サービス指向アーキテクチャーとコンポーネント・ベース開発を使用したWebサービス・アプリケーション作成

Rational Software ホワイト・ペーパー

Alan Brown,
Simon Johnston,
Kevin Kelly

Rational®
the software development company

目次

インターフェース・ベース設計 5
インターフェースの動作 6
アプリケーション設計のレイヤー化 7
カスタマー・モデルの例 8
コンポーネント・ベース設計 9
サービス指向設計 9
サービス指向設計のキャッシュ 11
Web サービスの設計と実装のパターン 12
パフォーマンスと信頼性 13
非同期の動作とキューイングによる拡張性 14
情報リースの更新 15
結果的な Web サービス設計モデル 16

概要

全社的なソフトウェア・システムの構築は、複雑な作業です。数十年に渡って技術が進歩し続けてきたにも関わらず、今日の情報システムの必要に応えて、主幹業務ソフトウェア・ソリューションを設計、構築、および展開することは決して簡単ではありません。特に、新しいシステムをゼロから設計するケースはまれです。むしろ、ソフトウェア・アーキテクトの作業は、既存のデータ・リポジトリーを扱う新しいビジネス・ロジックを作成したり、既存のデータやトランザクションをインターネット・ブラウザーやハンドヘルド・デバイスなどの新しいチャネルで表示できるようにしたり、これまで接続していなかった、重複したビジネス・アクティビティーをサポートするシステムを統合したりして、既存のソリューションを引き続き使えるようにするのが一般的です。

ソフトウェア開発者を支援するため、Microsoft や IBM などのベンダーはソフトウェア・インフラストラクチャー製品を用意しています。それらのベンダーは、それぞれ .NET および WebSphere 製品ラインで推奨しているソフトウェア開発アプローチを中心に据えています。どちらのアプローチも、分散サービスからのシステムの組み立てに焦点を合わせています。しかし、サービスから全社的なソリューションを構築することには何か新しい点がありますか。コンポーネント・ベース・システムで習得したスキルは、どのようにサービス・ベースのアーキテクチャー (SOA) の構築に適用できるでしょうか。この新世代のソフトウェア・インフラストラクチャー製品をデプロイするための、高品質システムを構築する最適なアプローチとは何でしょうか。これらの重要な質問がこのホワイト・ペーパーの内容です。

近年のソフトウェア技術者団体の関心は、大規模なソフトウェア・システムを、それぞれ独立した再利用可能な機能の集合で組み立てるという概念をサポートする設計アプローチ、プロセス、およびツールに注がれてきました。機能のいくつかはすでに入手可能であり、社内で実装されているか、サード・パーティーから入手できますが、そうでないものは作成する必要があります。このような場合、これらすべての要素が矛盾なく 1 つとなるように、システム全体を想定し、設計する必要があります。今日、このことは*コンポーネント・ベース開発 (CBD)* で例証されています。これは、Microsoft の .NET プラットフォームや Java 2 Enterprise Edition (J2EE) 標準 (IBM の WebSphere や Sun の iPlanet などによってサポートされる) の技術的なアプローチで実現している概念です。

さらに、運用システムは通常、パフォーマンス、可用性、および拡張性を高めるために、多くのマシンに分散されるということを考慮に入れる必要があります。全社的なソリューションでは、一連のハードウェア上で実行される機能を組織する必要があります。このようなシステムを計画する 1 つの方法は、システムを対

話式の *サービス* の集合から構成すると考えることです。各サービスは、正しく定義された一連の機能にアクセスできます。システムは全体として、これらのサービス間の一連の対話として設計され実装されます。機能をサービスとして表すことが柔軟性の鍵となります。これにより、他の機能 (それ自体サービスとして実装される場合もある) が、物理的な場所に関係なく、他のサービスを自然な方法で使用できるようになります。システムは、新しいサービスが追加されることによって成長していきます。つまり、*サービス指向アーキテクチャー (SOA)* は、システムを構成するサービスを定義し、特定の動作を実現するためにサービス間で発生する対話を記述し、特定の技術による 1 つ以上の実装にサービスをマップします。

サービスがビジネス機能をカプセル化する場合、サービスの対話および通信を行うためにサービス間のいくつかの形態のインフラストラクチャーが必要です。サービスは 1 つのマシンにインプリメントしたり、ローカル・エリア・ネットワーク上の一群のコンピューターに分散したり、いくつかの会社のネットワーク上に幅広く分散したりできるため、このインフラストラクチャーにはさまざまな形態があります。特に興味深いのは、サービスが通信メカニズムとしてインターネットを使用する場合です。その場合、*Web サービス* は一般的なサービスの特性を共用しますが、サービス間の対話のために、セキュリティー・レベルおよび精度の低い公共のメカニズムを使用するため、特別な考慮が必要となります。

これまで、ソフトウェア業界の多くは、Web サービスとそれらの対話を実現するためにその基礎となる技術に注目していました。しかし、全社的なソリューションを簡単に組み立てるための Web サービスの最適な設計方法に関して、別の問題が生じました。Web サービスを構成する全社的なソフトウェア・ソリューションを設計するために必要なスキルやツールへの関心が著しく欠如していました。複雑な建造物の設計と同じように、高品質のソリューションは、多数の設計技術や構造パターン、およびスタイルを十分に理解した上で下される、早い段階からの体系的な決定によって生み出されます。こうした方法をとれば、拡張容易性、信頼性、およびセキュリティーなど、サービスに共通する問題を解決できます。

この資料では、全社的なソフトウェア・ソリューションのためのサービスおよびサービス指向アーキテクチャーに関する理解を深めるための背景情報を紹介します。特に、ソフトウェア・コンポーネントで確立された概念と関連付けながらサービスについて説明し、現在のコンポーネント・ベース開発で行っている方法が、サービス指向アーキテクチャーの実装のための実証済みの基礎を据えるためにどのように役立つかを紹介します。インターフェース・ベース設計を、サービス設計とコンポーネント設計両方に対する鍵として取り上げます。また、それらによって表されるインターフェースには、それぞれ異なる特定の制約や基準があることを説明します。コンポーネント設計とサービス設計の両方に関する特定のパターンだけでなく、論理設計と実装設計の両方を記述するためのツールとして統一モデリング言語 (UML) [1] を使用します。

最後に、このホワイト・ペーパーでは、一般的なサービスの実装に関係した問題を調べるための手段として Web サービスに焦点を当てます。特に、サービス指向アーキテクチャーの一般的な指針を、Web サービスの特定の設計パターンとして解釈する方法を紹介します。このホワイト・ペーパーに記述されているパターンはすべて、Rational® XDE™ の固有のパターンとコード・テンプレートの機能を使用して実装されており、developerWokrs: Rational[8] で公開されています。

サービス指向アーキテクチャーとは

サービス指向アーキテクチャー (SOA) とは何でしょうか。簡単に言えば、公開され検出可能なインターフェースにより、エンド・ユーザー・アプリケーションまたは他のサービスに対してサービスを提供する、ソフトウェア・システムの設計方法のことです。多くの場合、サービスはビジネスの機能を個別に表すための良い方法であり、ビジネス・プロセスをサポートするアプリケーションを開発するための優れた方法です。

