un moyen de créer des versions de vos services sans nuire à l'implémentation ou à vos interfaces de service préalablement publiées.

Gestion de versions. Il convient de faire la distinction entre la gestion de versions du contrat concret ou abstrait et la gestion de versions de l'implémentation interne. Fournir une gestion de versions indépendante des deux constitue un objectif clé en matière d'architecture. Le contrat, à proprement parler, consiste en des données, des messages, des interfaces et des points terminaux. Vous pouvez gérer les versions de chacun de ces artéfacts indépendamment si vous concevez l'architecture du service de manière que chaque artéfact de composition fasse référence à une version unique de l'artéfact composé. Par exemple, le contrat du message composé du contrat de données doit faire référence à une version précise de ce dernier. Cette référence doit être vraie pour la représentation XSD/WSDL et CLR de ce contrat.

Les versions des contrats de données et de message doivent être gérées conformément à la stratégie d'extensibilité et de gestion de versions XML (voir Ressources). Vous pouvez gérer les versions des interfaces de service en fournissant une nouvelle interface héritant de la précédente (CLR) et mappée dans un nouveau type de port WSDL.

Six étapes pour créer un service

Pour parvenir à ces projets Visual Studio 2005 et créer un service à l'aide de cette approche, effectuez le processus suivant composé de six étapes :

1. Créez le contrat de données et de message.
2. Créez le contrat de service.

3. Créez les adaptateurs.
4. Implémentez les aspects internes du service.
5. Connectez les aspects internes aux adaptateurs.
6. Créez les interfaces de transport.

Examinons chaque étape en détail. Pour la première étape (création du contrat de données et de message), prenez le schéma de données canonique qui définit la représentation des données (le résultat de votre conception et analyse orienté services) et définissez les classes de données et de messages. Il existe deux approches pour créer vos schémas et vos classes de données. En utilisant une approche basée sur les schémas, vous pouvez créer un schéma XSD, puis utiliser un outil tel que Xsd.exe ou XsdObjectGen.exe pour générer les classes de données automatiquement. Si vous préférez une approche basée sur les codes, vous pouvez également définir vos classes de données dans C# ou Visual Basic, puis utiliser Xsd.exe pour créer un schéma XSD équivalent. Voici un exemple de contrat de données :

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

Une fois vos classes de données définies, vous pouvez définir vos messages entrants et sortants, qui contiennent leurs classes de données en tant que charge utile. Par exemple, cet extrait de code indique un message entrant appelé Message de Commande pour un service d'ordre hypothétique qui contient une commande en tant que charge utile :

```
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;
   }
}
```

À l'étape 2 (création du contrat de service), définissez le contrat de service abstrait en utilisant une approche basée sur le WSDL ou en définissant vos interfaces à l'aide de C# ou de Visual Basic. Vos interfaces définissent les messages reçus et, le cas échéant, renvoyés par votre service. Pour générer l'interface à partir du WSDL, vous devez utiliser le commutateur Wsdl.exe /si :

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

Notez que cette option fonctionne uniquement avec Microsoft .NET Framework version 2.0. L'exemple suivant définit une interface à l'intention du service de commandes qui définit une méthode PlaceOrder() unique, laquelle accepte un message OrderMessage et renvoie un message OrderTrackingMessage :

```
namespace Interfaces
{
   using MessageContracts;

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

Notez que dans ce cas, un modèle d'échange de message de type requête/réponse est utilisé : par conséquent, vous devez veiller à ce que le temps de traitement du message (le temps entre la requête et la réponse) soit inférieur à une seconde. Si tel n'est pas le cas, utilisez un autre modèle d'échange de messages, tel que l'échange de messages duplex, qui met en corrélation deux messages à sens unique avec un modèle requête/réponse logique.

Pour générer le WSDL à partir de l'interface précédente, vous pouvez créer une classe Web service ASMX qui implémente l'interface, puis appelle le service Web passant ?WSDL, par exemple, http://localhost/OrderService/OrderService.asmx?wsdl. Le WSDL est généré et renvoyé à partir de ce service Web.

À l'étape 3 (création des adaptateurs), créez la classe d'adaptateurs fournissant l'indirection entre l'interface et les aspects internes. Cette classe garantit que les aspects internes ignorent tout du contrat de message. L'adaptateur ouvre le message entrant et transmet les données de charge utile à l'implémentation interne, effectuant toutes les transformations de format de données requises du message au format interne. De même, au retour, l'adaptateur effectue toutes les transformations de format nécessaires sur les données renvoyées par l'implémentation du service et les intègre à un message de service sortant, s'il y en a un. Voici un exemple d'adaptateur :

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

   public class OSA : IOrderService
   {
      public virtual
         OrderTrackingMessage
         PlaceOrder(
         OrderMessage placeOrderMsg)
      {
```

