解説 1

SOAによるITアーキテクチャー(構造)の本質

市場にはSOAを題材にした解説が「ビジネスの視点」「テクノロジーの視点」「CEOの視点」「CIOの視点」など、さまざまな観点から発信されています。

しかしながら、「ITアーキテクトの視点」で解説している記事はそれほど多くありません。何よりもSOAに取り組む上で重要なのは、「サービスを指向したITシステムのアーキテクチャー」とは何なのかを理解することです。本稿では、「ITアーキテクトの視点」から、SOAによるITアプリケーションのアーキテクチャー(構造)やITインフラのアーキテクチャー(構造)を中心に説明します。国内におけるSOAの‘普及度’は残念ながらあまり高くありませんが、筆者はその理由を主に「SOAに基づくサービスの実装やアーキテクチャーに関する理解が今一歩進んでいないこと」と考え、ITアーキテクトがSOAに取り組む上でかなめとなる「サービス」と「アーキテクチャー」の2点に絞り、その本質を明らかにしていきます。

日本アイ・ビー・エム株式会社
ソフトウェア事業
ディスティングイッシュト・エンジニア(技術理事)
ITアーキテクト
長島 哲也 Tetsuya Nagashima

[プロフィール]
1978年 日本IBM入社。製造システム事業部にて、システムズエンジニアとして、お客様のシステム構築や安定運用をリード。1993年 ITアーキテクトとして、メインフレームからPCに至るまでのさまざまな技術・言語を担当する。1999 年テクニカル・サポートに移籍。その後e-ビジネス分野の先端プロジェクトの立ち上げに従事。
2003年ソフトウェア事業部に移籍。現在は、e-ビジネスシステム構築経験に基づくお客様サポートや各種コンソーシアム活動を通じてIT業界に貢献。2004年 EA & SOAのエバンジェリストに任命され、EA+SOAの普及活動を実施。

Article 1

Essence of SOA Based IT Architecture

There are a lot of articles in the market on Service-Oriented Architecture (SOA) , which covers a variety of perspectives, including the “business perspective,” the “technology perspective,” the “CEO perspective” and the “CIO perspective.” However, there are few articles which comment on the “IT architect perspective.” The most important point in “SOA” is understanding what is meant by the “structure (architecture) of Service-Oriented systems.” In this paper, we focus on the architecture (structure) of SOA-based IT applications and the architecture (structure) of SOA-based IT infrastructures from the “IT architecture perspective.” If we consider how little “penetration” of SOA there has been in Japan, I believe that the main reason is because “no progress has been made in understanding the implementation or architecture of SOA-based services.” In working on SOA, IT architects narrow their focus to the two key aspects of “service” and “architecture,” and they bring to light the true nature of SOA-based IT architecture.

❶ SOAの「サービス」とは何か

まずは、「サービス」についての解説から始めましょう。

図1に、SOA (Service Oriented Architecture) の最もシンプルな構成を示します。この図から、SOAには「コンシューマー」と「プロバイダー」という二つの“登場人物”が存在することが分かるでしょう。「コンシューマー」がサービスの利用者、「プロバイダー」がサービスの提供者に当たります。また、それぞれが「インターフェース」と「実装」に分離された構造を持つことも確認できます。

ここで重要となるのが、SOAでは、コンシューマーとプロバイダーのいずれもがIT(情報技術)の要素(換言すればプログラム)であることが前提になっているという点です。従って、コンシューマーがプロバイダーに対して「要求を出し、回答をもらう」という行為は、プログラム間の通信として行われます。この考え方は別段新しいものではなく、古くはCORBA (Common Object Request Broker Architecture)やDCOM(Distributed Component Object Model)で実現されていたRPC(Remote Procedure Call:

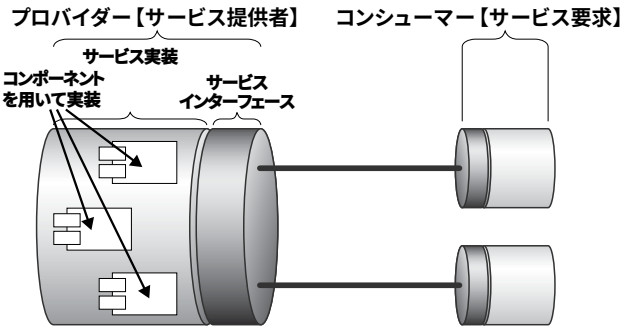

図1.　SOAの最もシンプルな構成

リモートプロシージャー呼び出し）などと同様です。

次に、インターフェースと実装に分離された構造ですが、SOAの「サービス」は一枚岩（モノリシック）の構造ではなく、必ずインターフェース（サービスインターフェース）と実装（サービス実装）に分離して作られます。これにより、コンシューマーがプロバイダーに要求を出す際、プロバイダーのサービスインターフェースさえ分かっていれば、そのサービス実装のことは気にせずに呼び出せるようになります。

もし、サービスインターフェースがなく、プロバイダーの実装（特定のミドルウェアやプログラミング言語、通信方式などに基づく実装）が“むき出し”になっていたとしましょう。その場合、コンシューマーがプロバイダーに要求を出すには、プロバイダーの実装について深く理解していなければなりません。SOAでは、サービスインターフェースとサービス実装を分離することにより、こうした制約を回避しているのです。

❷ サービスインターフェースとサービス実装の実体

ここまで説明した内容は、既にさまざまなメディアで解説されているので、違和感を持つ読者は少ないでしょう。しかし、筆者がお客様にこうした説明を行うと、必ずといってよいほど「では、SOAのサービスインターフェースやサービス実装の実体は何なのか。その点があいまいなので、SOAは分かりづらい」といった指摘を受けます。

ここで、図1の「サービス実装」の部分に注目してください。図に記したように、その実体は「アプリケーションコンポーネント」です。ここでいうアプリケーションコンポーネントとは、「内部にデータと状態を保持し、データにアクセスするためのインターフェースを持つプログラムの単位」を指します。

アプリケーションコンポーネントは、最終的に（アプリケーションサーバーなどの）ミドルウェアが提供するコンテナー上に、プログラミング言語を用いて実装され、稼働可能となるものです。従って、アプリケーションコンポーネントが持つインターフェース（コンポーネントインターフェース）は、通常、プログラミング言語や通信プロトコルに依存します。ということは、コンシューマーが直接アプリケーションコンポーネントに要求を出す場合、プログラミング言語や通信プロトコルの制約を受けるということになります。

SOAのサービスインターフェースとは、上記のような依存性をなくすために抽象化されたものです。これは、プログラミング言語・通信プロトコルに依存しない存在であり、業界標準のサービスインターフェース記述言語で定義されます。この記述言語としてはWSDL（Web Services Description Language：Webサービス記述言語）が、ご存じのように最有力です。また、2005年末ごろから、SOAのコンポーネント、すなわちサービスそのもの（以下、「サービスコンポーネント」と表す）を定義するための標準仕様として、SCA（Service Component Architecture）を策定する動きが活発化しています。

❸ サービスコンポーネントの構成

ここで、サービスコンポーネントについて少し説明しておきます（52ページ「SOAのテクノロジーと日本の環境」6章参照）。

図2に示すのが、SCAで定義されるサービスコンポーネントの構成です。

SCAではサービス群を配置可能な単位（Java™ EE：Java Platform, Enterprise EditionではEAR：Enterprise Archiveに相当）にパッキングします。この単位を「サービスモジュール」と呼びます。