サービス指向アーキテクチャーは新しい概念ではありませんが、Web サービス・テクノロジーの出現により、最近注目を集めています。例えば、サービス指向アーキテクチャーの価値について説明している資料 (2000 年出版) からの引用文を以下に紹介します。

> *サービス指向ソリューション… アプリケーションは、潜在的なユーザーに明確なインターフェースを提供する対話式の独立したサービスとして開発される。同様に、アプリケーション開発者がサービスの集合を参照したり、それらの中から必要なものを選択したり、それらを組み合わせて必要な機能を作成したりできるようにするための技術のサポートが必要となる。[2]*

この資料では、サービスに関する以下の定義について考えます。

> *サービスは、粗粒度の検出可能なソフトウェア・エンティティーとして実装され、1 つのインスタンスとして存在し、疎結合 (多くの場合、非同期) のメッセージ・ベース通信モデルによってアプリケーションおよび他のサービスと対話する。*

多くの場合、サービスの用語はコンポーネント・ベース開発の記述に使用される技術用語と良く似ていますが、Web サービスで要素を定義するために使用される特定の用語があります。図 1 をご覧ください。

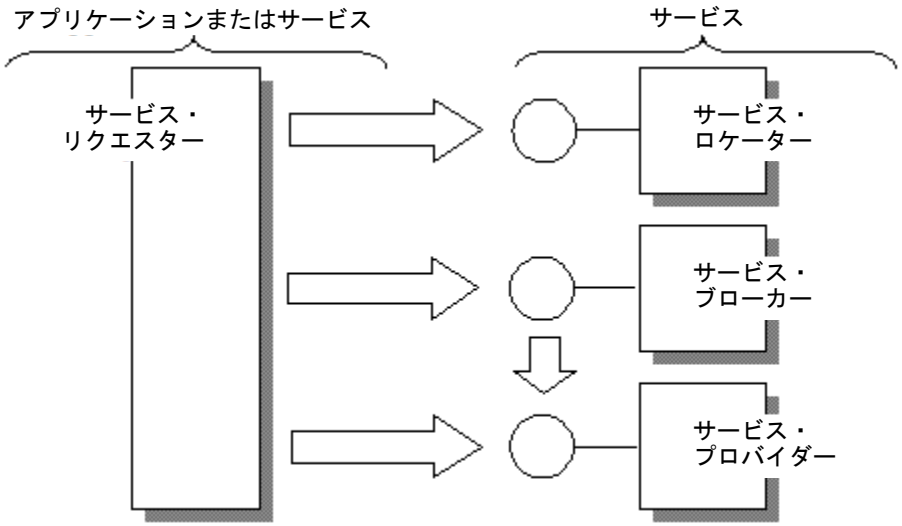

図 1 - サービスの用語

- **サービス** — 論理エンティティー。公開された 1 つ以上のインターフェースによって定義されるコントラクト。
- **サービス・プロバイダー** — サービスの仕様を実装するソフトウェア・エンティティー。
- **サービス・リクエスター** — サービス・プロバイダーを呼び出すソフトウェア・エンティティー。従来、これは「クライアント」と呼ばれてきましたが、サービス・リクエスターはエンド・ユーザー・アプリケーションまたは別のサービスになる場合があります。

- **サービス・ロケーター** — レジストリーとして機能する特定のサービス・プロバイダー。これにより、サービス・プロバイダー・インターフェースおよびサービス・ロケーションのルックアップが可能になります。
- **サービス・ブローカー** — サービス要求を、1 つ以上の他のサービス・プロバイダーに渡す特定のサービス・プロバイダー。

こうしたサービスの説明とその使用状況は、一連の制約となります。さらに、サービスを効率よく使用することによって、いくつかの高レベルのベスト・プラクティスが提案されます。サービスを効率よく使用するための鍵となる特性の一部を以下にあげます。

- **粗粒度** — コンポーネント・インターフェース設計と比較して、多くの機能を含み、大きなデータ・セットで機能するように、サービス上の操作は頻繁に実装されます。
- **インターフェース・ベース設計** — サービスは、定義されたインターフェースを個別に実装します。この利点は、複数のサービスが共通のインターフェースを実装でき、1 つのサービスが複数のインターフェースを実装できるということです。
- **検出可能** — サービスは設計時と実行時の両方で検出される必要があります。しかも、単に固有の ID によってだけではなく、インターフェース ID とサービスの種類によっても検出できなければなりません。
- **単一のインスタンス** — 必要に応じてコンポーネントをインスタンス化するコンポーネント・ベース開発とは異なり、各サービスが単一のインスタンスであり、常に実行されて多数のクライアントと通信します。
- **疎結合** — サービスは、XML 文書交換など、依存度や結合度の少ない標準のメッセージ・ベース・メソッドを使用して、他のサービスやクライアントと接続します。
- **非同期** — 一般に、サービスは非同期のメッセージ送受信アプローチを使用しますが、必須ではありません。実際、多くのサービスは同期的なメッセージ送受信アプローチを使用します。

インターフェース・ベース設計や検出可能性など、これらの基準の一部は、コンポーネント・ベース開発でも使用されます。しかし、これらの属性によって、J2EE や .NET などのコンポーネント・アーキテクチャーを使用して開発されるアプリケーションと、サービス・ベースのアプリケーションの違いが生じます。

インターフェース・ベース設計

コンポーネント開発とサービス開発のどちらにおいても、インターフェースの設計は、ソフトウェア・エンティティーを実装して、その定義の鍵となる部分を表現することによって行われます。そのため、コンポーネント・ベースのシステムとサービス指向のシステムのどちらにおいても、「インターフェース」の概念が設計を成功に導く鍵となります。以下に示すのが、インターフェースに関係した鍵となる定義の一部です。

- **インターフェース** — 論理的にグループ化された一連の public メソッド・シグニチャーを定義します。実装は行いません。インターフェースは、サービスのリクエスターとプロバイダーの間のコントラクトを定義します。インターフェースの実装ですべてのメソッドが提供されます。
- **公開済みインターフェース** — クライアントに登録されて動的に検出できるようになっている、固有に識別可能で使用可能なインターフェース。[3]
- **パブリック・インターフェース** — クライアントが使用できるが公開はされていないインターフェースのこと。そのため、クライアント側に一定の認識が必要です。
- **二重インターフェース** — 多くの場合、インターフェースは、1 つのインターフェースが別のインターフェースに依存するようにして組として開発されます。例えば、クライアント・インターフェースがいく

つかのコールバック・メカニズムを用意しているため、そのクライアントはリクエスターを呼び出すインターフェースを実装する必要があります。この概念は Web サービスによって取り入れられたものです。