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

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

Liste 1 Interface IOrderService

```
        // Call internals here
      ...
      }
   }
}
```

Notez que l'adaptateur est toujours en traitement avec l'appelant et que l'identité du processus dépend de l'hôte choisi. Si l'implémentation interne est hébergée dans IIS, l'interface (« périphérique ») ASMX crée une instance de l'aspect interne du service. Si l'aspect interne est hébergé dans les Enterprise Services, mais que les appels parviennent via un service Web ASMX, l'interface ASMX délègue l'appel à l'interface Enterprise Services, qui crée ensuite une instance de l'adaptateur et lui transmet le message. Dans la mesure où tous les transports implémentent la même interface, vous pouvez relier tous les appels ensemble.

À l'étape 4 (implémentation des aspects internes du service), effectuez l'implémentation interne de votre service. Il n'existe qu'une implémentation indépendamment du/des transport(s) choisi(s). L'extrait de code ci-dessous indique la structure requise pour effectuer une implémentation du service de traitement des commandes :

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

Notez que l'implémentation interne n'a pas connaissance du message. Il connaît uniquement le contrat de données. Dans ce cas, il est transmis un objet Commande unique. Si vous souhaitez être totalement indépendant du format circulant, l'aspect interne ne doit même pas avoir accès au contrat de données. Dans ce scénario, l'adaptateur doit mapper le contrat de données externe aux types de données internes.

À l'étape 5 (connexion des aspects internes aux adaptateurs), appelez l'implémentation interne de votre service à partir du code de votre adaptateur. Notez que le message n'est pas transmis à l'implémentation, ce qui découple l'implémentation interne du contrat de message et vous permet de modifier l'un sans que cela ait un impact sur l'autre. L'extrait de code ci-dessous indique à nouveau le code de l'adaptateur, cette fois-ci avec un appel à l'implémentation de service interne :