一つのサービスモジュール内には複数のサービスコンポーネントを含めることができます（図2では「サービスコンポーネント-1」「サービスコンポーネント-2」と

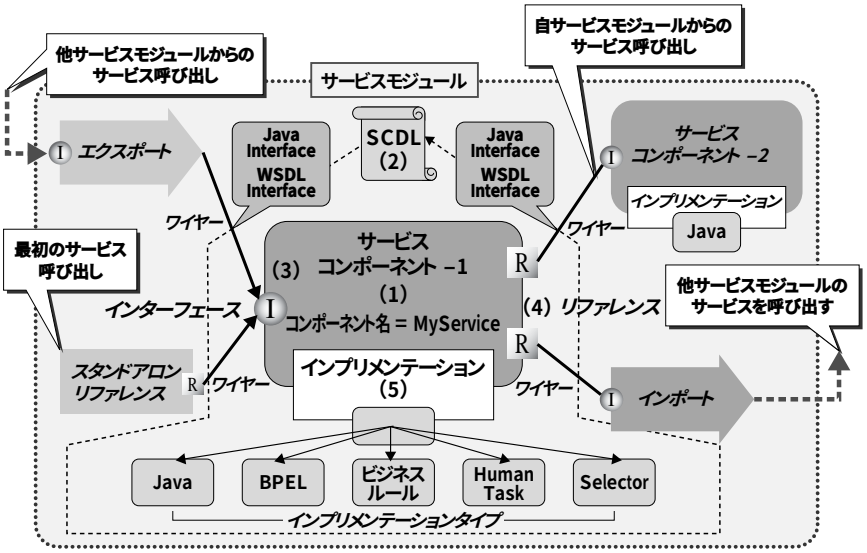

図2. サービスコンポーネントの構成

いう二つのサービスコンポーネントを含めています)。このとき、同一のサービスモジュール内に配置されたサービスコンポーネントは直接呼び出すことが可能ですが、異なるサービスモジュールへのサービス呼び出しでは、外部コンポーネントとやり取りするための部品(図の「インポート」と「エクスポート」)を介して連携することになります。

また、サービス化されていないアプリケーション(通常のJavaアプリケーションなど)からサービスコンポーネントを呼び出す際には、初回呼び出し時に「スタンドアロンリファレンス」と呼ばれる部品を経由します。

ここで、サービスコンポーネントとその周辺に注目してみましょう。サービスコンポーネントは、大別して以下の五つの要素から構成されます。

(1)サービスコンポーネント
(2)サービスコンポーネントの定義体。これはサービスコンポーネント記述言語(SCDL:Service Component Description Language)によって記述されます。
(3)インターフェース
(4)リファレンス(参照)
(5)インプリメンテーション(実装)

このうち(1)には、サービスコンポーネントの呼び出し方法、サービスの実装、サービスコンポーネントが呼び出すほかのサービスなどの情報が含まれます。

また(2)は、サービスコンポーネントに関する定義情報であり、これは、XML(Extensible Markup Language)形式で記述されます。

さらに、(3)はそのサービスコンポーネントを利用するためのインターフェース、(4)はほかのサービスコンポーネントへの参照です。これらは、WSDLインターフェースやJavaインターフェースに関連付けられます。

そして(5)は、図2に示すとおり、Javaや BPEL(Business Process Execution Language)などによる実装となります。

実は、上のようなサービスコンポーネントの構成そのものに、わたしたちITアーキテクトを悩ませる要因が潜んでいます。その要因とは、次のようなものです。

アプリケーションのサービス化にしても、コンポーネント化にしても、その目的は「ほかのプログラムへの依存性を弱め、再利用性を高めること」です。例えば、最近のオブジェクト指向開発では、DI(Dependency Injection)やAOP(Aspect Oriented Programming)といった“ アプリケーションコンポーネント間の依存性を弱める技術 ”が注目を集めています。

では、ITアーキテクトは、比較的大きな粒度のサービスコンポーネントを作る場合、上位のサービスコンポーネントを下位のサービスコンポーネントの複合(コンポジション)として実現するのが良いのでしょうか(複合サービス。**図3(1)**)、それとも「1対1」の関係でサービスコンポーネントとアプリケーションコンポーネントを配置し、上位のアプリケーションコンポーネントの複合として実現する方が良いのでしょうか(複合コンポーネント。**図3(2)**)。こうした疑問に対し、明確な回答が欲しいわけですが、統一的な見解はないのが現状です。

