連載　Web サービス
第 7 回

Web サービスのパターンとベストプラクティス

日本アイ・ビー・エム（株）
天野 富夫　*amano@jp.ibm.com*

編集：XML コンソーシアム

SOA は 1 日にして成らず？

前回の連載ではビジネスの変化に IT のシステムが俊敏に対応するための設計原則として SOA（サービス指向アーキテクチャ）の意義が紹介されました．WSDL でサービスのインタフェースを記述する，BPEL4WS を使ってサービスを組み合わせてより高機能のサービスを合成する，など Web サービスの規格はこのような SOA の実現に重要な役割を果たします．

しかし，Web サービスの実装を使えばそれで SOA は完成かというと実はそうとも言えません．ビジネスの定義から始まって Java や .NET による実装に到る e-business の設計過程のどこで Web サービスを意識するのでしょうか？　また，Web サービスを使うべき場所や逆に使ってはいけない状況というのがあるのでしょうか？　このような質問に応え SOA の開発を容易にするため，サービスを識別し仕様を決めるための方法論や Web サービス適用のパターンが整備されつつあります．パターンといってもビジネスやアプリケーションの基礎となる構造を決定する粒度の大きいものから個別の実装技術の適用の際のベストプラクティスを規定する細粒度のものまでさまざまですが，本稿ではその一部を紹介したいと思います．

Web サービスが使われるパターン

まずは既存のパターンと Web サービスがどのように関係するかを見てみましょう．IBM の Adams らはさまざまな e-business の形態を，情報交換を行う参加者の種類によって表 **-1** に示すような 4 つのビジネスパターンに分類しています．

e-business アプリケーションの中に現れるさまざまな情報交換の多くはこれら 4 つのパターンのどれかに当てはめることができます．

実際のアプリケーションはこれらのビジネスパターンを複数組み合わせて実現されます．ビジネスパターンの組合せ方法にもパターンがあって，こちらの方はアプリケーション統合パターンと呼ばれています．実はこのアプリケーション統合パターンの実装に関して SOA や Web サービスが重要な役割を果たします．

アプリケーション統合パターンは，統合対象となるアプリケーション間のインタラクション（機能を提供する側と利用する側のやりとり）の並列性（図 **-1(b)** と逐次性（図 **-1(c)**）の観点から表 **-2** に示す 4 つに分類されます．

これらの 4 パターンは実装の技術とは切り離された一般的なアプリケーション統合のメカニズムを示しています．さまざまなアプリケーション統合のソリューションは，多くの場合この 4 つのパターンのどれかにあてはまることが経験的に知られています．これらのパターンを要件に応じて選択する基準や実現の際必要になる構成要素やミドルウェアも過去の実装の事例から分かっています．並列性を持つインタラクションの実現には，アプリケーションからの要求を複数の要求元に伝えたり複数の相手の中から動的に要求先を決めるブローカやルータと呼ばれるミドルウェアが使われます．また，逐次性を持つインタラクションの実現には決められたプロセス定義にしたがってアプリケーションへの要求を順次発行する

パターン名	参加者	実施例
セルフサービス	人間と企業	Web サイトでの商品購入
コラボレーション	人間と人間	ビデオコンファレンス，メール
情報集約	人間とデータ	ナレッジマネジメント，ビジネスインテリジェンス
拡張エンタープライズ	企業と企業	EDI，サプライチェーン管理

表 **-1**　ビジネスパターンの分類

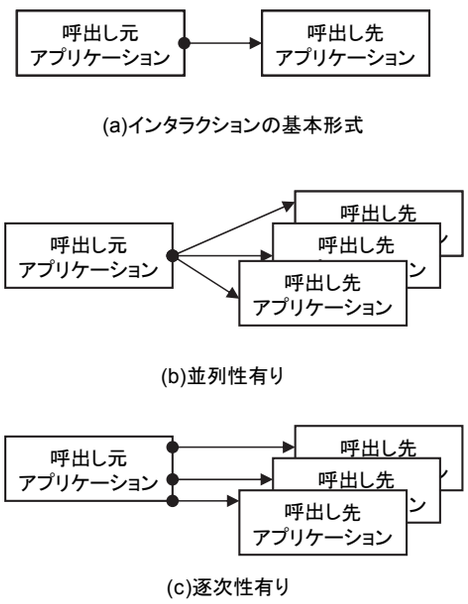

図 **-1**　アプリケーション間のインタラクションの種類

パターン名	インタラクションの性質	
	並列性	逐次性
直接接続（Direct Connection）	無し	無し
ブローカ（Broker）	有り	無し
順次プロセス（Serial Process）	無し	有り
並列プロセス（Parallel Process）	有り	有り