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

   public class OSA : IOrderService
   {
      public virtual
         OrderTrackingMessage
         PlaceOrder(
         OrderMessage placeOrderMsg)
      {
         OrderTrackingMessage otm =
            new OrderTrackingMessage();
         otm.TrackingId =
            BusinessLogic.OSI.
            AcceptOrder(
            placeOrderMsg.Order);
         return otm;
      }
   }
}
```

À l'étape 6 (création des interfaces de transport), liez vos interfaces de service abstraites définies à l'étape 2 à un transport spécifique. La liste 1 indique l'interface IOrderService définie à l'étape 2 et rattachée à un transport de services Web.

Utilisation de l'aide automatisée sur les processus

En suivant le processus en six étapes décrit ci-dessus, vous pouvez créer des services conformes à tous les principes énoncés dans cet article. Toutefois, dans la mesure où tous les éléments décrits dans ce processus peuvent l'être à l'aide de métadonnées, il est possible d'automatiser d'importantes sections de la génération de service. Vous pouvez utiliser des outils tels que la boîte

à outils GAT (Guidance Automation Toolkit) pour effectuer l'automatisation de l'aide (voir Ressources). L'aide automatisée sur les processus est particulièrement utile pour transformer des artéfacts XML et CLR identiques sur un plan sémantique, tels que les classes XSD et CLR, transformer le type de port WSDL et les interfaces CLR, générer les adaptateurs basés sur la description de l'interface et générer plusieurs technologies de point terminal basées sur la description de l'interface.

L'ancien mode de réflexion sur l'orientation des services est inopérant et un changement de perspective s'impose. En tant qu'architectes, nous pouvons adopter ce nouveau mode de réflexion afin d'*imposer* la prise en compte explicite des artéfacts de modèles de services à votre processus de conception, ce qui aidera à identifier correctement les artéfacts, avec le niveau d'abstraction approprié, afin de satisfaire et de s'aligner aux besoins métier.

Perçu sous l'angle de la modélisation, le fossé entre les modèles commerciaux et technologiques usuels est trop grand, ce qui est un facteur décisif de l'échec de nombreuses initiatives en matière d'orientation des services. Nous venons de présenter un modèle triparti, avec l'insertion d'un modèle de service entre les modèles métier et technologique, pour encourager une meilleure adéquation entre les services et les besoins de l'entreprise. Après avoir créé un modèle de service détaillé, adapté au mieux au modèle métier dont il est tiré, vous êtes désormais bien placé pour le mapper sur un modèle technologique qui identifie la manière d'implémenter, d'héberger et de déployer chaque service. Le mappage de capacités et la méthode Motion constituent un moyen efficace d'identifier les capacités métier et, à terme, les services. La décomposition de l'entreprise en capacités permet un découplage de haut niveau des contrats de service sous-jacents, et non l'inverse, comme c'est souvent le cas de nos jours.

Les systèmes connectés sont des exemples du modèle triparti et ils respectent les quatre principes de l'orientation des services. Ils peuvent être implémentés de manière *plus complète* à l'aide des cinq piliers des technologies Microsoft en matière de plateformes. Rappelez-vous les questions que nous avions posées au départ : Comment éviter de répéter les erreurs commises en matière de SOA par des initiatives précédentes, pourtant porteuses d'espoir ? Comment s'assurer que l'architecture d'implémentation choisie est adaptée à l'état réel ou prévisionnel de l'entreprise ? Comment mettre en place une solution viable, capable de réagir au caractère dynamique et changeant de l'entreprise ? En d'autres termes, comment mettre en place et gérer une entreprise agile ? Comment migrer vers ce nouveau modèle sans heurts et à un rythme maîtrisable ? Comment réaliser ce changement avec une perception claire des domaines susceptibles d'optimiser la valeur ajoutée de l'entreprise dès le départ ?

L'orientation des services au moyen de services Web n'est que l'implémentation d'un modèle spécifique. Ce sont la qualité et les bases du modèle qui déterminent les réponses à ces questions.

Remerciements

Les auteurs tiennent à remercier Ric Merrifield (directeur, Microsoft Business Architecture Consulting Group), David Ing (architecte logiciel indépendant), Christian Weyer (architecte, Thinktecture), Andreas Erlacher (architecte, Microsoft Autriche) et Sam Chenaur (architecte, Microsoft Corporation) d'avoir bien voulu donner leur avis sur les premières moutures de cet article. Nous tenons également à exprimer notre reconnaissance à Alex Mackman (technologue principal, CM Group Ltd.), chercheur et auteur hors pair qui nous a rendu un énorme service. •

Ressources

« Designing Extensible, Versionable XML Formats » (Conception de formats XML versionnables et extensibles) Dare Obasanjo (Microsoft Corporation, 2004) http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexxml/html/xml07212004.asp Conception de formats XML versionnables et extensibles

« Modeling and Messaging for Connected Systems » (Modélisation et messagerie pour les systèmes connectés) Arvindra Sehmi et Beat Schwegler Une diffusion sur Internet d'une présentation faite lors du Sommet des architectes d'entreprise tenu à Barcelone (FTPOline.com, 2005) www.ftponline.com/channels/arch/reports/easbarc/2005/video/

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

Pour obtenir une étude de cas sur la méthode Microsoft Motion, envoyez une demande à motion@microsoft.com.

« Modélisation de services pour systèmes connectés – Partie 1 », Arvindra Sehmi et Beat Schwegler, *The Architecture Journal,* numéro 7, mars 2006. http://www.thearchitecturejournal.net

À propos des auteurs

Arvindra Sehmi est le responsable de l'architecture d'entreprise du Microsoft EMEA Developer and Platform Evangelism Group. Travaillant essentiellement à l'adoption des meilleures pratiques en matière de conception de logiciels d'entreprise par les développeurs et les architectes de la région EMEA, il dirige également la promotion en matière d'architecture auprès du secteur des services financiers. M. Sehmi est un rédacteur émérite de The Architecture Journal. Titulaire d'un doctorat en génie biomédical, il possède également un Master en économie.

Beat Schwegler est architecte au sein du Microsoft EMEA Developer and Platform Evangelism Group. Assistant et conseiller en architecture logicielle et sujets connexes auprès d'entreprises, il participe souvent à des conférences et événements internationaux. Fort de plus de 13 ans d'expérience en développement de logiciels professionnels et en architecture, il a pris part à des projets très diversifiés, qui vont des systèmes de contrôle d'immeubles en temps réel et des produits sur étagère les plus vendus aux systèmes CRM et ERP de grande envergure. Ces quatre dernières années, M. Schwegler a consacré l'essentiel de ses efforts à l'orientation des services et aux services Web.


ready_
ready_
ready_
Welcome to the people ready business.

Your potential. Our passion.™

Microsoft®

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

ARC

Microsoft®

THE ARCHITECTURE JOURNAL™

Des données pour de meilleurs résultats

098-105499

Inscription à : www.architecturejournal.net