ちなみに筆者の見解は、最初にサービス化を検討する段階では、サービスコンポーネントとアプリケーションコンポーネントを「1対1」で実装し、下位のアプリケーションコンポーネント内でDIやAOPによる疎結合な複合関係を形成するのが現実的だと考えていま

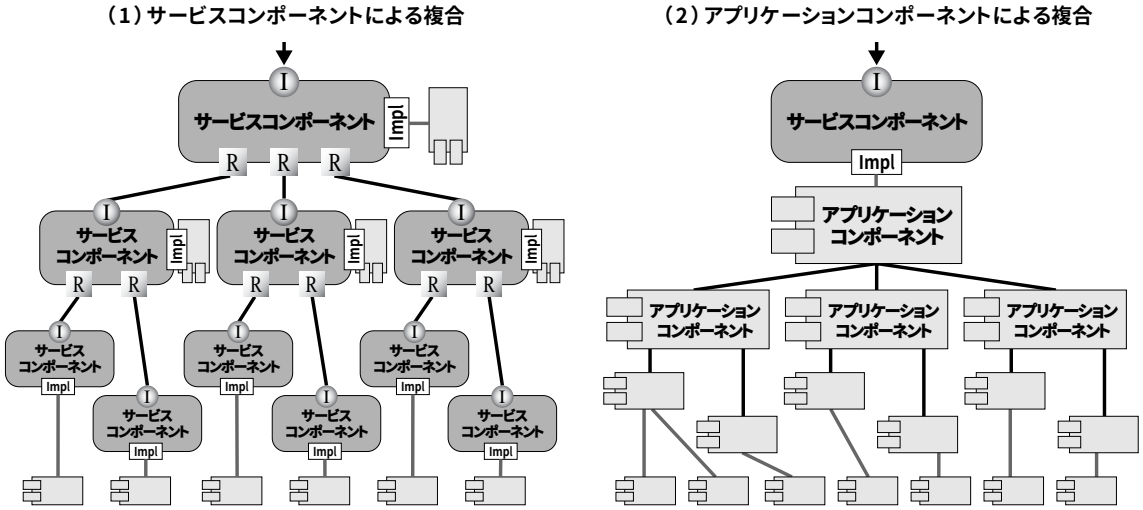

図3. 複合サービスと複合コンポーネント

す。そうすれば、呼び出し時のオーバーヘッドを最小化したり、一見同じように見えて、実は違う内容の処理に対応しやすいからです。

❹ SOAの「アーキテクチャー」とは何か

上記のような構造を持つ「サービス」を連携させてアプリケーションとして振る舞わせる「構造」を取ることが、SOAのもう一つの側面です。しかし、この構造には、一つ問題点があります。それは、サービスの物理的な提供場所を知らないと、連携することができないという点です。

例えば、グローバルに事業を展開する製造業の、全世界レベルの部品調達アプリケーションを想像してください。グローバルな部品調達では、各生産拠点での生産情報が必須となります。ある企業の生産拠点が、日本・北米・欧州の3カ所に存在するとしたら、「部品の全所要量」は、「日本の所要量サービス」→「北米の所要量サービス」→「欧州の所要量サービス」と順に問い合わせて求めることになります(**図4(1)**)。

昨今の製造業が置かれたビジネス環境では「コスト競争に打ち勝つために、組み立てコストの削減を達成しなければならない」という要求が常に存在します。そして、この要求に応えるために「欧州の生産拠点を閉鎖し、アジアに生産拠点を新設する」といったビジネス上の判断が下されることがあります。