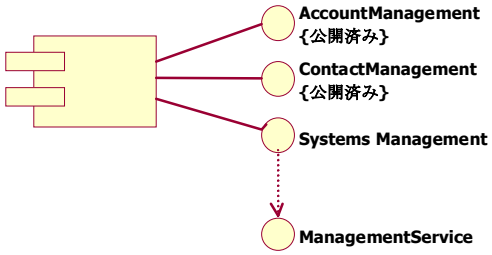

図 2 - 実装されたサービス

図 2 は、CRM (Customer Relationship Management) サービスの UML 定義を UML コンポーネントとして示しています。インターフェースとして、AccountManagement、ContactManagement、および SystemsManagement を実装します。最初の 2 つは公開済みインターフェースですが、最後のものはパブリック・インターフェースです。SystemsManagement インターフェースと ManagementService インターフェースは、二重インターフェースであることに注意してください。CRM サービスはこのようなインターフェースを任意の数だけ実装できます。このサービス (コンポーネント) の機能によって、クライアントに応じて複数の動作を可能にして、動作の実装を柔軟性のあるものにします。また、クライアントの特定のクラスに、異なるまたは追加のサービスを用意することもできます。実行時環境によっては、このような機能は 1 つのコンポーネントまたはサービスで、同じインターフェースの異なるバージョンをサポートするためにも使用されます。

インターフェースの動作

Java や C#、または IDL などの言語でのインターフェースの定義は、メソッドの署名を提供するだけです。この定義では、「方法」に関するガイダンスは与えず、「対象」のみを規定します。例えば、図 3 の「セキュリティー」インターフェースを考えてみましょう。このインターフェースの実装を呼び出すクライアントは、3 つの public メソッドのいずれかを呼び出せることがわかります。

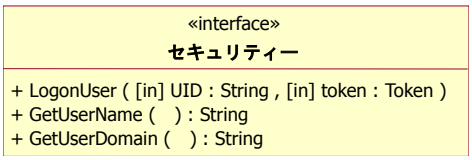

図 3 – UML でのインターフェース

対象を定義するだけでは、ユーザーがログオンするまでクライアントは GetUserName( ) または GetUserDomain( ) を呼び出すことができないという点を表現することはできません。以下に示す状態マシンは、この依存関係または動作を示しています。インターフェース・ベースの設計に関する資料には、この種の制約が含まれていることが多いのですが、制約をサポートするプログラミング言語はなく、インターフェースのインプリメンターが動作仕様に準拠していることは確認できません。

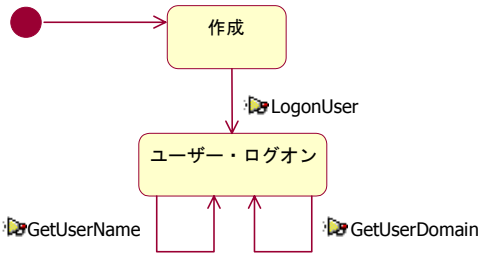

図 4 – インターフェースの動作

しかし、サービスのコラボレーションによってビジネス・プロセスを実現し、ビジネスをさらに簡単に統合および構成できることを目指して、ビジネスはさらにサービス指向システムに移行しつつあります。結果的に、インターフェースの動作を定義する概念と、さらに重要なこととして、関連する一連のインターフェースの動作を定義するという概念は、ますます業界の注目を集めています。残念ながら、現在までのところ、こうした概念に対する標準的なアプローチはわずかしかありません。

1 つのアプローチは、このホワイト・ペーパーで紹介するような設計モデルを使用することです。これは、UML のような標準化言語で定義され、サービス・インターフェース間の依存関係を記述します。そのようなモデルを共用、公開、使用して、特定の標準が現れたときに、それに合わせることができます。

さらに、Rational はこの問題に対応するため、資産をパッケージして共用するためのメカニズムを規定する RAS (Reusable Asset Specification) を支援してきました。例えば、RAS メカニズムを使用してサービスの詳細を分散する場合に、その動作を記述したモデルもパッケージに含めることができます。そのようなモデル内では、シーケンス線図を使用して、インターフェースでの呼び出し間で行われる必要な対話を示します。

サービス指向システムの設計

最新のソフトウェア開発技術では、以前のプロジェクトで使用したのと同じ技術やツールを流用することが非常に簡単になっています。コンポーネントとサービスは、似てはいるものの同じものではないことをすでに説明しました。これらは、設計基準や設計パターンの点で異なっています。このセクションでは、重要な点として実際の結果について説明します。*結論は、必ずしも優れたコンポーネントが良いサービスを提供するサービスに変換されるわけではないということです。*

アプリケーション設計のレイヤー化

新しい問題を時代遅れのソリューションで解決しようという傾向は新しいものではありません。同じように、開発者はコンポーネント・ベースのシステムの作成に取り組むときに、オブジェクト指向開発で培った経験で、同じような問題を解決しようとしてきました。多くの経験を通して、オブジェクト指向の技術と言語はコンポーネントを実装するすばらしい方法であることが分かりました。しかし、決定と実装によるトレードオフを理解しなければなりませんでした。多様な形態の動作を実装するために継承するか集約させるか、あるいは 1 つの C++ アプリケーションのための基本ではなく、一連のコンポーネントで使用できるクラス・ライブラリーを再設計するかに関してトレードオフが生じます。

同様に、典型的なコンポーネント・ベース・アプリケーションが典型的なサービス指向アプリケーションを作り出すわけではないことを理解する必要がありましたが、*コンポーネントはサービスを実装する最適な方法である* ことが分かりました。アプリケーション・アーキテクチャーにおけるサービスごとの役割分担を理解でき

れば、社内のコンポーネント開発者および既存のコンポーネントを活用する優れた機会を見いだすことなります。この転換を実現するために鍵となるのは、サービス指向のアプローチが追加のアプリケーション・アーキテクチャーのレイヤーであることを理解することです。以下に示す図 5 は、技術レイヤーがどのようにアプリケーション・アーキテクチャーに適用されるかを示し、アプリケーションの利用者に近づくにつれて、実装の粒度が粗くなっていくことを示しています。システムのこの面のことを「アプリケーション・エッジ」と呼びます。これは、サービスが、従来のコンポーネント設計を使用した内部での再使用および組み立てを活用するシステムの、外部から見た姿を表す最適な方法であることを表しています。