表 **-2**　アプリケーション統合パターンの分類

ワークフローエンジンが必要になります．

Web サービスでは上記のブローカ／ルータやワークフロー制御を行う技術／製品がすでに提供されています．SOAP over HTTP で DNS を使って URL からサービス提供者の IP アドレスを決定するのもルーティングの簡易実装とみなすことができます．後述の ESB（Enterprise Service Bus）のように，より複雑なルーティングやメッセージ変換の仕掛けをサポートする製品も現れています．ワークフローエンジンとしては BPEL4WS をプロセス定義に用いる製品が出荷されています．BPEL4WS を含めて多くのプロセス定義言語では並列実行を記述できるのでこれらの対応製品を用いれば並列プロセスパターンも実現可能です．

Web サービスにはアプリケーション統合のパターンを実現するのに必要な道具立てが整っているわけです．もちろん，Web サービスだけが 4 つのアプリケーション統合パターンを実現する唯一の方法だということではありません．従来の EAI 製品やワークフロー製品でも 4 つのアプリケーション統合パターンは実現可能です．しかし，ベンダ独自の製品では統合対象となるアプリケーションの実装が各ベンダがサポートするものに限られる傾向があります．SOAP や WSDL，BPEL4WS などベンダ製品に依存しないオープンな規格を用いて統合パターンを実現できる点に Web サービスの価値があります．

SOA と ESB パターン

昨今，SOA の実現のための重要な道具立てとして ESB というキーワードを耳にしている読者も多いと思います．いろんな説明のしかたがありますが，筆者自身は ESB を SOA におけるサービスの一元管理のためのアーキテクチャパターンとしてとらえる見方[1)] が分かりやすいと感じています．

図 **-2** に示すように企業／組織内では複数の個別に実現されたサービスが提供されています．個別に実現されたため各サービスを使用する際のインタラクションの性質もまちまちです．商品の受注サービスと発送状況照会サービスは別々のサーバによって提供され要求元を認証する仕組みも異なっているかもしれません．メッセージフォーマットやプロトコルについては WSDL の記述を公

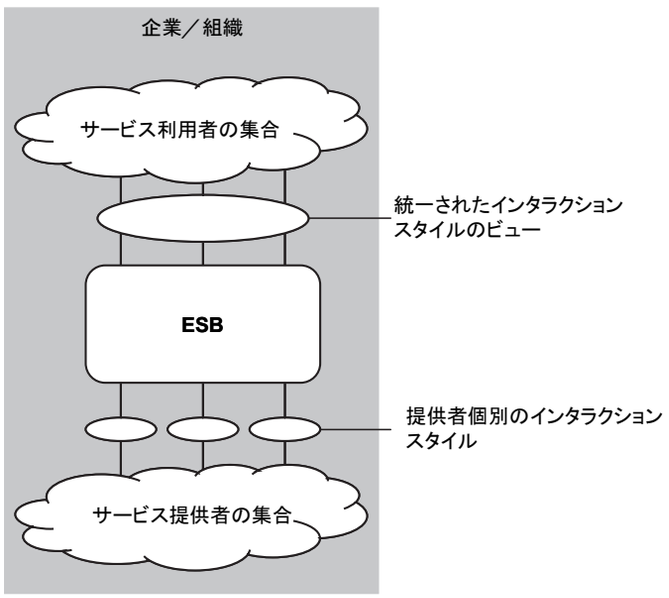

図 -2　Enterprise Service Bus パターン

開することで呼出し側のメカニズム（WSIF など）でアプリケーションから差異を隠蔽することができるかもしれませんが，それ以外のインタラクションのスタイル（たとえば Quality of Service やセキュリティの実現方式，サービスのバージョン管理，ステート管理のポリシーなど）のバリエーションについては呼出し側で個別に対応することは困難です．

