WebSphere Process Server for Multiplatforms

バージョン 6.2.0

# モジュールの開発とデプロイ

WebSphere Process Server for Multiplatforms

バージョン 6.2.0

# モジュールの開発とデプロイ

**お願い**

本書に記載されている情報をご使用になる前に、本書末尾の特記事項セクションに記載されている情報をお読みください。

新しい版で明記されるまで、WebSphere Process Server for Multiplatforms バージョン 6、リリース 2、モディフィケーション 0 (製品番号 5724-L01) 以降のすべてのリリースとモディフィケーションが本書の対象となります。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

**原典：** WebSphere® Process Server for Multiplatforms
Version 6.2.0
Developing and Deploying Modules

**発行：** 日本アイ・ビー・エム株式会社

**担当：** トランスレーション・サービス・センター

第1刷 2009.1

**© Copyright International Business Machines Corporation 2005, 2008.**

# PDF ブックおよびインフォメーション・センター

PDF ブックは、印刷およびオフラインでの参照用に提供されています。最新情報は、オンラインのインフォメーション・センターを参照してください。

セットとして、PDF ブックには、インフォメーション・センターと同一の内容が含まれます。

PDF 資料は、バージョン 6.0 またはバージョン 6.1 など、インフォメーション・センターのメジャー・リリースの後の四半期以内にご利用いただけます。

PDF 資料の更新頻度は、インフォメーション・センターより低いですが、Redbooks® よりも頻繁に更新されます。通常、PDF ブックはブックに十分な変更が累積されたときに更新されます。

PDF ブックの外部にあるトピックへのリンクを選択すると、Web 上のインフォメーション・センターに移動します。PDF ブックの外部にあるターゲットへのリンクには、そのターゲットが PDF ブックと Web ページのどちらなのかを示すアイコンによるマークが付いています。

© Copyright IBM Corp. 2005, 2008

*表 1. 本書の外部にあるトピックへのリンクのプレフィックスとなるアイコン*

| アイコン | 説明 |
| --- | --- |
|  | インフォメーション・センターのページを含む、Web ページへのリンク。<br><br>インフォメーション・センターへのリンクは、ターゲット・トピックが新しい場所に移動した場合でもその機能を保つように、間接参照ルーティング・サービスを経由します。<br><br>ローカルのインフォメーション・センターでリンク先ページを見つけたい場合は、リンクのタイトルを検索することができます。あるいは、トピック ID を検索することもできます。検索の結果、タイプが異なる製品についてのトピックがいくつか見つかった場合は、検索結果の「**グループ別 (Group by)**」コントロールを使用して、表示するトピック・インスタンスを識別できます。以下に例を示します。<br>1. リンク URL をコピーします。例えば、リンクを右クリックして「リンク先をコピーする (Copy link location)」を選択します。例: `http://www14.software.ibm.com/webapp/wsbroker/redirect?version=wbpm620&product=wesb-dist&topic=tins_apply_service`<br>2. `&topic=` の後のトピック ID をコピーします。例: `tins_apply_service`<br>3. ローカル・インフォメーション・センターの検索フィールドに、トピック ID を貼り付けます。文書機能がローカルにインストールされている場合は、検索結果にそのトピックが表示されます。以下に例を示します。<br>`1 result(s) found for`<br>`Group by: None \| Platform \| Version \| Product`<br>`Show Summary`<br>`Update Installer を使用したフィックスパックおよびリフレッシュ・パックのインストール`<br>4. 検索結果のリンクをクリックしてトピックを表示します。 |
|  | PDF ブックへのリンク。 |

# 目次



© Copyright IBM Corp. 2005, 2008

# 図



© Copyright IBM Corp. 2005, 2008

# 表



© Copyright IBM Corp. 2005, 2008

# 第 1 部 アプリケーション開発

© Copyright IBM Corp. 2005, 2008

# 第 1 章 ビジネス・インテグレーション・ソリューションの開発

このセクションでは、ビジネス・インテグレーション・プログラミング・モデルの基礎について説明します。ここでは、Service Component Architecture (SCA) を紹介し、ビジネス・インテグレーションに関連するパターンについて説明します。

ビジネス・インテグレーションとは、企業がビジネス・プロセスを識別して統合し、最適化することを可能にする 1 つの専門分野です。その目標は、生産性を向上させ、組織の有効性を最大限にすることです。企業の吸収合併が進み、既存のさまざまな情報資産が増えていくにつれ、ビジネス・インテグレーションへの関心が強くなってきています。多くの場合、これらの資産には一貫性がなく、資産間で調整が取れていないことから、「情報の孤島」が次第に増えていきます。

ビジネス・インテグレーションには、ビジネス・プロセス・マネージメント (BPM) およびサービス指向アーキテクチャー (SOA) との強い結び付きがあります。企業の性質、そしてどの程度の統合が必要かによって、ビジネス・インテグレーションが IT 部門に課す要件は異なります。数少ない側面に対処すればいいだけのプロジェクトもあれば、これらの要件の多くを包含する大規模なプロジェクトもあります。例えば、ビジネス・インテグレーション・プロジェクトで最も一般的な側面には以下があります。

- **アプリケーションの統合**は、共通要件です。アプリケーション統合プロジェクトの複雑さは、場合によって異なります。単純な場合は、少数のアプリケーションが情報を確実に共用できるようにすればいいだけです。一方、複雑な場合には、複数のバックエンド・アプリケーションで、トランザクションとデータ交換が同時に反映されるようにしなければなりません。複雑なアプリケーション統合には大抵、複雑な作業単位の管理、そして変換とマッピングが必要になります。
- **プロセスの自動化**は、個人または組織が行うアクティビティーによって、結果的に別の場所で体系的にアクティビティーがトリガーされるようにする、もう 1 つの重要な側面です。これにより、ビジネス・プロセス全体が正常に完了することが確実になります。例えば、企業が従業員を雇用するときには、給与計算情報を更新し、セキュリティー部門が適切な措置を講じ、必要なツールを従業員に提供するなどの必要があります。プロセスのなかには、人間による入力と相互作用を取り込むアクティビティーもあれば、環境のなかのバックエンド・システムのスクリプトやその他のサービスを呼び出すアクティビティーが含まれることもあります。
- **接続**は、抽象的ながらも企業だけでなく、ビジネス・パートナーの点から見ても重要な側面です。ここで言う接続とは、組織間または企業間での情報のフローと、分散 IT サービスへのアクセス可能性の両方を意味します。

ビジネス・インテグレーション実装におけるいくつかの技術的課題は、以下のように要約できます。

- フォーマットが異なるために効率的なデータ変換を不可能にしているデータに対処すること
- 大幅に異なるテクノロジーを使用して開発された IT サービスにアクセスするための各種プロトコルとメカニズムに対処すること

© Copyright IBM Corp. 2005, 2008

- 地理的に分散されていたり、異なる組織によって提供されている各種 IT サービスの調整を取ること
- 使用可能なサービスを分類および管理するルールとメカニズムを提供すること (ガバナンス)

以上のように、ビジネス・インテグレーションには SOA にも共通する多くの主題とエレメントがあります。IBM のビジネス・インテグレーションの構想は、SOA で見られる基本的な概念の多くをベースに構築されています。この構想による直接の結果として、ビジネス・インテグレーション・ソリューションの実現には、さまざまな製品が必要となる可能性があります。IBM® では、さまざまなステージと操作の側面すべてをサポートするツールとランタイム・プラットフォームを豊富に揃えています。

簡単に言い換えると、IBM のビジネス・インテグレーションの構想によって、企業は SOA IT インフラストラクチャーで実行されるアプリケーションを使用して、ビジネス・プロセスを定義、作成、マージ、統合、そして簡素化できるようになります。ビジネス・インテグレーションの作業は、まさにロール・ベースです。マクロ・レベルでは、ビジネス・インテグレーションにはビジネス・プロセス・アプリケーションのモデル化、開発、ガバナンス、管理、およびモニターが必要です。適切なツールと手順の支援により、企業内外の人と異種のシステムが関係するビジネス・プロセスを自動化することが可能になります。ビジネス・インテグレーションの主要な側面の 1 つは、効率的でスケーラブルかつ信頼性に優れ、変更に対処できるだけの柔軟性を持つようにビジネス・オペレーションを最適化できるかどうかです。