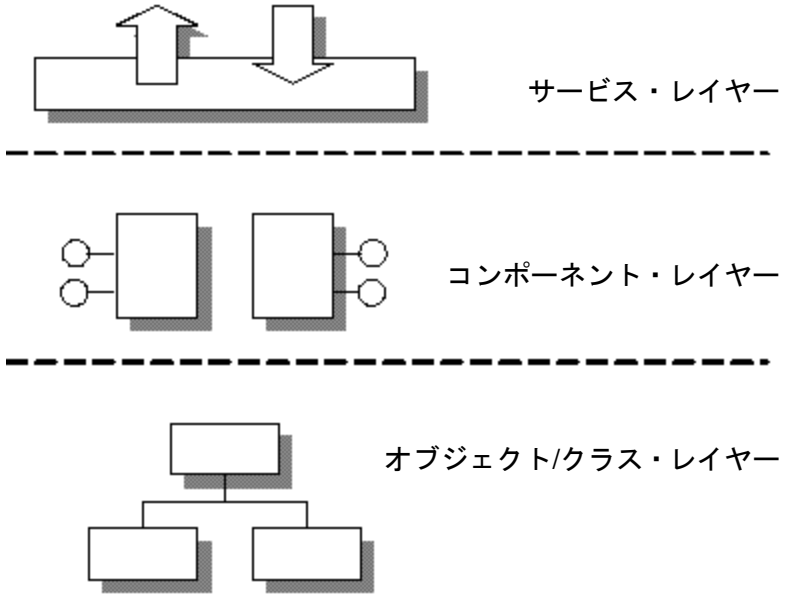

図 5 – アプリケーション実装レイヤー

一般に、オブジェクト指向からコンポーネント・ベースの考え方に移行するには、開発者はこの新しい技術とその実現に必要なものを学ぶために 6 カ月から 18 カ月かかります。うまくいけば、サービス指向システムへはさらに短期間で移行できます。それまでに、開発者は、サービス指向アプリケーションがサポートするコンポーネントの開発と再使用を考慮して、問題点、トレードオフ、および設計上の決定について理解する必要があります。

カスタマー・モデルの例

アプリケーションの実現においてコンポーネントとサービスがどのように対話するかを理解するために、以下の図 6 の UML クラス図で定義されている、カスタマー・リレーションシップ情報の管理の例について考えてみましょう。

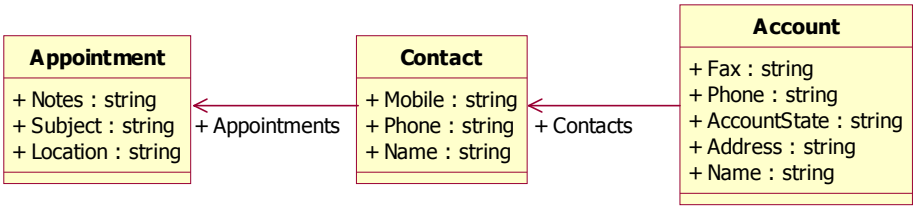

図 6 – 論理カスタマー・モデル

この後のセクションで、開発者がこのような論理モデルをコンポーネント・ベース・アプリケーションの実装モデルに書き換え、さらにサービス・ベースのアプリケーションの実装モデルに書き換える方法について説明します (このモデルはシステムの動作については全く示していないことにご注意ください)。これらの変換の多くは自動的に生成できることがお分かりになることでしょう。Rational Software では、アプリケーションのアーキテクチャーのモデル化、そのようなパターンの取得と適用、および開発ライフ・サイクル全体を通して得られるこれらのモデルまたはコードの成果物の管理などを行うツールを各種用意しています。

コンポーネント・ベース設計

論理モデルの例をどのようにコンポーネント設計 (COM または J2EE など) に変換できるのでしょうか。モデルのコンポーネント・ベース設計は、下記の図 7 のように表すことができます。

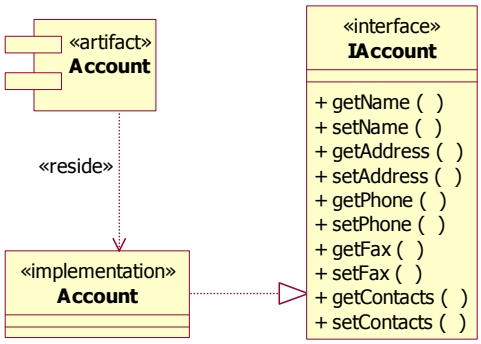

図 7 – 一般的なコンポーネントの図

上記のインターフェースには、*鍵* となる設計上の特徴があります。その鍵とは、共通のパターンを持つ既存のコンポーネント・プラットフォームのためのものです。分析クラスの各属性について、2 つの操作を用意する必要があります。1 つは値を設定する操作で、もう 1 つは値を戻す操作です。ローカル・コンポーネントの場合、メソッドの呼び出しのオーバーヘッドはほとんどなく、リモート・オブジェクトの場合、リモート・プロシージャー・コール (RPC) メカニズムが最適化されてオーバーヘッドは最小限に抑えられます。多くの場合、アプリケーションではクライアントはプロパティーのサブセットだけを必要とし、必要に応じてそれらにアクセスできます。

サービス指向設計

上記の設計モデルは、各コンポーネント・インスタンスが 1 つのオブジェクトを表す場合にコンポーネントの実装について考えるための正しい方法を示しています。例えば、CRM データベースの個々の連絡先は論理的に別個のコンポーネントとなります。それで、コンポーネントの ID は連絡先の ID と結びつけられます。

しかし、1 つのインスタンスが一連のリソースを管理するサービスの場合、それらのほとんどはステートレスになります。これはつまり、サービスを、タイプまたは一連のタイプのインスタンスを作成して管理できる マ

*ネージャー・オブジェクト*として見る必要があることを示しています。これは、インスタンスの状態を表す *値オブジェクト* (コンポーネント間の転送で状態が持続する分散システムの共通パターン)、実際に単純な連続した状態のオブジェクトを使用する設計パターンを生成します。この結果で興味深いのは、図 7 のモデルのような、コンポーネント定義を行い、それをサービスに変換する規則を定義できる場合に、この一連の流れをパターンとして実装できるということです。そのようなパターンの作成と再使用は Rational XDE を使用して行うことができます。

このプロバイダーからリクエスターへの状態の受け渡しには、多数の小さな操作でコンポーネントの状態を検索するのではなく、1 つの大きな操作が使用されます。これにより、大きな値オブジェクトを扱う場合のリクエスターの動作に加えて、リモート・サービス (ほとんどのサービスはリモートです) によるネットワークの使用状況も一定となります。これにより、一部のエンティティーの状態のコピーがリクエスターに提供されるという別の結果も生じますが、このコピーは古いものではないでしょうか。例えば、株価情報や気象情報を入手する場合、それらが最新の情報ではない可能性があることがわかっていてもそれらを受け入れます。また、データのタイプによっても条件は異なります。通常、株価情報のデータは気象データよりも頻繁に更新されます。ここで説明するように、そのようなアーキテクチャーでは、リクエスターは状態のコピーを受け入れる必要があります。詳細については、このホワイト・ペーパーの『粗粒度の対話の影響』のセクションを参照してください。

こうした条件がある中で、どのようなサービスを想定できるでしょうか。図 8 には部分的なモデルが示されており、そのようなパターンを設計レベルでどのように記述できるかを示しています。また、コンポーネントによって公開されるインターフェースとインターフェースが扱う値オブジェクトを示しています。