サービスの提供者と利用者の間にインタラクションスタイル全般の変換を行う ESB を導入することで利用者側からは統一されたインタラクションのビューでサービスを利用することができます．前述の受注サービスと照会サービスも ESB によって同一のサーバ上で動き共通の認証機構を利用する統一された使い勝手を持つサービスとして利用者に提供されます．利用者に対して提示しているサービスの URL を ESB が実際の提供者の URL にルーティングし認証情報の変換を行うからです．ESB の実装にはメッセージのルーティングやフォーマットの変換，分解／合成を行うブローカやルータが含まれます．その意味でアプリケーション統合パターンのブローカパターンなどとは密接な関係があります．また，ESB はある Web サービスのセキュリティ要件やバージョン管理を行うために他の Web サービスを呼び出す等のワークフロー的な制御を行います．ただし，ESB の行うルーティングやプロセス制御はサービスの業務内容に立ち入らないインフラ的な部分に限定するのが一般的なようです．

物理的にはこれらの機能は 1 カ所で集中して行われても分散していてもかまいません．ある ESB のドメインに属するサービスについてインタラクションのスタイルを規定するさまざまなパラメータを 1 カ所で管理するという点が重要です．

Web サービス実装時のベストプラクティス

Web サービスの具体的な設計・実装レベルでの考慮点やノウハウについては**表 -3**，**4**，**5** に示すようにすでにさまざまなところ[2), 3)]で言及されています．ここではその一部を紹介します．

【項番 2】SOAP over HTTP をアプリケーション内の論理的なレイヤー間のやりとりに使ってはいけない

対象をいくつかの論理的な階層に分けて設計すること（これ自体 Layers パターンと呼ばれるアーキテクチャレベルのパターンの 1 つです）はアプリケーションの構造化の有効な手段です．このようなレイヤー間の連携を SOAP/XML で行うのは適切な設計と言えるでしょうか．物理的には同じ場所で動いているアプリケーションを論理的な観点からレイヤーに分割しているのだとすると，SOAP の使用はあまり意味がないことかもしれません．本来 SOAP/XML を使う最大の理由は異なる実装間の相互運用性（本連載 10 月号参照）のはずですが，同じアプリケーションサーバ（しばしば同一コンテナ）上で

番号	概要
1	小規模な Web サービスのセットから始めて要件に応じて拡張していく
2	SOAP over HTTP をアプリケーション内の論理的なレイヤー間のやりとりに使ってはいけない
3	再利用される可能性のあるアプリケーションモジュールは WSDL でインタフェースを記述する

表 -3　Web サービス適用の方針や範囲に関するベストプラクティス

番号	概要
4	ユースケース単位で Web サービスの粒度を決める．複数の細粒度のサービスはファサード (細粒度のインタフェースをまとめる玄関口の役割を果たすオブジェクト) で統合する
5	要件に応じて非同期の呼出しスタイルを検討する．同期式のリクエスト／リプライが当たり前と思ってはいけない
6	WSDL のインタフェースとインプリメントを分離して別々のファイルに記述しておく
7	複数の Web サービス間でプロセスの状態を共有したい場合は，サービスの呼出し側で状態を保持するか，あるいは BPEL4WS 等の標準が提供する機能を利用する
8	Web サービスのロケーション (URL) 等の情報は呼出し側でキャッシュしておく．呼び出しのたびに UDDI に問い合わせたりしない
9	メッセージスキーマの名前空間 URI には会社名や組織名を入れて他者との競合を避ける
10	参照やキャッシュの機能を活用して不必要に大きなサイズの XML メッセージを生成しない
11	WS-I のプロファイルに準拠する
12	DOM よりは SAX の XML パーサーを使う
13	バイナリデータの交換が必要な場合は SOAP Attachment か Base64 エンコーディングを使用する

表 -4　Web サービスの設計や実装に関するベストプラクティス

番号	概要
14	下位互換性のない変更 (オペレーションの削除／名前変更，パラメータの追加／削除／順番変更やデータ構造の変更) はバージョンではなく別 Web サービスとして管理せよ
15	メッセージフォーマットのバージョンの違いを名前空間 URI に反映させる
16	UDDI では bindingTemplate 中の tModelInstanceDetails コレクションを使って 1 つのサービスに対する複数のバージョンを表現する