ビジネス・インテグレーションには開発ツール、ランタイム・サーバー、モニター・ツール、サービス・リポジトリー、ツールキット、およびプロセス・テンプレートが必要になります。ビジネス・インテグレーションには多くの側面があるため、ソリューションを開発するには複数の開発ツールを使用する必要が出てきます。これらのツールにより、統合開発者が複雑なビジネス・ソリューションを組み立てることが可能になります。サーバーは、複雑なアプリケーションを実行するハイパフォーマンス・ビジネス・エンジンまたはサービス・コンテナーです。管理には常に、組織内で誰が何を行っているかを知る必要があります。そこで活躍するのが、モニター・ツールです。企業がこれらのビジネス・プロセスまたはサービスを作成するときには、サービスのガバナンス、分類、および保管が重要になってきます。この機能を提供するのが、サービス・リポジトリーです。レガシー・システムに対するコネクターやアダプターなど、ソリューションの特化した部分を作成するために、特定のツールキットが必要になることもよくあります。

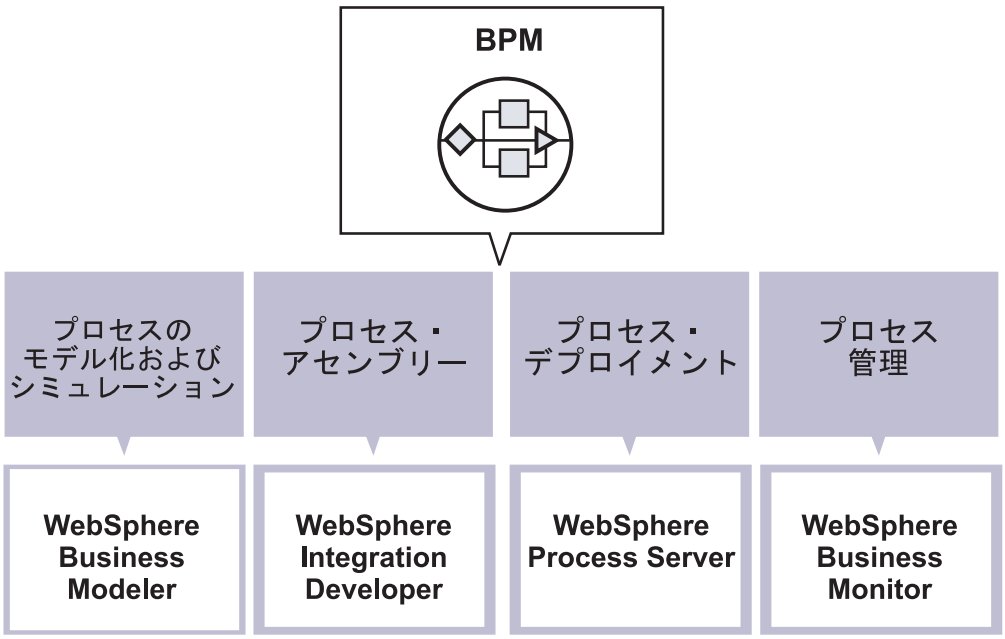



*図 1. IBM ツールは BPM ライフ・サイクル全体を網羅するため、プロセスのモデル化からアセンブル、デプロイ、および管理までを行うことができます。*

ビジネス・インテグレーションがベースとするのは、単一のプロジェクトではありません。ビジネス・インテグレーションには、ほぼすべての人、そして組織内および組織間のビジネスの側面すべてが関連します。ビジネス・インテグレーションには、SOA リファレンス・アーキテクチャー内のサービスとエレメントの多くが含まれます。

これらの概念の詳細およびプログラミング例については、以下を参照してください。

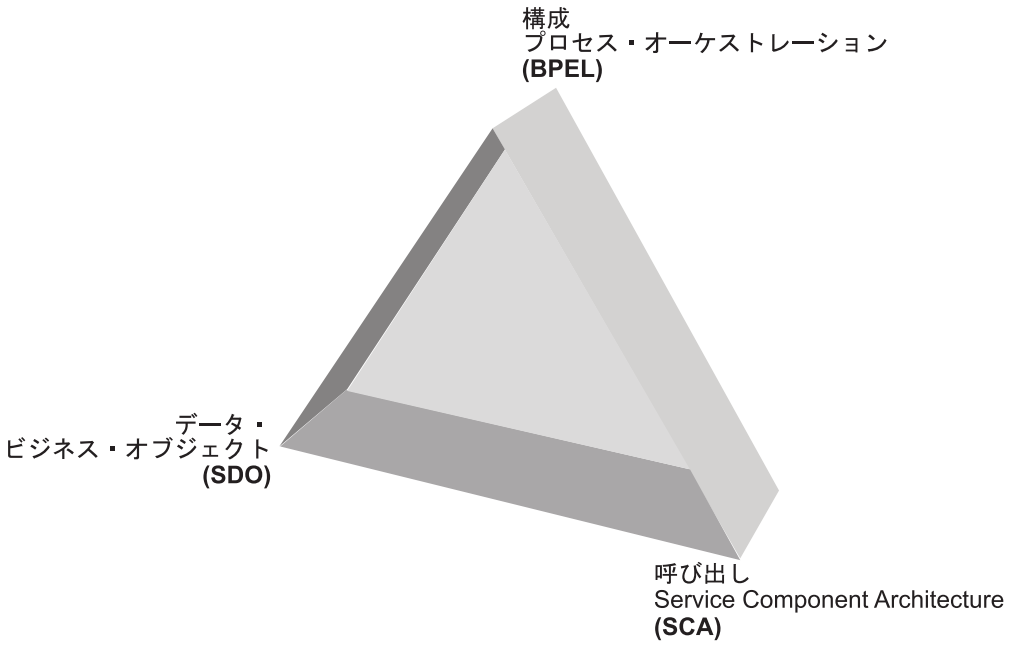
- 「*WebSphere® Business Integration Primer: Process Server, BPEL, SCA, and SOA*」(IBM Press、2008 年)
- 「*Getting Started with IBM WebSphere Process Server and IBM WebSphere Enterprise Service Bus Part 1: Development*」(IBM Redbooks、SG24-7608-00、2008 年 6 月)

## ビジネス・インテグレーションのプログラミング・モデル

ビジネス・インテグレーションは容易なタスクではありません。多種多様なテクノロジーがあり、データを表現する方法やデータと相互作用する方法がさまざまにあることが、統合のタスクを簡単に達成できなくさせています。データ、呼び出し、構成というプログラミング・モデルの 3 つの側面を捉え、サービスをベースとした手法の新しいパラダイムのいくつかを適用すると、SOA の新たなプログラミング・モデルが見えてきます。

第一に分かるのは、データは主に Extensible Markup Language (XML) によって表され、サービス・データ・オブジェクト (SDO) を使用するか、または XPath や XSLT (Extensible Stylesheet Language Transformation) などのネイティブ XML 機能を介してプログラムされることです。第二に、サービス呼び出しは Service Component Architecture (SCA) にマップされます。最後に、Business Process

Execution Language (BPEL) を使用したプロセス・オーケストレーションのなかに構成が統合されます。以下の図に、この新しいプログラミング・モデルの 3 つの側面を示します。




### Service Component Architecture

SCA は一貫性のある構文とメカニズムをサービス呼び出しに提供するだけでなく、呼び出しフレームワークとして、開発者にサービス実装を再使用可能コンポーネントにカプセル化する方法を提供します。これによって、開発者はテクノロジーにとらわれない方法でインターフェース、実装、参照を定義できるようになるため、エレメントを任意のテクノロジーにバインドすることが可能になります。SCA はビジネス・ロジックをインフラストラクチャーから分離し、アプリケーション・プログラマーがビジネス問題の解決に集中できるようにします。

## ビジネス・インテグレーションのアーキテクチャーおよびパターン

標準的なビジネス・インテグレーション・プロジェクトでは、数種類の IT 資産を調整する必要があります。これらの IT 資産はおそらく異なるプラットフォームで実行されていて、開発された時点も、開発に使用されたテクノロジーもさまざまに異なります。そのため、一連の多様なコンポーネントで、情報を簡単に操作して交換できるようにすることが、技術上の大きな課題となります。最適な対処方法は、ビジネス・インテグレーション・ソリューションの開発に使用されるプログラミング・モデルを使用することです。