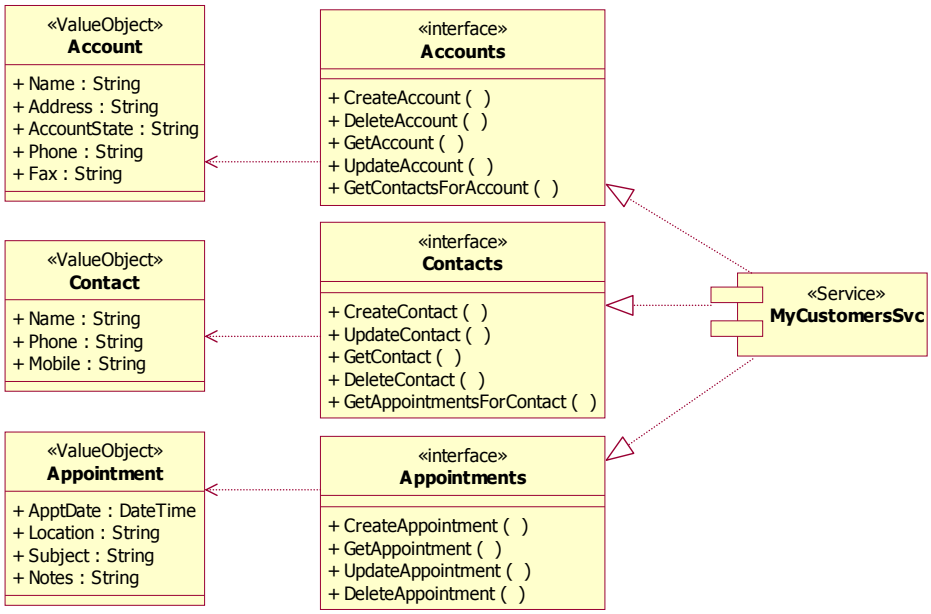

図 8 – 一般的なサービス指向設計

ここで、サービスをどのように設計できるかについて考えるために、この設計の属性のいくつかに注目してみましょう。始めに、値オブジェクトには、指定の対話に関して、プロバイダーである MyCustomerSvc からリクエスターへ、単なる「get」または「set」よりも多くの情報が渡されることは明らかです。さらに、これはネットワークの処理能力にも影響します。しかし、Web サービスの性質を考慮すると、サービスの実装に使用されるプロトコルは、コンポーネントの実装に使用されるプロトコルと全く異なります。このようなプラットフォ

ームでは、ネットワークに障害を起こさない範囲で、各値オブジェクトの内容が最大になるように、設計者あるいは情報技術者は注意深く値オブジェクトとその構成を選択する必要があります。

サービス指向設計のキャッシュ

前のセクションで取り上げた、プロバイダーからリクエスターに情報の「古い」コピーが渡されるという点を再び考えてみましょう。例えば、株式のポートフォリオ管理アプリケーションを開発する場合、証券の時価について、証券ごとに Web サービスに何度も照会し、3 文字から 5 文字のデータを渡して、5 文字から 7 文字の価格データが戻される、ということはしません。そうすることで、ネットワークとサービス・プロバイダーに限界を超えた負荷をかける場合があります。ここでリクエスターが行えるのは、ポートフォリオ全体の内容を要求することです。それは、記号のリストかポートフォリオの ID をサービスに渡して、各証券のすべての情報を取り出すことによって行えます。ここで、ユーザーが 1 つの記号に対する更新についてのみ問い合わせをすると、過剰なやり取りが発生するように思えるかもしれません。しかし、リクエスターはその結果をキャッシュできるため、ユーザーが別の記号に対して問い合わせをする場合、リクエスターはそのキャッシュからデータを返すことができます。ここで、リクエスターでは、データの「リース」期間が識別されます。それで、ポートフォリオの場合、株価情報サービスに 20 分の遅れがあることが分かっていれば、25% のマージンで作業し、結果を 5 分間キャッシュに入れるように設定できます。

これは、情報システムでよく見られるパターンです。ユーザーが注文管理システムから注文を取得する場合には、他のユーザーが同時にその注文を更新している可能性があるため、その注文のコピーを渡すのが実際的です (システムがその注文への追加のアクセスをロックしない場合)。これは、Web サービス・プロバイダーがリクエスターとの対話の一部として実際にキャッシュまたはリース期間を識別する場合に有効です。こうした問題は、Microsoft Message Queue Server (MSMQ) および IBM MQSeries などのメッセージング・システムでは良く知られた問題です。こうしたシステムでは、メッセージのタイムアウトや有効期限はルーチンとして管理されます。

この問題については後ほど再び取り上げて、リクエスターおよびプロバイダーを開発する場合にこの問題をどのように解決できるかに関する指針を紹介します。

XML Web サービスのアプリケーション設計

XML Web サービスは今や非常に身近なものとなり、さまざまなベンダーがサポートするようになっています。また、Web サービスの展開と開発をサポートするプラットフォームとツールも増えています。Web サービスを構成する多くの要素については、他の資料で十分に紹介されているため、このホワイト・ペーパーで詳しく取り上げることはしません。以下は、World Wide Web Consortium (W3C) の Web Services Architecture Working Group による定義からの引用です。

> *Web サービスとは、URI によって識別されるソフトウェア・アプリケーションのことであり、そのインターフェースとバインディングは XML の成果物によって定義、記述、および検出できる。さらに、インターネット・ベースのプロトコルを介して XML ベース・メッセージを使用して、他のソフトウェア・アプリケーションとの直接的な対話をサポートする。*[4]

現在 Web サービスで使用されている技術の一部が、サービス指向アーキテクチャー用語の説明で示した図 1 にどのように当てはまるかを考えてみましょう。

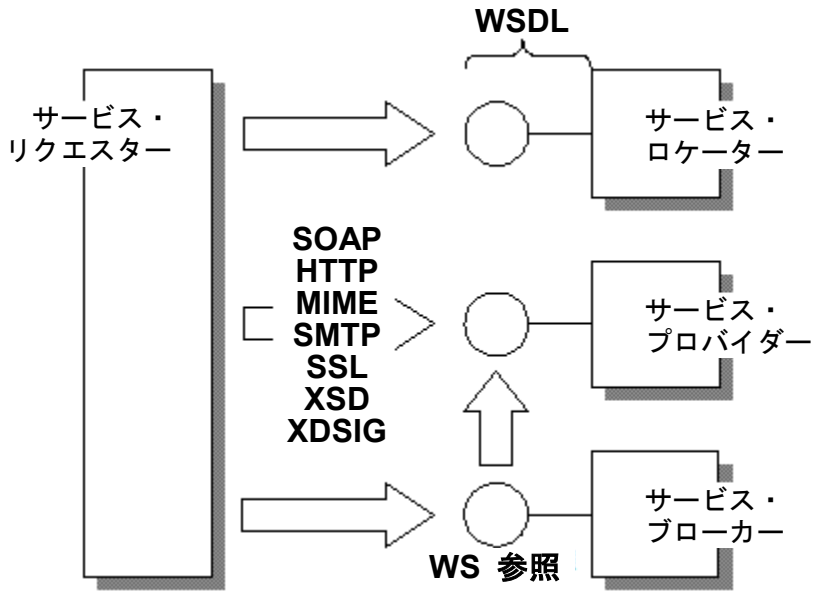

図 9 – XML Web サービスの標準

まず最初に、すべての Web サービスが、Web の事実上のプロトコルである、SOAP (Simple Object Access Protocol) over HTTP による XML メッセージを使用しているというのは誤解であることを知っておく必要があります。これは必ずしも真実ではありません。Web サービスのメッセージは XML を使用しますが、それはバイナリー・データを送付する場合です。一般的に SOAP ヘッダーを使用しますが、メッセージの本文に SOAP エンコード方式を使用する必要はありません。送信のために HTTP を使用する場合もありますが、SMTP や他のものの方が頻繁に使用されます。 For the description of, and discovery of, Web services there are two well-defined standards: WSDL (Web Services Definition Language) and UDDI (Universal Discovery, Description and Integration).