その場合には、生産拠点の移転に伴い「サービスの物理的な提供場所の変更」という、ITシステムにも影響が及ぶ変更が生じます。その結果、「部品調達」の観点で見れば、「部品の全所要量」を求める処理の本質は変わっていないにもかかわらず、アプリケーションの修正が必要となります。なお、各拠点の生産情報提供サービスは、「調達」「原価」「物流」などのさまざまなシステムで共通サービスとして利用されることが多いので、生産拠点の移転がほかのシステムに与える影響は非常に大きなものになります。

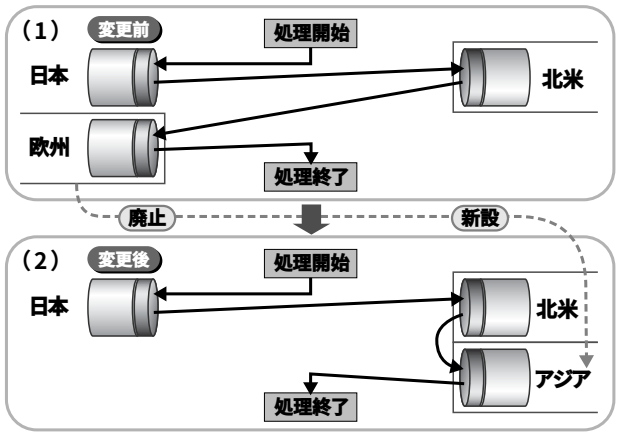

図4. サービス連携でアプリケーションの処理を進める

❺ ESBで「サービスの物理的な提供場所」を透過的にする

SOAでは、上記の問題点、すなわち「サービスの物理的な提供場所を知らなければ、連携処理を進められない」という問題を解決するために、ESB

（Enterprise Service Bus）を導入します。それにより、「サービスの物理的な提供場所（アウトバウンド・エンドポイント）」を透過的にするアーキテクチャーを取るのです（**図5(1)**）。そのため、ESBでは「サービスの物理的な提供場所」のほかに、「サービスの論理的な提供場所（インバウンド・エンドポイント）」と「サービスの論理的な提供場所と物理的な提供場所の関係」の三つを定義します。

ここで、ESBを用いない図4(1)のアーキテクチャーのシステムと、ESBを用いる図5(1)のアーキテクチャーのシステムに変更が生じたとしましょう。つまり、先の部品調達システムの例において「欧州の生産拠点を閉鎖し、アジアに生産拠点を新設する」というビジネス上の判断が下されたとします。この場合、それぞれのシステムには、**図4(2)**、**図5(2)**のような変更が発生することになります。

このとき、ESBを使っていない図4のシステムに関しては「サービスの物理的な提供場所」の変更に伴い、サービス連携の変更、すなわちシステムの変更が生じます。

一方、ESBを用いた図5のシステムでは、「サービスの物理的な提供場所」の変更に伴って「サービスの論理的な提供場所」を変更する必要はありません。「サービスの論理的な提供場所と物理的な提供場所の関係」を修正するだけで、つまりESB側の定義を変更するだけでよく、システムの変更は不要となります。

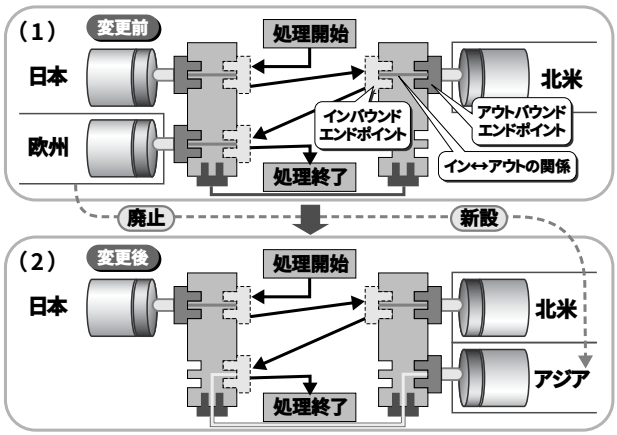

図5.　ESBを介したサービス連携

❻ SOAによるシステムの全体像

ここまで「サービス」と「アーキテクチャー」という二つの観点から解説してきましたが、再度SOAによるシステム全体の構成について考えてみましょう（**図6**）。なお、この図は論理的な構成を示しており、実際の配置構成は異なる場合もあります。