このセクションでは、Service Component Architecture (SCA) を紹介し、ビジネス・インテグレーションに関連するパターンについて説明します。パターンは私たちの生活に浸透しています。裁縫のパターン、子供たちの学習パターン、住宅建設のパターン、木彫パターン、飛行パターン、風のパターン、医療活動パターン、顧客の購買パターン、ワークフロー・パターン、情報科学のデザイン・パターンなど、パターンはさまざまなところに存在します。

パターンがソリューションの設計者と開発者に役立つことは実証されています。したがって、ビジネス・インテグレーションとエンタープライズ・インテグレーションにパターンを使用するのも当然です。要求と応答のルーティング・パターン、チャネル・パターン (パブリッシュ/サブスクライブなど) をはじめ、ビジネス・インテグレーションに適用できるパターンは幅広くあります。抽象パターンは特定の問題カテゴリーを解決する際のテンプレートとなる一方、具体的パターンは固有のソリューションを実装する方法を具体的に指示します。このセクションでは、データおよびサービス呼び出しに対処するパターンに焦点を当てます。WebSphere ビジネス・インテグレーションに対する IBM ソフトウェア戦略では、これらのパターンをプログラミング・モデルの基盤としています。

## ビジネス・インテグレーションのシナリオ

企業はさまざまなソフトウェア・システムを使用してビジネスを運営します。さらに、これらのビジネス・コンポーネントは企業独自の方法で統合されます。

最も一般的なビジネス・プロセス統合のシナリオには、以下の 2 つがあります。

- **統合ブローカー**: このシナリオでは、ビジネス・インテグレーション・ソリューションが各種「バックエンド」アプリケーションの間に位置する仲介の役割を果たします。例えば、顧客がオンライン注文管理アプリケーションを使用して注文を行うと、トランザクションがカスタマー・リレーションシップ・マネージメント (CRM) バックエンドで関連する情報を更新するようにしなければなりません。このシナリオでは、統合ソリューションが注文管理アプリケーションから必要な情報を取り込み、場合によっては情報を変換した上で、CRM アプリケーションの適切なサービスを呼び出すことが可能でなければなりません。
- **プロセス自動化**: このシナリオでは、統合ソリューションが関連付けられていない異なる IT サービス間のグルーとしての役割を果たします。例えば、企業が従業員を雇用する場合、以下の一連のアクションが発生する必要があります。
  - 従業員の情報を給与計算システムに追加します。
  - 従業員に設備への物理アクセスを認可し、バッジを提供する必要があります。
  - 企業が一連の物理資産 (オフィスのスペース、コンピューターなど) を従業員に割り当てる必要がある場合もあります。
  - IT 部門が従業員のユーザー・プロファイルを作成し、一連のアプリケーションへのアクセスを認可する必要があります。

    このプロセスの自動化も、ビジネス・インテグレーションのシナリオでは一般的なユース・ケースです。このシナリオでは、ソリューションが実装する自動フローは、給与計算システムへの従業員の追加によってトリガーされます。続いて、このフローはアクションを実行する担当者の作業項目を作成するか、または適切なサービスを呼び出すことによって、他のステップをトリガーします。

いずれのシナリオでも、統合ソリューションは以下のことを行う必要があります。

1. さまざまな情報ソースと各種のデータ・フォーマットを操作し、情報のフォーマット変換を行えるようにする。
2. 異なる呼び出しメカニズムとプロトコルを使用する各種のサービスを呼び出せるようにする。

## 役割、製品、および技術に関する課題

ビジネス・インテグレーション・プロジェクトの成功は、特化した開発役割、プログラミング手法、およびツール・スイートの組み合わせによって左右されます。

ビジネス・インテグレーション・プロジェクトには、以下の基本要素が必要です。

- 開発組織内での明確な役割の分離。通常、役割を明確に分離して特殊化を促進することで、開発される個々のコンポーネントの品質が改善されます。
- 共通ビジネス・オブジェクト (BO) モデル。共通の論理モデルでビジネス情報を表現することを可能にします。
- プログラミング・モデル。インターフェースを実装から明確に分離するとともに、実装とは完全に独立し、インターフェースの操作だけが必要な汎用サービス呼び出しメカニズムをサポートします。
- 統合されたツールと製品のセット。開発役割をサポートし、それぞれの役割の分離を維持します。

以降のセクションでは、上記の構成要素のそれぞれについて詳しく説明します。

### 明確な役割の分離

ビジネス・インテグレーション・プロジェクトには、互いに協調しながらも明確に分離した役割を持つ担当者が必要です。これらの役割には以下があります。

- **ビジネス・アナリスト**: ビジネス・アナリストはドメイン・エキスパートとして、プロセスのビジネスの側面を捉え、プロセス自体を的確に表現するプロセス・モデルを作成します。この役割が焦点とするのは、プロセスの財務的パフォーマンスを最適化することです。ビジネス・アナリストは、プロセス実装の技術的な側面には関与しません。
- **コンポーネント開発者**: コンポーネント開発者は、個々のサービスとコンポーネントの実装を担当します。この役割が焦点とするのは、実装に使用する特定のテクノロジーです。この役割には、プログラミングの経歴を積んでいることが必要となります。
- **統合スペシャリスト**: これは比較的新しい役割で、この役割が表すのは、既存の一連のコンポーネントをまとめて 1 つの大きなビジネス・インテグレーション・ソリューションにアセンブルするという職務を課せられた担当者です。統合開発者は、再使用およびワイヤリングするコンポーネントやサービスそれぞれの技術詳細を知る必要はありません。理想的には、統合開発者は、アセンブルしているサービスのインターフェースについて理解することだけに専念します。統合開発者は、アセンブル・プロセス用の統合ツールに依存する必要があります。
- **ソリューション・デプロイヤー**: ソリューション・デプロイヤーおよび管理者が対象とするのは、エンド・ユーザーが使用可能なビジネス・インテグレーション・ソリューションの作成です。理想的には、ソリューション・デプロイヤーは主にソリューションを作動準備の整った物理リソース (データベース、キュー・マネージャーなど) にバインドすることに専念し、ソリューションの内部構造を深く理解することはしません。ソリューション・デプロイヤーが焦点とするのは、サービスの品質 (QoS) です。

### 共通ビジネス・オブジェクト・モデル

前述のとおり、ビジネス・インテグレーション・プロジェクトの重要な側面には、複数のコンポーネントの呼び出しを調整できること、そしてこれらのコンポーネントの間でのデータ交換を処理できることが含まれます。これは特に、複数のコンポーネントがそれぞれに異なる手法で、注文や顧客情報に含まれるデータなどのビジネス・アイテムを表現する可能性があるからです。例えば、Enterprise Java™Bean (EJB) エンティティーを使用してビジネス・アイテムを表す Java アプリケーションと、COBOL コピーブックで情報を編成しているレガシー・アプリケーションを統合しなければならない場合も考えられます。そのため、統合ソリューションの作成を単純化することを意図するプラットフォームは、バックエンド・システムがデータ処理に使用する手法とは関係なくビジネス・アイテムを表現する汎用方法も提供しなければなりません。この目標は、WebSphere Process Server および WebSphere Enterprise Service Bus では、*ビジネス・オブジェクト・フレームワーク* によって達成されています。

ビジネス・オブジェクト・フレームワークは、開発者が XML スキーマを使用してビジネス・データの構造を定義し、これらのデータ構造 (ビジネス・オブジェクト) のインスタンスに XPath または Java コードでアクセスして操作することを可能にします。ビジネス・オブジェクト・フレームワークは、サービス・データ・オブジェクト (SDO) 標準に基づきます。

### Service Component Architecture (SCA) プログラミング・モデル

SCA プログラミング・モデルは、WebSphere Process Server および WebSphere Enterprise Service Bus で開発されるすべてのソリューションの基盤となります。SCA は、開発者にサービス実装を再使用可能コンポーネントにカプセル化する方法を提供します。これによって、開発者はテクノロジーにとらわれない方法でインターフェース、実装、参照を定義できるようになるため、エレメントを任意のテクノロジーにバインドすることが可能になります。また、これらのコンポーネントの呼び出しを可能にする SCA クライアント・プログラミング・モデルもあります。このモデルは具体的には、Java ベースのランタイム・インフラストラクチャーが Java 以外のランタイムと対話することを可能にします。SCA は、サービスを呼び出すためにビジネス・オブジェクトをデータ項目として使用します。