このフォーマットとトランスポート・プロトコルの柔軟性は、Web サービスの現在の課題の 1 つ (相互運用性) となっています。SOAP フォーマット、エンベロープ、トランスポート・プロトコルなどの選択が可能な上で、2 つの実装はどのように情報を交換できるのでしょうか。この問題に取り組んでいるのが WS-I (Web Services Interoperability) グループです。このグループは、現在の標準や新しく生まれる標準の使用に関する指針を業界に提供しています。ベンダーは柔軟性のある Web サービス開発環境を提供することによってこのグループを支援しています。例えば、IBM WebSphere Studio Application Developer Integration Edition では、複数のフォーマットとトランスポート・プロトコルを使用して Web サービスを作成することができ、必要に応じて高速で正しいセットが使用されます。

Web サービスの設計と実装のパターン

Web サービスは、実際にはアプリケーションの機能の要件に関する分析と設計の処理は変更しません。保険金請求を処理するアプリケーションは、引き続き保険金請求を処理します。ここでは、機能外要件における一連の制約と潜在的な問題について紹介します。この後のセクションでは、そのような潜在的な問題の一部について取り上げます。

パフォーマンスと信頼性

Web サービスのパフォーマンス、信頼性、および拡張容易性に必要とされる機能は、速度が遅く信頼性が低いとされる HTTP および SOAP に基づくアーキテクチャーで実現できるのかという質問はよく提起されます。まず最初に、「速度が遅く信頼性が低い」という点を明確にしてから、結局のところ、信頼性の高いトランスポートでさえ、信頼性の低い手段に基づいているということを認める必要があります。全社的なソリューションを設計する場合、機能要件と機能外要件を常に念頭におき、ビジネスの目的を達成するための正確なトレードオフと決定を確実に行えるようにする必要があります。

例えば、SOAP over HTTP を使用する場合、メッセージ確認やセキュリティーのための追加の機能を用意するアプリケーション・レベルのプロトコルおよび対話を作成することができます。しかし、特定のサービスが同じセキュリティーまたはアプリケーションのコンテキスト内で通信することを考慮する場合、HTTP 以外の手段の使用を検討します。図 10 の例を考えて見ましょう。

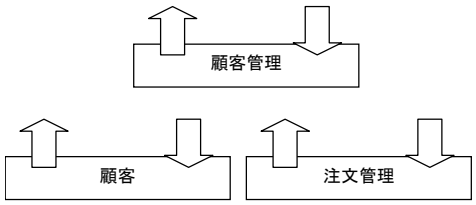

図 10 - サービスの依存関係

基本的に、外部クライアントはすべて顧客管理サービスと対話しますが、顧客管理サービスは 2 つの内部サービスと対話します。ここで問題となるのは、これらの内部サービスと通信するために HTTP と SOAP の柔軟性が必要なのはなぜかということです。顧客管理と顧客の間で行われる対話において、パフォーマンスがもっとも重要な要件であると仮定しましょう。その場合、バイナリー・エンコード形式を備えており、高いパフォーマンスを発揮するコンポーネント RPC 通信 (Microsoft .NET Remoting または Java の RMI など) の使用を決定するかもしれません。それに対して、顧客管理から注文管理への注文の受け渡しが確実に行われることが重要な要件である場合は、パフォーマンスと引き換えに高い信頼性を得るために、キューイング技術 (IBM MQSeries または MSMQ など) を使用するでしょう。

Web サービスが単純なモデルおよび単純で柔軟性のあるプロトコルのセットを提示する場合でも、それらしか選択できないわけではないことを覚えておくことはとても重要です。同時に WSDL には SOAP と HTTP GET/PUT の両方をバインドする機能があり、リクエスターにこれらの選択肢を与えることは重要です。例えば、ある 1 つのサービスがメッセージ・キュー・バインディングと SOAP バインディングを使用してメッセージを示す場合、リクエスターはそれらのどちらが使用するバインディングとして最適かを選択できます。このとき、プロバイダーはさらに、メッセージ・キューが使用される場合はサービス・レベルが保証されるが、HTTP による対話の場合はサービスの保証はないというような、選択の根拠となるものも提供する場合があります。

また、メッセージの交換を、べき等、可換、またはその両方のいずれにするかについても、設計の早い段階で決定しておくべきです。*べき等* にすると、同じメッセージを複数受け取り、それが *影響する* 場合でも、それぞれの場合に有害な影響は及びません。*可換* にすると、関連のある 2 つのメッセージが任意の順序で到着する場合でも、有害な影響は及びません。サービスの設計を、メッセージ交換が少なくともべき等として識別されるようにできる場合、信頼性の低いトランスポートでもさらに魅力のある (そして安価な) 選択肢となります。

セキュリティー (このホワイト・ペーパーでは論じていない) には、単純で安価なものから複雑で高価なものまで幅広い選択肢があるのと同様、パフォーマンス、信頼性、および拡張性などの設計目的によっても一連の決定が行われます。How much do you need? How much can you afford?サービスには、多くのソリューションがあり、それに伴って多くの選択肢があり、開発アプローチも多数存在するのです。

非同期の動作とキューイングによる拡張性

サービス指向アーキテクチャーの紹介で言及したように、Web サービスを非同期にすることには利点があります。Web サービスに関連した追加のトランスポートのオーバーヘッドと、サービスがその性質上リモートとなることが予想されることを踏まえて、リクエスターが応答を待つために費やす時間を抑えることは重要です。サービスの呼び出しを非同期にすることで、リターン・メッセージを個別にすることで、プロバイダーが応答可能な間はリクエスターも実行可能であるようにします。これは同期的なサービスの動作が間違いであることを示すものではありません。単に経験上、非同期的なサービスの動作が、特に通信コストが高いまたはネットワークの潜在的なトラフィックが予測できない環境では効率的であることが実証されているということです。