図6の最上位の「コンシューマー層」には、サービスの利用者、すなわちコンシューマーを配置します。先にも述べたように、SOAの世界では、コンシューマーは

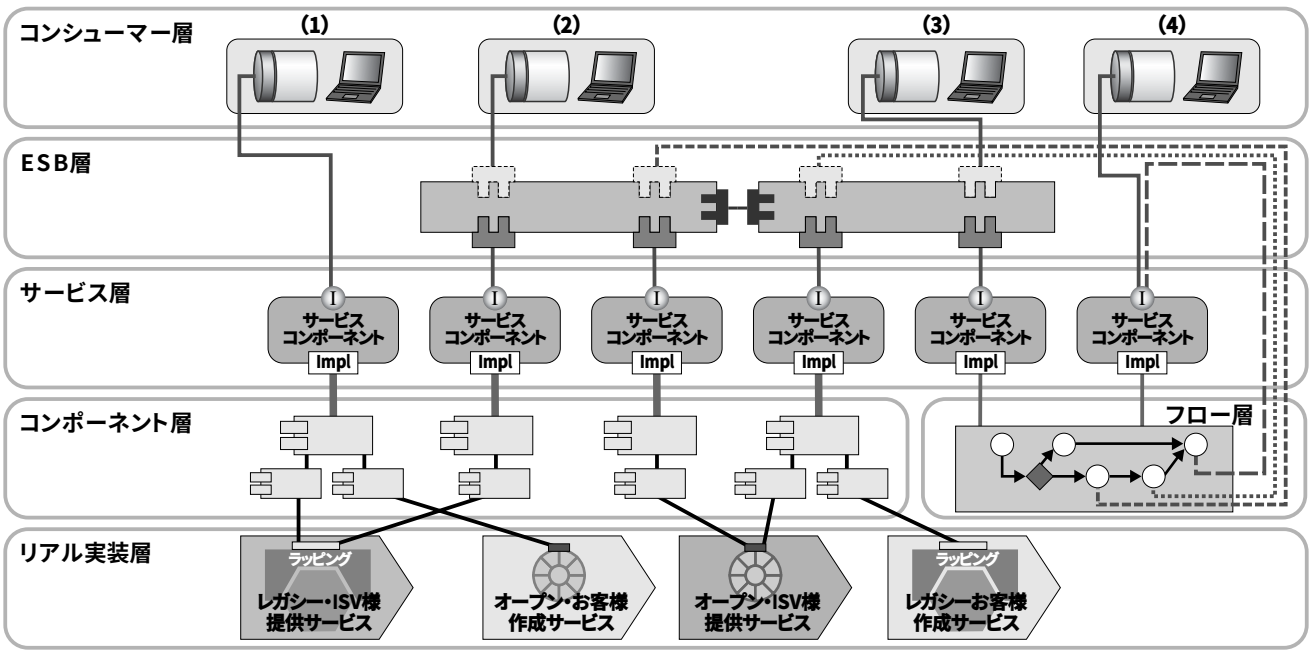

図6.　SOAを取り入れたITシステムの構造

プログラムです。従って、もしコンシューマー層にユーザーとの対話機能が必要になったら、対話をつかさどるIT資源を配置します（図6では、インターフェースと実装に分離されたドラム型の要求処理資源がこれに相当します。例えば、Webコンテナー上のStrutsアプリケーションが提供するWebブラウザーベースのユーザーインターフェースを利用する場合、Strutsのアクションクラスがこのドラムにマッピングされます）。