### ツールおよび製品

IBM WebSphere Integration Developer は、上述のテクノロジーをベースとしたビジネス・インテグレーション・ソリューションを作成および構成するために必要なすべてのツールを備えた統合開発環境です。これらのソリューションは通常、WebSphere Process Server にデプロイされますが、場合によっては WebSphere Enterprise Service Bus にデプロイされることもあります。

## ビジネス・オブジェクト・フレームワーク

コンピューター・ソフトウェア業界は、開発者がビジネス・オブジェクト (BO) 情報をカプセル化することを可能にする複数のプログラミング・モデルとフレームワークを開発してきました。一般に、BO フレームワークに必要とされるのは、データベースの独立性を提供し、カスタム・ビジネス・オブジェクトをデータベース表またはエンタープライズ情報システム内のデータ構造に透過的にマップし、ビジネ

ス・オブジェクトをユーザー・インターフェースにバインドすることです。現在ではおそらく XML スキーマが、ビジネス・オブジェクトの構造を表す方法として最もよく使用され、支持されています。

ツールの点から言うと、WebSphere Integration Developer は、開発者にさまざまなドメインからの各種のエンティティーを表す共通 BO モデルを提供します。開発時には、WebSphere Integration Developer はビジネス・オブジェクトを XML スキーマとして表します。しかし実行時には、これらのビジネス・オブジェクトはメモリー内で SDO の Java インスタンスによって表されます。SDO は、IBM と BEA Systems が共同で開発し、同意した標準仕様です。IBM はこの SDO 仕様を拡張し、ビジネス・オブジェクト内でのデータ操作を容易にする追加サービスを組み込みました。

BO フレームワークの詳細に入る前に、操作対象の基本的なデータ・タイプについて説明しておきます。

- **インスタンス・データ**は、スカラー・プロパティーを持つ単純な基本オブジェクトから、大規模で複雑なオブジェクト階層にまで至る、実際のデータおよびデータ構造です。インスタンス・データには、基本属性タイプの記述、複合タイプの情報、カーディナリティー、デフォルト値などのデータ定義も含まれます。
- **インスタンス・メタデータ**は、インスタンス固有のデータです。基本データには、変更のトラッキング (変更サマリーとしても知られています)、オブジェクトまたはデータを作成した方法に関連付けられたコンテキスト情報やメッセージ・ヘッダーおよびフッターなどの増分情報が追加されます。
- **タイプ・メタデータ**は通常、アプリケーション固有の情報です。この情報には、宛先エンタープライズ情報システム (EIS) データ列との属性レベルのマッピングなどがあります (例えば、BO フィールド名と SAP 表の列名とのマッピング)。
- **サービス**は、基本的に、データの取得、データの設定、サマリーの変更、またはデータ定義タイプのアクセスを行うヘルパー・サービスです。

以下の表に、WebSphere プラットフォームでは基本タイプのデータがどのように実装されているかを記載します。

*表 2. データ抽象化および対応する実装*

| データ抽象化 | 実装 |
|---|---|
| インスタンス・データ | ビジネス・オブジェクト (SDO) |
| インスタンス・メタデータ | ビジネス・グラフ |
| タイプ・メタデータ | エンタープライズ・メタデータ、ビジネス・オブジェクト・タイプ・メタデータ |
| サービス | ビジネス・オブジェクト・サービス |

## IBM ビジネス・オブジェクト・フレームワークの操作

前述したように、WebSphere Process Server BO フレームワークは SDO 標準の拡張です。そのため、WebSphere Process Server コンポーネント間で交換されるビジネス・オブジェクトは、`commonj.sdo.DataObject` クラスのインスタンスです。ただし、WebSphere Process Server BO フレームワークは、基本的な DataObject 機能を単純化し、充実させる複数のサービスと関数を追加しています。

ビジネス・オブジェクトの作成と操作を容易にするため、WebSphere BO フレームワークは一連の Java サービスを提供することによって SDO 仕様を拡張します。これらのサービスは、com.ibm.websphere.bo という名前のパッケージに含まれています。

- **BOFactory**: ビジネス・オブジェクトのインスタンスをさまざまな方法で作成できるようにする主要なサービスです。
- **BOXMLSerializer**: ストリームからのビジネス・オブジェクトを拡張する方法や、ビジネス・オブジェクトのコンテンツを XML フォーマットでストリームに書き込む方法を提供します。
- **BOCopy**: ビジネス・オブジェクトのコピーを作成するメソッドを提供します (「ディープ」および「シャロー」のセマンティクス)。
- **BODataObject**: ビジネス・オブジェクトのデータ・オブジェクトの側面 (変更サマリー、ビジネス・グラフ、イベント・サマリーなど) にアクセスできるようにします。
- **BOXMLDocument**: サービスに対するフロントエンドで、ここではビジネス・オブジェクトを XML 文書として操作できます。
- **BOChangeSummary** および **BOEventSummary**: ビジネス・オブジェクトの変更サマリーおよびイベント・サマリーの部分に簡単にアクセスして操作できるようにします。
- **BOEquality**: 2 つのビジネス・オブジェクトに同じ情報が含まれているかどうかを判別できるようにするサービスです。ディープおよびシャロー両方の等価をサポートします。
- **BOType** および **BOTypeMetaData**: これらのサービスは commonj.sdo.Type のインスタンスを実体化して、関連するメタデータを操作できるようにします。これによって、Type のインスタンスを使用して「タイプ別」にビジネス・オブジェクトを作成することができます。
- **BOInstanceValidator** : ビジネス・オブジェクト内のデータを検証して、XSD に準拠しているかどうかを調べます。

## サービス・コンポーネント・アーキテクチャー

SCA は、さまざまな方法で実装できる抽象化です。特定のテクノロジー、プログラミング言語、呼び出しプロトコル、あるいはトランスポート・メカニズムを必要とはしません。SCA コンポーネントを記述するために使用する Service Component Definition Language (SCDL) は、XML ベースの言語です。

SCA コンポーネントには、以下の特性があります。

- コンポーネントが実行可能なロジックを含む実装成果物をラップします。
- 1 つ以上のインターフェースを公開します。
- 1 つ以上の参照を他のコンポーネントに公開できます。コンポーネントが参照を公開するかどうかは、実装のロジックによって決まります。実装が他のサービスを呼び出さなければならない場合、SCA コンポーネントは参照を公開する必要があります。

この情報は、WebSphere Process Server が提供する SCA 実装と、SCA コンポーネントを作成および組み合わせるために使用できる WebSphere Integration Developer

ツールに重点を置きます。WebSphere Process Server および WebSphere Integration Developer は、以下の実装成果物をサポートします。

- プレーン Java オブジェクト
- ビジネス・プロセス
- ビジネス・ステート・マシン
- ヒューマン・タスク
- ビジネス・ルール
- メディエーション・フロー

SCA はビジネス・ロジックをインフラストラクチャーから分離し、アプリケーション・プログラマーがビジネス問題の解決に集中できるようにします。IBM の WebSphere Process Server も、同じ前提に基づきます。図 2 に、WebSphere Process Server のアーキテクチャー・モデルを示します。

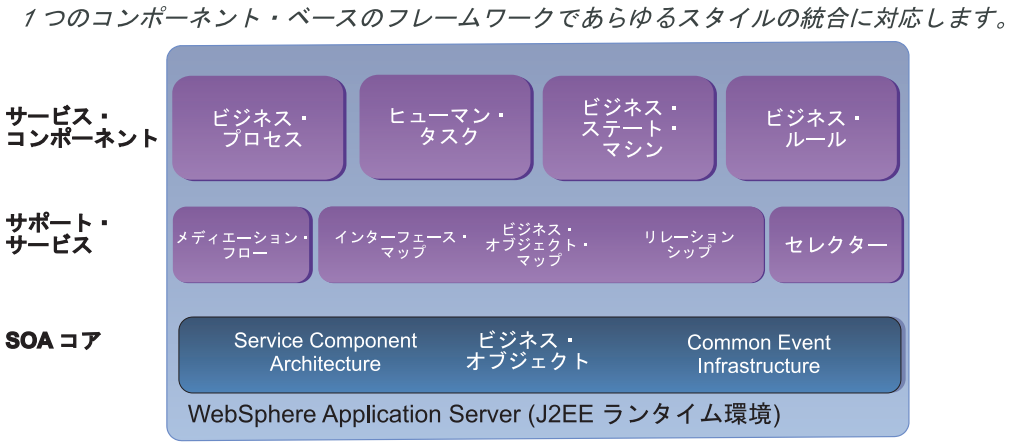



*図 2. WebSphere Process Server コンポーネント・ベースのフレームワーク*

WebSphere 環境では、SCA フレームワークは WebSphere Application Server の Java 2 Platform, Enterprise Edition (J2EE) ランタイム環境をベースとします。全体的な WebSphere Process Server フレームワークは、SOA コア、サポート・サービス、およびサービス・コンポーネントで構成されます。WebSphere Enterprise Service Bus では、これと同じフレームワークで、特にビジネス・インテグレーションの接続およびアプリケーション統合のニーズをターゲットとした全体的機能のサブセットを使用できます。