図 11 の例を考えて見ましょう。

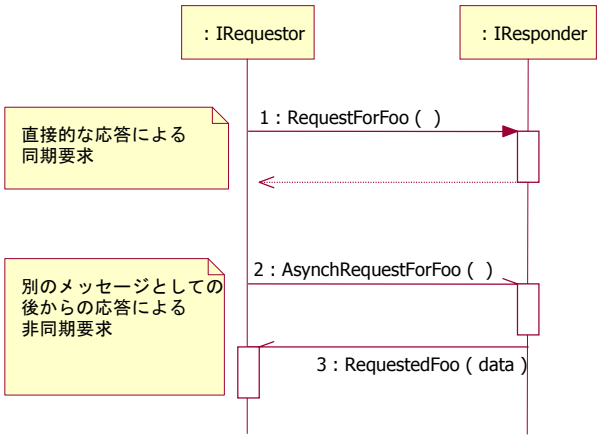

図 11 – 同期と非同期

図 11 に示されている動作は、拡張性の高い Web サービスを実装する点で非常に重要です。サービスの呼び出しを非同期にすることによって、プロバイダーは複数の作業者スレッドを使用して、複数のクライアント要求を扱うことができます。この場合、単にクライアントに早くデータを戻すというよりも、非同期モードの操作をサポートするために行うべきことが多数あります。まず最初に、二重のインターフェースを指定する必要があります。リクエスターは、戻されるメッセージを受け取るインターフェースを実装するサービスへ、戻りアドレスで渡す必要があります。これは、両者の間のやり取りの状態を管理する必要があることを示しています。これを行うさまざまな方法を知るためには、Web サービスに基づいていない Web セッションの設計に注目します。

しかし、これはある程度までの拡張性に限ってのことです。負荷が非常に高いことが予想されるサービスの場合、リクエスターを待機する部分と、要求そのものを行うサービスの部分を切り離す必要があります。これは既知のパターンであり、メッセージ・キューを使用して表面的なサービスとサービスの実装とを切り離します。図 12 は、キューを使用してプロバイダーを実装して、その実装から要求を切り離す方法を示しています。

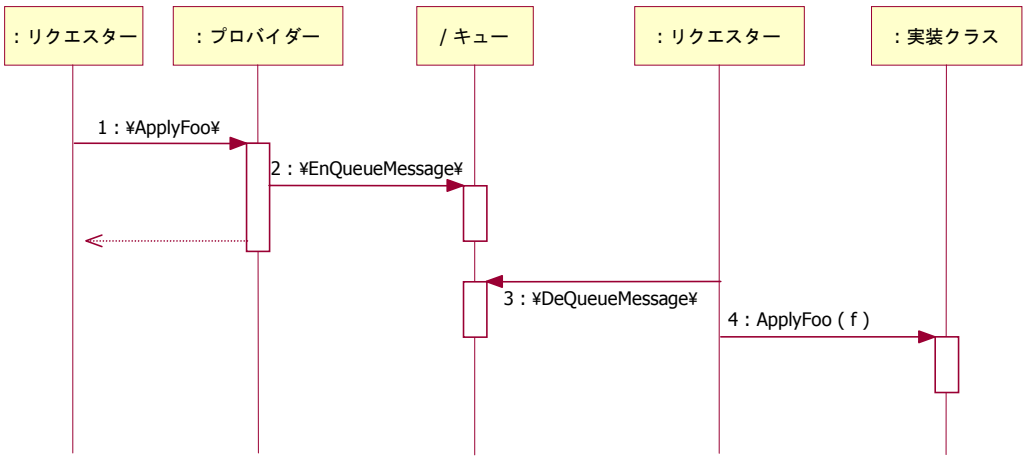

図 12 - キューを使用した実装

Such a pattern can be easily implemented in both .NET and J2EE using services provided by those platforms: MSMQ for .NET and Java Message Queue Service (JMS) or message-driven beans for J2EE.これにより、開発者には、要求を同期化して一連のスレッドを扱う場合よりも、簡単な拡張性のあるモデルが与えられます。この実装では、追加のキュー・リスナーを簡単に追加して、複数のマシンからではなく、特定のキューからメッセージを取り出すことができます。

情報リースの更新

情報のリースについて考える場合、家や車などの資産のリースよりも、図書館から本を借りる状況の方が分かりやすいでしょう。リクエスターがサービスに要求を出す場合、それは情報のコピーを要求していることになり、特定の時間の状態のスナップショットが提供されることになります。これを明確に理解して考慮に入れていないと、問題となる場合があります。1 つの方法として、プロバイダーに情報を提供する場合に *有効期限* も与えることができます。あるいは、リクエスターがリースに関する「チケット」(図書館の本と同様) を受け取る方法もあります。これによって、情報が有効である場合に依頼すればリースを延長でき、サーバーはデータを再度取得せずにリースを再設定することができます。

これは、HTTP、SOAP、またはトランスポート・プロトコルの 1 つが処理すると期待できるような基本的な問題であるように見えます。HTTP キャッシング・セマンティクスを再使用してブラウザーおよびファイアウォールがページをキャッシュするようにできますが、それはプロバイダーが制御できるものではなく、リクエスターは HTTP をトランスポートとして使用しない場合もあります。1 つの方法は、図 13 に示すように、そのようなサポートを文書交換の中に作成して、リクエスターとプロバイダーの間のメッセージにクライアントに関するリース情報をエンコードするというものです。

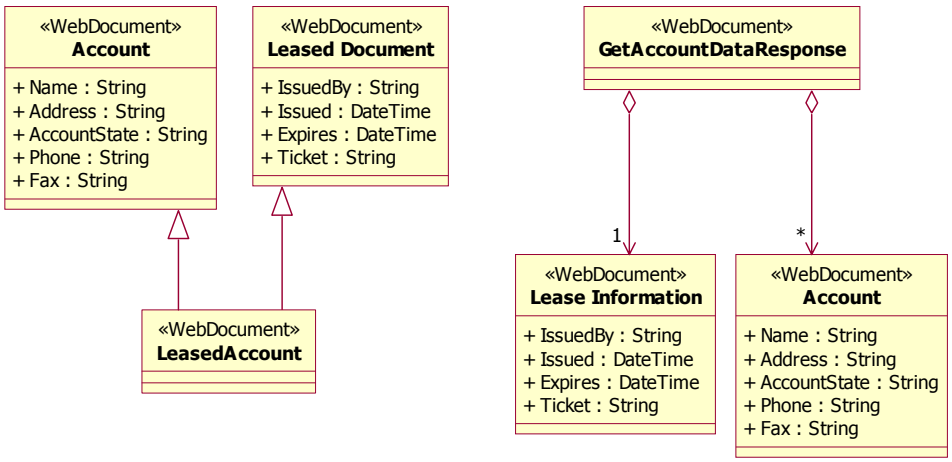

図 13 – 情報リースの 2 つの実装

図 13 は、情報リースのパターンに関する 2 つの実装について示しています。最初のものは、継承を使用して Account XML 文書を特定の形態に変える方法を示しています。この文書は、Account だけでなく Leased Document にも変換され、追加の情報が組み込まれます。2 番目のものは、リース情報とアカウントを、別個の応答メッセージとして戻します。これらのアプローチはどちらも有効なものですが、結果の構造データが異なります。これは、継承か集約のどちらを選ぶかという選択の問題です。