表 -5　Web サービスの保守やバージョン管理に関するベストプラクティス

動いているレイヤー間のやりとりで SOAP を使って解決しなければならないような相互運用性の問題は発生しません．逆に XML と内部オブジェクトの間のシリアライズ／デシリアライズのために無視できないパフォーマンス低下が生じます．Web サービスは既存技術で実現できない点を補完する手段であり，既存技術による実装を置き換えることが可能だからといって必ずそうすべきだということではありません．

アプリケーション内のレイヤーが複数のアプリケーションサーバ上にまたがって存在している場合でも 2 つのサーバが J2EE など同一の実装基盤を提供しているのなら RMI/IIOP など SOAP over HTTP よりも効率が良くて相互運用性も確保できるプロトコルが存在します．セキュリティ上の理由でアプリケーションサーバが属するドメイン間で使えるプロトコルを制限している場合などには HTTP(S) に乗った SOAP の使用が有効です．しかし，特にファイアウォール関連の問題がないのであればあえて SOAP を使う理由が他にあるのか考えてみるべきでしょう．

【項番 3】再利用される可能性のあるアプリケーションモジュールは WSDL でインタフェースを記述する

一点，注意していただきたいのは項番 2 のベストプラクティスの主張はワイヤーフォーマットレベルでの Web サービス規格の使用に関する注意点であって，WSDL のようなインタフェース記述の規格の使用を否定するものではないということです．同一ドメイン内のサーバ間あるいは同一のサーバ上に存在しているアプリケーション同士であってもインタフェースの仕様を WSDL で記述し

ておくことで，

1）プログラミング言語等の実装に依存しない共通インタフェース記述の蓄積

2）アプリケーションの機能を利用するクライアントのコードのツールによる生成

というメリットを得ることができます．

【項番 15】メッセージフォーマットのバージョンの違いを名前空間 URI に反映させる

一般の業務アプリケーション同様，Web サービスも業務で使用しているうちにさまざまな改良が施されその仕様も少しずつ変化していきます．Web サービスでは提供者と利用者間の「縛り」が緩やかであるためサービスがバージョンアップしても利用者プログラムは古い Web サービスを呼び続けるといった状況が起こりやすくなっています．サービスの提供者は移行期間中は新旧両方のサービスを用意し適切なバージョンのサービスを利用者に提供する必要があります．

Web サービスのバージョンの違いをメッセージボキャブラリの名前空間 URI を使って実行時に識別することができます．提供者側は要求 XML メッセージの名前空間 URI を http://example.com/2005/02/14/buyChocolate といったバージョンを示す日付を含む形式で記述しておきます．サービス提供者は外部に対してはただ 1 つの URL を Web サービスの受付窓口として公開していますが，内部的には新旧異なるバージョンの Web サービスを異なる URL（異なるサーバ）で実装しています．新旧のメッセージは皆 1 つの公開 URL に送られてきますが，前述の ESB が XML メッセージの名前空間をチェックしてバージョンを判別し適切な内部 URL に要求をルーティングしてくれます．このようなメカニズムを用いることで同時に複数のバージョンの Web サービスをサポートすることが可能になり，新旧の移行を円滑に行うことができます．

おわりに

SOA の考え方自体は '90 年代半ばから言われていた一般的な設計原則であり決して新しいものではありません．最近になって SOA が注目されている原因の 1 つは実現手段として Web サービスとその関連ツールが整備されてきたことにありますが，残念ながら Web サービスの実装だけで SOA は実現できません．SOA の実現に必要な方法論やパターンにはさまざまなレベルがあり，今は SOA を推進するアーキテクトや先駆的な実践者の努力により蓄積・体系化が行われつつある段階にあるのだと思います．今後，これらが整備された時点で Web サービスや SOA は新たな普及の段階に入っていくことでしょう．

参考文献

1）Keen,M. et al.: Patterns: Implementing an SOA Using an Enterprise Service Bus, http://www.redbooks.ibm.com/redbooks/pdfs/sg246346.pdf

2）Endrei,M. et al.: Patterns: Service-Oriented Architecture and Web Services, http://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf

3）Brown,K. and Ellis,M.: Best Practices for Web Services Versioning, http://www-106.ibm.com/developerworks/webservices/library/ws-version/

（平成 17 年 1 月 5 日受付）