13 ページの図 3 に示している SCA コンポーネントのインターフェースは、以下のいずれかとして表すことができます。

- Java インターフェース
- WSDL ポート・タイプ (WSDL 2.0 では、ポート・タイプはインターフェースと呼ばれます)

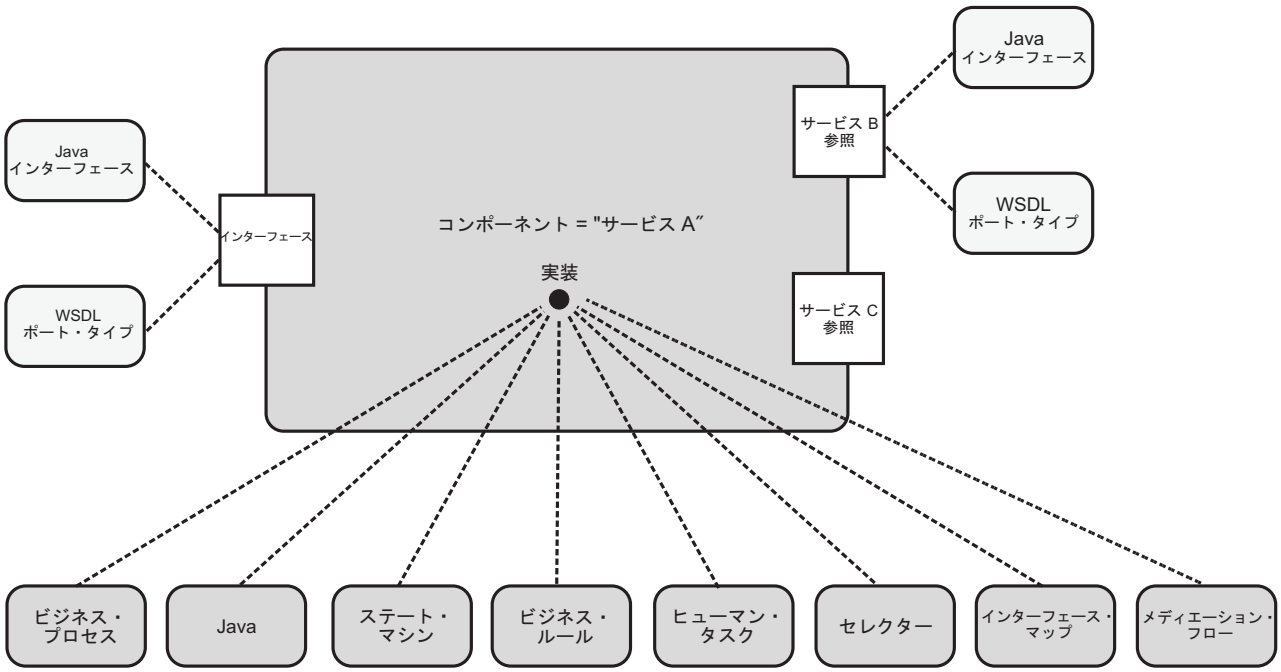



*図 3. WebSphere Process Server での SCA*

SCA モジュールは、参照および実装を直接リンクしてワイヤリングされたコンポーネントのグループです。WebSphere Integration Developer では、各 SCA モジュールにアセンブリー・ダイアグラムが関連付けられます。アセンブリー・ダイアグラムは統合ビジネス・アプリケーションを表し、SCA コンポーネントとこれらのコンポーネントを接続するワイヤーで構成されます。統合開発者の主な職務の 1 つは、ソリューションを形成するコンポーネントを接続してアセンブリー・ダイアグラムを作成することです。WebSphere Integration Developer には、このタスクを支援するグラフィカル・アセンブリー・エディターが用意されています。アセンブリー・ダイアグラムを作成するときに、統合開発者は以下の 2 つの手法のいずれかを開始できます。

- **トップダウン**では、実装を作成する前に、コンポーネントとそのインターフェースおよび相互作用を定義します。統合開発者は、プロセスの構造を定義し、必要なコンポーネントとその実装タイプを識別してから、実装スケルトンを生成できます。
- **ボトムアップ**では、既存のコンポーネントを組み合わせます。この場合、統合開発者に必要な作業は、既存の実装をアセンブリー・ダイアグラムにドラッグ・アンド・ドロップすることだけです。

顧客が既存のサービスを持っていて、それらのサービスを再使用して組み合わせたいという場合には、一般的にボトムアップ手法が使用されます。新しいビジネス・オブジェクトを一から作成する必要がある場合には、トップダウン手法を採用することになります。

## SCA プログラミング・モデル: 基本

SCA プログラミング・モデルの基本となるのは、ソフトウェア・コンポーネントの概念です。前述したように、コンポーネントとは、あるロジックを実装し、そのロジックをインターフェースを介して他のコンポーネントに使用可能にする単位の

ことです。コンポーネントには、他のコンポーネントによって使用可能にされたサービスが必要な場合もあります。その場合、コンポーネントは該当するサービスへの*参照* を公開します。

SCA では、すべてのコンポーネントが 1 つ以上のインターフェースを公開する必要があります。図 4 に示すアセンブリー・ダイアグラムには、C1、C2、C3 の 3 つのコンポーネントがあります。それぞれのコンポーネントのインターフェースは、丸で囲んだ文字 I で表されています。コンポーネントは他のコンポーネントを参照することもできます。参照は、四角で囲んだ文字 R で表されています。参照とインターフェースは、アセンブリー・ダイアグラムでリンクされます。基本的に、統合開発者は、必要なロジックを実装するコンポーネントのインターフェースに参照を接続することによって参照を「解決」します。

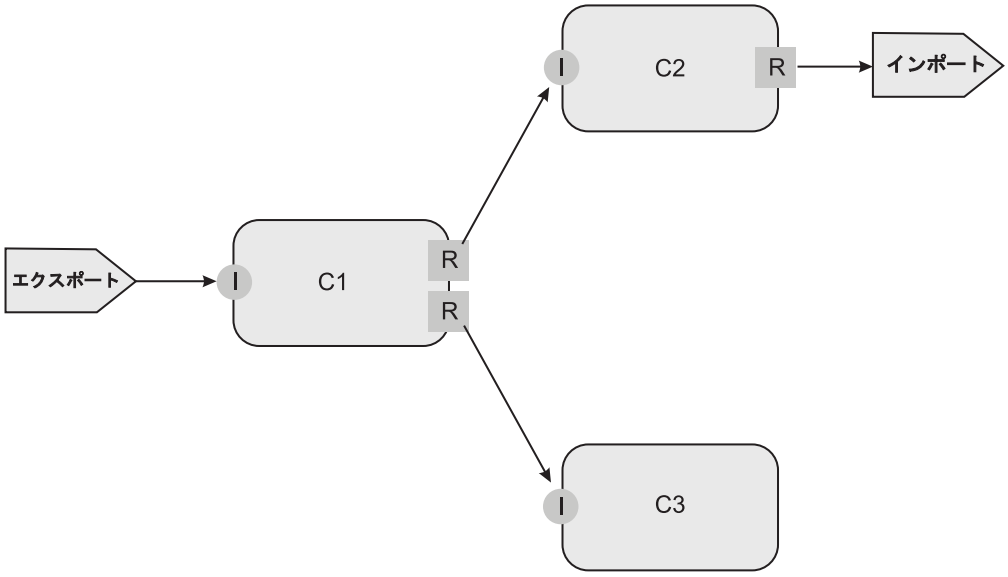



*図 4. アセンブリー・ダイアグラム*

## SCA コンポーネントの呼び出し

呼び出すサービスへのアクセスを提供するために、SCA プログラミング・モデルには *ServiceManager* クラスが組み込まれます。このクラスは、開発者が名前を基準に使用可能なサービスを検索することを可能にします。以下は、サービス検索を説明する典型的な Java コード・フラグメントです。ServiceManager を使用して、BOFactory サービスへの参照を取得します。これはシステム提供のサービスです。

```
//サービス・マネージャー singleton を取得
ServiceManager smgr = new ServiceManager();
//BOFactory サービスにアクセス
BOFactory bof =(BOFactory)
        smgr.locateService("com/ibm/websphere/bo/BOFactory");
```