結果的な Web サービス設計モデル

図 14 は、図 7 の一般的なサービス設計を、Web サービス設計と開発に固有の UML プロファイルを使用してモデル化する方法を示しています。このプロファイルは非常にシンプルで、«WebService» と «WebDocument» のための 2 つの新しい*ステレオタイプ*(既存の UML 言語を拡張したもの) を採用しただけです。UML にすでに存在しているインターフェース・セマンティクスを再使用することによって、WSDL に定義されているように、サービスの公開されている面を簡単に視覚化することができます。

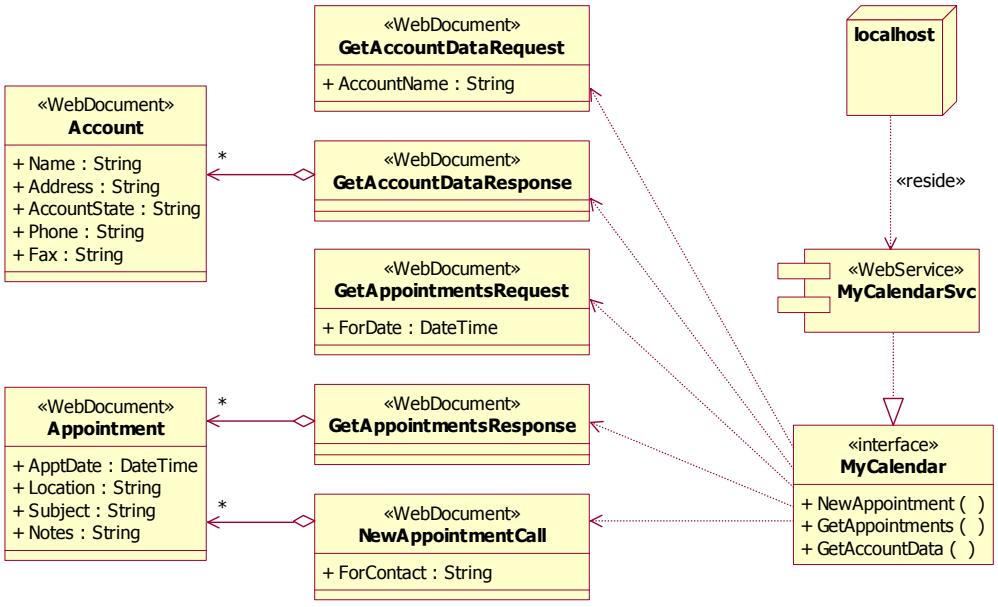

図 14 – XML Web サービス設計

Rational
the software development company

以下の表は、図 14 のプロファイルの要素が MyCalendar サービスの WSDL の成果物にどのように関連しているかを示しています。

WSDL 成果物	UML 要素	説明
サービス	«WebService»	サービスは、1 つ以上のインターフェースを実現する UML コンポーネントとして表され、特定の位置に置かれます。«reside» 関係により、実際の URL 位置情報が取り込まれます。
portType	インターフェース	各 portType は、1 つ以上のサービスを実現する UML インターフェースとして表されます。実現関係は、バインディング情報を取り込みます。
メッセージ	«WebDocument»	各メッセージは、UML クラスとして表されます。XML スキーマから UML へ、またはその逆のマッピングは、メッセージとパート構造体をモデル化する必要があります。
パート	属性または関連の終了	メッセージの各パートは、«WebDocument» の UML 属性、または別の «WebDocument» との関連として表されます。
アドレス位置	ノード	ノードは、サービスが存在するサーバーを表します。ノードは、一連の存在するサービスを識別し、サービスは複数のノードに存在する場合があります。

この Web サービス設計の方法は、データ交換を定義する文書と、サービスによってサポートされるインターフェースの両方の再使用を促進することに注意してください。これは、全社的なソリューションを設計する場合に重要な機能となります。Account 文書と対話するすべてのサービスは、可能な限り同じ文書定義に基づいて対話します。図 2 の SystemsManagement のような運用可能で非公開の操作インターフェースは、この分野の専門家が定義し利用可能にすることで、ソリューションを通じて共通の方法で実装できます。

結論

Web サービスはソフトウェア業界に革命を起こすと期待される一方で、そうした期待は過大であるいう声もあります。一般に、新技術に関する真実は、そうした意見の中間にあるものです。これまで見てきたように、サービス指向アプリケーションのアーキテクチャーにはコンポーネント・ベース開発のものと基本的に異なる点があります。しかし、既存のコンポーネント・ベース開発技術は、そのようなサービスの実装で引き続き有効です。

図 6 の Account、Contact、Appointment モデルに示されていたように、1 つのドメイン・モデルからコンポーネントとサービスの両方を実装することができます。さらに、結論として重要なのは以下の点です。

- サービス・モデルは*既存* のコンポーネント・ベース・モデルから派生できる
- サービス指向モデルへの変換のために共通のパターンと設計を適用できる

このような共通パターンは、Rational XDE などのモデルからモデルへの、およびモデルからコードへの変換ツールを使用することによって、設計の生成、自動化、およびインスタンス化することができます (以下の図に示されているように行うことができます)。

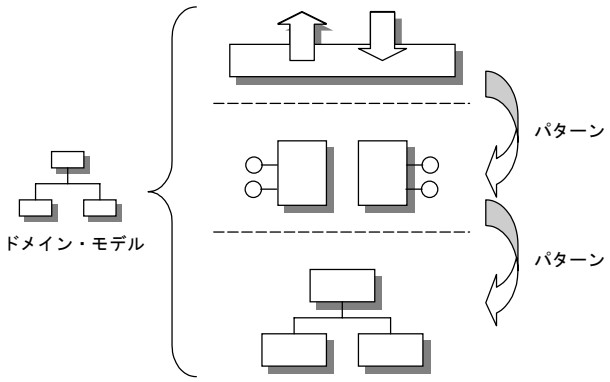

図 15 – サービス実装のアプローチ

この方法にはたくさんの利点があります。その中には、開発者の負担を軽くすること、実装の予測可能度を高めること、実証済みのパターンを再使用することによる品質の向上などがあります。専門知識に「サービス指向アーキテクチャーを使用する」ようなベスト・プラクティスを加え、さらにそれらのベスト・プラクティスを Rational XDE のような自動化ツール・セットで体系化することによって、サービスの実装に関する良い選択とトレードオフ決定を、素早く高い信頼性を持って行うことができるようになります。

参考資料

[1] For more information, see http://www.rational.com/uml/.
[2] 「Large-Scale, Component-Based Development」Alan W. Brown 著 Prentice Hall, 2000
[3] 「Public versus Published Interfaces」Martin Fowler 著 IEEE Software, March/April 2002 (Vol. 19, No. 2)
[4] W3C Web Services Architecture Requirements; http://www.w3.org/TR/2002/WD-wsa-reqs-20020429
[5] Web Services Security, Version 1,0; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-security.asp
[6] Web Services Referral Protocol, Draft; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-referral.asp
[7] XML Web Services Basics; http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-referral.asp
[8] Rational Developer Network http://www.rational.net/