2階層目の「ESB層」に配置したESBの主要な目的は、前述したように「サービスの物理的な提供場所を透過的にすること」です。SOAによるシステムでは、直接サービスに要求を出さずに、ESB経由で要求を出すことが望まれます。ただし、システムの環境的な制約によっては、ESBを経由しないこともあります。例えば、インターネットを経由した緊密な関係にない取引先との接続では、物理的な提供場所を透過的にすることの意義が薄いため、ESBを介さずに直接接続するケースが多いようです。なお、ESBを使うと、コンシューマーとサービスの入出力形式が異なる場合に、それを調整・整形させることができます（これを「メディエーション」と呼びます）。

3階層目の「サービス層」には、サービスコンポーネントと、サービスの実装には依存しないサービスインターフェースを配置します。

4階層目の「コンポーネント層」には、サービスコンポーネントの実装としてアプリケーションコンポーネントを配置します。このアプリケーションコンポーネントは実体を持ち（プログラミング言語で書かれ、実際に動作します）、JavaであればEJB（Enterprise JavaBeans）のSession BeanやJavaBeansコンポーネントなどを用いて、単独または複合（コンポジション）構造として作られます。

なお、4階層目の一部には、サービスコンポーネントの実装として「フロー層」も置かれます。この層が、図4や図5で「処理開始」から「処理終了」までとして示したサービス連携をつかさどります。SOAでは、業界標準のBPELによるフロー定義に基づき、サービス連携が行われます。フロー定義から呼び出される処理はサービスであり、図6に示すように、フロー定義上の処理はESBまたはサービスコンポーネントのサービスインターフェースへと接続されます。

そして、一番下の5階層目に位置するのが、「リアル実装層」です。もし、アプリケーションコンポーネントをJava EE（旧名J2EE）などのオープンな分散技術上に構築するのならば、アプリケーションコンポーネント のJavaによる実装がリアル実装となりますし、ISV（Independent Software Vendor）が提供するパッケージ製品がアプリケーションコンポーネントのリアル実装になることもあります。また、COBOLやC言語などによるレガシーシステムをアプリケーションコンポーネントにする場合には、アダプター技術を用いてそれらをラッピングし、リアル実装にします。

最後に、コンシューマーからサービスを利用する方法について触れておきましょう。図6に幾つかの線で示したように、以下のようなパターンがあります。

(1) サービスコンポーネントを直接利用する。
(2) ESB経由でサービスコンポーネントを利用する。
(3) ESB経由でサービスコンポーネントの実装であるフロー定義を呼び出し、その先のサービスコンポーネントをESB経由で（または直接）利用する。
(4) 直接サービスコンポーネントの実装であるフロー定義を呼び出し、その先のサービスコンポーネントをESB経由で（または直接）利用する。

これら四つのパターンのうちどれを選択するかは、システムの形態によって異なります。この部分は、ITアーキテクチャーの設計指針によって決まるところです。

[参考文献]

[1] Dirk Krafzig, Karl Banke, and Dirk Slama：*Enterprise SOA*, PrenticeHall, ISBN 0-13-146575 -9（2004.11.9）.

[2] Dirk Krafzig, Karl Banke, and Dirk Slama著，山下真澄監訳：SOA大全, 日経BP社, ISBN4-8222-8250-3（2005.11.18）.

[3] 米持幸寿：基礎からわかるSOA, 日経BP社, ISBN4-8222-8230-9（2005.5.30）.

[4] David A Chappell著, 渡邊了介訳：エンタープライズ サービス バス, オライリー・ジャパン, ISBN4-87311-220-6（2005.2.8）.

[5] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F Ferguson著, 丸山宏, 上野憲一郎, 牧野有紀, 天野富夫, 鈴木豊太郎, 吉濱佐知子, 竹村司, 浦本直彦, 寺口正義共訳：Web サービス プラットフォーム アーキテクチャ, 株式会社エスアイビー・アクセス, ISBN4-434-07343-5（2006.2.1）.

[6] 和田 周：技術者の目で理解するSOA, 月刊JavaWorld 2005年4月号, IDGジャパン.