**注:** ServiceManager のパッケージは、`com.ibm.websphere.sca` です。

開発者は同様のメカニズムを使用して、*locateService* メソッド内で参照されるサービスの名前を指定することにより、独自のサービスへの参照を取得します。

ServiceManager クラスを使用してサービスへの参照を取得した後は、呼び出しプロトコルや実装のタイプに依存しない方法で、そのサービスで使用可能な任意の操作を呼び出せます。

SCA コンポーネントを呼び出すには、以下の 3 種類の呼び出しスタイルを使用できます。

- **同期呼び出し**: この呼び出しスタイルを使用すると、呼び出し側は応答が返されるまで同期的に待機します。これは従来からの呼び出しメカニズムです。
- **非同期呼び出し**: このメカニズムを使用すると、呼び出し側は、応答が作成されるまで待機することなく、直ちにサービスを呼び出せます。応答を受け取る代わりに、呼び出し側は「チケット」を取得します。このチケットを使用して、後で応答を取得することができます。呼び出し側が応答を取得するために呼び出す特殊な操作は、呼び出し先がこの目的専用に提供する必要があります。
- **コールバックを使用した非同期呼び出し**: この呼び出しスタイルは上記のスタイルと同様ですが、応答のリターンは呼び出し先に委任されます。呼び出し側は、応答の準備ができたときに呼び出し先が呼び出すことのできる特殊な操作 (コールバック操作) を公開する必要があります。

## インポート

場合によっては、レガシー・アプリケーションなどの外部システムやその他の外部実装で使用可能なコンポーネントまたは関数によって、ビジネス・ロジックが提供されることがあります。このような場合、統合開発者は、実装が含まれるコンポーネントに参照を接続するという方法で参照を解決することができません。参照は、該当する外部実装を「指す」コンポーネントに接続する必要があります。これらのコンポーネントはインポート と呼ばれます。インポートを定義するときには、外部サービスにアクセスする方法を、ロケーションおよび呼び出しプロトコルという点で指定する必要があります。

## エクスポート

同様に、コンポーネントに外部アプリケーションからのアクセスが必要なこともよくあります。その場合には、コンポーネントをアクセス可能にしなければなりません。それには、ロジックを「外部世界」に公開する特殊なコンポーネントを使用します。これらのコンポーネントはエクスポート と呼ばれます。これらのコンポーネントは、同期的または非同期的に呼び出すことができます。

## スタンドアロン参照

WebSphere Process Server では、SCA サービス・モジュールは J2EE EAR ファイルとしてパッケージされます。このファイルには、他の J2EE サブモジュールもいくつか含まれます。WAR ファイルなどの J2EE エレメントは、SCA モジュールと併せてパッケージできます。JSP などの非 SCA 成果物も SCA サービス・モジュールと併せてパッケージできます。このようにすると、これらのエレメントが、スタンドアロン参照と呼ばれる特殊なコンポーネント・タイプを使用した SCA クライアント・プログラミング・モデルによって SCA を呼び出せるようになります。

SCA プログラミング・モデルは厳格な宣言型です。統合開発者は、呼び出しのトランザクション振る舞い、セキュリティー資格認定の伝搬、宣言による同期呼び出しまたは非同期呼び出しの指定などの側面をアセンブリー・ダイアグラムに直接構成

できます。これらの修飾子で指定された振る舞いを実装する責任は、開発者ではなく、SCA ランタイムにあります。SCA の宣言に関する柔軟性は、このプログラミング・モデルの最も強力な機能の 1 つです。開発者は、非同期呼び出しメカニズムに対応できるようにするなどの技術的側面に重点を置く代わりに、ビジネス・ロジックの実装に専念できます。これらの側面すべては、SCA ランタイムによって自動的に対処されます。

### 修飾子

修飾子は、サービス・クライアントとターゲット・サービスとの間の相互作用を管理します。修飾子はサービス・コンポーネント参照、インターフェース、および実装に指定することが可能で、通常は実装外部にあります。

修飾子には以下の各種カテゴリーがあります。

- トランザクション。SCA 呼び出しでトランザクション・コンテキストを処理する方法を指定します。
- アクティビティー・セッション。アクティビティー・セッションの伝搬方法を指定します。
- セキュリティー。アクセス権を指定します。
- 非同期信頼性。非同期メッセージ配信のルールを指定します。

SCA では、これらのサービスの品質 (QoS) 修飾子を宣言によってコンポーネントに適用することができます (プログラミングの必要も、サービス実装コードを変更する必要もありません)。これは、WebSphere Integration Developer 内で行われます。通常、QoS 修飾子を適用するのは、ソリューション・デプロイメントを検討する準備が整った時点です。詳しくは、『サービスの品質修飾子のリファレンス』を参照してください。

## ビジネス・プロセス

ビジネス・プロセス (具体的には BPEL ベースのビジネス・プロセス) は、SCA に含まれるサービス・コンポーネントの土台を形作ります。

単純な注文の承認であろうが、複雑な製造プロセスであろうが、企業には常にビジネス・プロセスがあります。*ビジネス・プロセス* とは、ビジネスに関連する一連のアクティビティーのことで、これらのアクティビティーはビジネス目標を達成するために特定のシーケンスで呼び出されます。ビジネス・インテグレーションの世界では、ビジネス・プロセスはある種のマークアップ言語を使用して定義されます。

ビジネス・プロセスは、他のサポート・サービスやサービス・コンポーネント、例えばビジネス・ステート・マシン、ヒューマン・タスク、ビジネス・ルール、データ・マップなどを呼び出すことができます。さらに、デプロイされたプロセスは、短時間で実行することも、長期間実行することもできます。プロセスが数年にわたって実行される場合もあります。

J2EE 世界のほとんどのコンポーネントと同じく、ビジネス・プロセスはコンテナー内で稼働します。 IBM WebSphere プラットフォームでは、この特定のコンテナーを Business Process Choreographer と呼びます。Business Process Choreographer は、WebSphere Process Server でビジネス・プロセスとヒューマン・タスクを実行する役割を果たします。

## ヒューマン・タスク

ヒューマン・タスクとは、人とサービスが対話できるようにするコンポーネントのことです。

一部のヒューマン・タスクは、人に対する指示を表します。これらのタスクは、個人または自動サービスのいずれかによって開始できます。ヒューマン・タスクは、ビジネス・プロセス内で人との対話を必要とするアクティビティー (例えば手動の例外処理や承認など) を実装するために使用できます。その他のヒューマン・タスクは、サービスを呼び出したり、または担当者同士のコラボレーションを調整したりするために使用できます。ただし、タスクがどのように開始されたかにはかかわらず、タスクに関連付けられた作業を行うのは、タスクを割り当てられたグループに属する担当者です。

ヒューマン・タスクの担当者は、静的に割り当てるか、または実行時に担当者ディレクトリーを使用して解決される基準 (役割やグループなど) を指定して割り当てます。あるいは、タスクを処理する適切な担当者を見つけるために、ヒューマン・タスクの入力データやビジネス・プロセスのデータが使用されることもあります。

# ビジネス・インテグレーション・アプリケーションの構築

*ビジネス・インテグレーション* とは、企業内または一連の企業間でアプリケーション、データ、およびプロセスを統合することを意味します。また、統合とはプロセスを開発することも意味します。これは、アプリケーションを組み立てて統合するシーケンスには特定のロジックがあるからです。WebSphere Integration Developer は、ビジネス・インテグレーション・アプリケーションを作成するために使用します。

このセクションでは、ビジネス・インテグレーション・モジュールの開発プロセスについての情報を概説します。

標準的なモジュールおよびメディエーション・モジュールの開発フローは以下のとおりです。

1. WebSphere Integration Developer を始動し、ワークスペースを開きます。
2. 開発用の Business Integration パースペクティブに切り替えます。
3. 成果物 (複数のモジュール間で共用するビジネス・オブジェクトおよびインターフェースなど) を保管するライブラリーを作成します。
4. 新規のモジュールまたはメディエーション・モジュールを作成します。
5. アプリケーション・データ (顧客や注文データなど) を含めるためのビジネス・オブジェクトを作成します。
6. コンポーネントごとにインターフェースを作成し、インターフェース操作を定義します。インターフェースによって、コンポーネント間で受け渡しできるデータが決まります。
7. サービス・コンポーネントを作成および実装します。
8. アセンブリー・ダイアグラムにサービス・コンポーネント、インポート、およびエクスポートを追加して、モジュール・アセンブリーを作成します。コンポーネント間をワイヤリングします。インポートとエクスポートをプロトコルにバインドします。

9. 統合テスト環境でモジュールをテストします。
10. モジュールを WebSphere Process Server にデプロイします。
11. テストしたモジュールをリポジトリーに配置して、チームの他のメンバーと共用します。

# 第 2 章 サービス・モジュールの開発

サービス・コンポーネントは、サービス・モジュール内に含まれていなければなりません。サービス・コンポーネントを含むためのサービス・モジュールを開発することが、ほかのモジュールにサービスを提供するための鍵となります。

### 始める前に

以下のタスクでは、要件を分析した結果、ほかのモジュールで使用できるように、サービス・コンポーネントをインプリメントすると有益であると判断されていることが前提となっています。

### このタスクについて

要件を分析した結果、サービス・コンポーネントの提供と利用が効率的な情報処理手段であると判断できる場合があります。ご使用の環境にとって再使用可能なサービス・コンポーネントが有効であると判断したうえで、サービス・コンポーネントを含むためのサービス・モジュールを作成してください。

**手順**

1. ほかのサービス・モジュールで使用できるコンポーネントを特定します。

   サービス・コンポーネントを特定したら、『サービス・コンポーネントの開発』に進みます。
2. ほかのサービス・モジュール内のサービス・コンポーネントを使用できる、アプリケーション内のサービス・コンポーネントを特定します。

   サービス・コンポーネントとそれぞれのターゲット・コンポーネントを特定したら、『コンポーネントの呼び出し』または『コンポーネントの動的呼び出し』に進みます。
3. クライアント・コンポーネントをワイヤー経由でターゲット・コンポーネントに接続します。

## モジュールの開発の概要

モジュールは、WebSphere Process Server アプリケーションのデプロイメントの基本単位です。モジュールには、アプリケーションが使用するコンポーネント、ライブラリー、およびステージング・モジュールを含めることができます。

モジュールを開発するには、アプリケーションが必要とするコンポーネント、ステージング・モジュール、およびライブラリー (モジュールによって参照される成果物の集合) が実動サーバー上で使用可能であることを確認する必要があります。

WebSphere Integration Developer は、WebSphere Process Server にデプロイするモジュールを開発するための主要なツールです。ほかの環境でモジュールを開発することもできますが、WebSphere Integration Developer を使用するのが最適な方法です。

© Copyright IBM Corp. 2005, 2008

WebSphere Process Server は、ビジネス・サービス用モジュールとメディエーション・モジュールをサポートします。モジュールとメディエーション・モジュールはどちらも、Service Component Architecture (SCA) モジュールのタイプです。メディエーション・モジュールは、サービス起動をターゲットが理解する形式に変換し、要求をターゲットに渡して結果をオリジネーターに戻すことによって、アプリケーション間の通信を可能にします。ビジネス・サービス用のモジュールは、ビジネス・プロセスのロジックを実装します。ただし、メディエーション・モジュール内にパッケージ可能なものと同じメディエーション・ロジックをモジュールに格納することもできます。

以降のセクションで、WebSphere Process Server のモジュールを実装および更新する方法を説明します。

## コンポーネント

SCA モジュールに含まれるコンポーネントは、再使用可能なビジネス・ロジックをカプセル化する基本ビルディング・ブロックです。サービスを提供および取り込むコンポーネントには、インターフェース、参照、および実装が関連付けられます。インターフェースは、サービス・コンポーネントと呼び出し側コンポーネントの間の取り決めを定義します。

モジュールは、WebSphere Process Server を使用して、他のモジュールが使用できるようにサービス・コンポーネントをエクスポートしたり、サービス・コンポーネントをインポートして使用したりすることができます。サービス・コンポーネントを呼び出すために、呼び出し側のモジュールはサービス・コンポーネントとのインターフェースを参照します。呼び出し側モジュールからそれぞれのインターフェースへの参照を構成することによって、インターフェースに対する参照が解決されます。

モジュールを開発するには、以下の作業を行う必要があります。

1. モジュール内でコンポーネントのインターフェースを定義または識別します。
2. コンポーネントが使用するビジネス・オブジェクトを定義または操作します。
3. コンポーネントをそれぞれのインターフェースを使用して定義または変更します。

   **注:** コンポーネントは、インターフェースを使用して定義されます。
4. サービス・コンポーネントをエクスポートまたはインポートします。
5. ランタイムにデプロイするエンタープライズ・アーカイブ (EAR) ファイルを作成します。このファイルを作成するには、WebSphere Integration Developer の EAR エクスポート機能または serviceDeploy コマンドのいずれかを使用します。

## 開発タイプ

WebSphere Process Server では、サービス指向のプログラミング・パラダイムを促進するコンポーネント・プログラミング・モデルを提供します。このモデルを使用するために、提供者はサービス・コンポーネントのインターフェースをエクスポートします。これにより、利用者はそのインターフェースをインポートして、そのサービス・コンポーネントがローカルであるかのように使用できるようになります。開発者は厳密に型指定されたインターフェースまたは動的型付きインターフェース

のいずれかを使用して、サービス・コンポーネントをインプリメントしたり、呼び出したりします。インターフェースとそのメソッドについては、このインフォメーション・センターの『参照』のセクションに説明があります。

サービス・モジュールをサーバーにインストールした後、管理コンソールを使用して、アプリケーションからの参照のターゲット・コンポーネントを変更することができます。新しいターゲットは、アプリケーションからの参照が要求しているものと同じビジネス・オブジェクト・タイプを受け入れ、同じ操作を実行する必要があります。

### サービス・コンポーネントの開発に関する考慮事項

サービス・コンポーネントを開発する場合は、以下の点を検討してください。

- このサービス・コンポーネントがエクスポートされ、ほかのモジュールによって使用されるかどうか。

  使用される場合、そのコンポーネントに定義したインターフェースを別のモジュールが使用できることを確認してください。
- サービス・コンポーネントを実行するのに比較的長い時間がかかるかどうか。

  長時間かかる場合は、サービス・コンポーネントに非同期のインターフェースをインプリメントすることを検討してください。
- サービス・コンポーネントを分散化することが有益かどうか。

  有益である場合は、サーバーのクラスター上にデプロイされているサービス・モジュール内にサービス・コンポーネントのコピーを配置して、並列処理の利点を活かすことを検討してください。
- アプリケーションが、一相および二相コミット・リソースの混用を必要とするか。

  必要とする場合、アプリケーションの Last Participant サポートを使用可能にしてください。

  **注:** WebSphere Integration Developer を使用してアプリケーションを作成したか、または serviceDeploy コマンドを使用してインストール可能な EAR ファイルを作成した場合、これらのツールは自動的にアプリケーションのサポートを使用可能にします。WebSphere Application Server Network Deployment インフォメーション・センターで、トピック「同一トランザクション内での 1 フェーズ・コミットおよび 2 フェーズ・コミットのリソースの使用」を参照してください。

## サービス・コンポーネントの開発

ご使用のサーバー内の複数のアプリケーションに再使用可能なロジックを提供するための、サービス・コンポーネントを作成します。

### 始める前に

この作業では、複数のモジュールで使用できる処理がすでに作成され、特定されていることが前提になっています。

## このタスクについて

複数のモジュールで 1 つのサービス・コンポーネントを使用することができます。サービス・コンポーネントをエクスポートすると、インターフェースを介してそのコンポーネントを参照するほかのモジュールが、そのサービス・コンポーネントを利用できるようになります。この作業では、ほかのモジュールがコンポーネントを使用できるように、そのサービス・コンポーネントを作成する方法を説明します。

**注:** 1 つのサービス・コンポーネントに、複数のインターフェースを設定することができます。

**手順**

1. 呼び出し元とサービス・コンポーネントの間のデータの移動のためのデータ・オブジェクトを定義します。

   データ・オブジェクトおよびそのタイプは、呼び出し元とサービス・コンポーネント間のインターフェースの一部となります。
2. 呼び出し元がサービス・コンポーネントを参照するときに使用するインターフェースを定義します。

   このインターフェース定義で、サービス・コンポーネントを指定し、サービス・コンポーネント内のすべての使用可能なメソッドをリストします。
3. サービスの呼び出しを実装するクラスを生成します。
4. 生成されたクラスの実装を開発します。
5. コンポーネントのインターフェース、および実装を拡張子が .java のファイルに保管します。
6. サービス・モジュールと必要なリソースを JAR ファイルにパッケージ化します。

   ステップ 6 から 8 までの詳しい説明については、このインフォメーション・センターの『実動サーバーへのモジュールのデプロイ』のセクションを参照してください。
7. serviceDeploy コマンドを実行して、アプリケーションを格納するインストール可能な EAR ファイルを作成します。
8. サーバー・ノード上にアプリケーションをインストールします。
9. オプション: ほかのサービス・モジュール内のサービス・コンポーネントを呼び出す場合は、呼び出し元とそれに対応するサービス・コンポーネント間のワイヤーを構成します。

   このインフォメーション・センターの『管理』セクションに、ワイヤーの構成についての説明があります。

## コンポーネントの開発例

この例では、1 つのメソッド CustomerInfo をインプリメントする同期型サービス・コンポーネントを示しています。最初のセクションでは、getCustomerInfo というメソッドをインプリメントするサービス・コンポーネントに対するインターフェースを定義しています。

```
public interface CustomerInfo {
 public Customer getCustomerInfo(String customerID);
}
```

以下のコード・ブロックで、サービス・コンポーネントをインプリメントします。

```
public class CustomerInfoImpl implements CustomerInfo {
 public Customer getCustomerInfo(String customerID) {
  Customer cust = new Customer();

  cust.setCustNo(customerID);
  cust.setFirstName("Victor");
  cust.setLastName("Hugo");
  cust.setSymbol("IBM");
  cust.setNumShares(100);
  cust.setPostalCode(10589);
  cust.setErrorMsg("");

  return cust;
 }
}
```

x

以下のセクションは、StockQuote に関連したクラスの実装です。

```
public class StockQuoteImpl implements StockQuote {

 public float getQuote(String symbol) {


     return 100.0f;
 }
}
```

### 次のタスク

サービスを起動します。

## コンポーネントの呼び出し

モジュールを含むコンポーネントは、WebSphere Process Server クラスターの任意のノード上でコンポーネントを使用することができます。

### 始める前に

コンポーネントを呼び出す前に、WebSphere Process Server に、コンポーネントを含むモジュールがインストールされていることを確認してください。

### このタスクについて

コンポーネントは、コンポーネントの名前を使用し、コンポーネントに適したデータ型を渡すことによって、WebSphere Process Server クラスター内で使用可能なすべてのサービス・コンポーネントを使用することができます。この環境内でコンポーネントを呼び出すには、必要なコンポーネントを見つけてから、そのコンポーネントへの参照を作成する操作が必要です。

**注:** モジュール内のコンポーネントは、同一のモジュール内のコンポーネントを呼び出すことができ、これはモジュール内呼び出しと呼ばれます。提供側コンポーネ

ント内のインターフェースをエクスポートし、呼び出し側コンポーネント内でインターフェースをインポートすることによって、外部呼び出し (モジュール内呼び出し) をインプリメントしてください。

**重要:** 呼び出し側モジュールが稼働するサーバーと異なるサーバー上に存在するコンポーネントを呼び出す場合は、サーバーへの追加構成を実行する必要があります。必要な構成は、コンポーネントが非同期に呼び出されるか、同期して呼び出されるかによって異なります。この場合のアプリケーション・サーバーの構成方法は、関連タスクで説明されています。

**手順**

1. 呼び出し側モジュールに必要なコンポーネントを判別します。

   コンポーネント内のインターフェースの名前と、そのインターフェースに必要なデータ型を書き留めます。
2. データ・オブジェクトを定義します。

   入力または戻りは Java クラスでかまいませんが、サービス・データ・オブジェクトが最適です。
3. コンポーネントを探します。
   a. ServiceManager クラスを使用して、呼び出し側モジュールが使用できる参照を取得します。
   b. locateService() メソッドを使用して、コンポーネントを探します。

      インターフェースは、コンポーネントに応じて、Web サービス記述言語 (WSDL) ポート・タイプまたは Java インターフェースのいずれかを使用することができます。
4. コンポーネントを同期式に呼び出します。

   Java インターフェースを使用してコンポーネントを呼び出すことも、 invoke() メソッドを使用してコンポーネントを動的に呼び出すこともできます。
5. 戻り値を処理します。

   コンポーネントが例外を生成することがあるので、クライアントでは例外の処理が可能である必要があります。

## コンポーネントの呼び出し例

次の例では、ServiceManager クラスを作成します。

```
ServiceManager serviceManager = new ServiceManager();
```

以下の例は、ServiceManager クラスを使用して、コンポーネントの参照を含んでいるファイルからコンポーネントのリストを取得します。

```
InputStream myReferences = new FileInputStream("MyReferences.references");
ServiceManager serviceManager = new ServiceManager(myReferences);
```

以下のコードは、StockQuote Java インターフェースをインプリメントするコンポーネントを探します。

```
StockQuote stockQuote = (StockQuote)serviceManager.locateService("stockQuote");
```

以下のコードは、Java または WSDL ポート・タイプ・インターフェースをインプリメントするコンポーネントを探します。呼び出し側モジュールは、Service インターフェースを使用して、コンポーネントと対話します。

**ヒント:** コンポーネントが Java インターフェースをインプリメントする場合は、コンポーネントをインターフェースまたは invoke() メソッドのいずれかを使用して呼び出すことができます。

```
Service stockQuote = (Service)serviceManager.locateService("stockQuote");
```

次の例は、別のコンポーネントを呼び出すコード MyValue を示しています。

```
public class MyValueImpl implements MyValue {

 public float myValue throws MyValueException {

  ServiceManager serviceManager = new ServiceManager();

     // variables
         Customer customer = null;
         float quote = 0;
         float value = 0;

     // invoke
         CustomerInfo cInfo =
      (CustomerInfo)serviceManager.locateService("customerInfo");
         customer = cInfo.getCustomerInfo(customerID);

     if (customer.getErrorMsg().equals("")) {

         // invoke
              StockQuote sQuote =
            (StockQuote)serviceManager.locateService("stockQuote");
              Ticket ticket =  sQuote.getQuote(customer.getSymbol());
    // ... do something else ...
       quote =  sQuote.getQuoteResponse(ticket, Service.WAIT);

         // assign
          value = quote * customer.getNumShares();
     } else {
         // throw
         throw new MyValueException(customer.getErrorMsg());
     }
     // reply
         return value;
 }
}
```

### 次のタスク

呼び出し側モジュールの参照とコンポーネントのインターフェースの間のワイヤーを構成します。

## コンポーネントの動的呼び出し

Web サービス記述言語 (WSDL) ポート・タイプ・インターフェースを指定したコンポーネントをモジュールから呼び出す場合、モジュールは invoke() メソッドを使用して、そのコンポーネントを動的に呼び出す必要があります。

## 始める前に

この操作では、呼び出し側コンポーネントがコンポーネントを動的に呼び出すことが前提となっています。

## このタスクについて

WSDL ポート・タイプ・インターフェースの場合は、呼び出し側コンポーネントは invoke() メソッドを使用して、コンポーネントを呼び出す必要があります。呼び出し側モジュールから、この方法で Java インターフェースを指定したコンポーネントも呼び出すことができます。

**手順**

1. 必要なコンポーネントを含んでいるモジュールを判別します。
2. コンポーネントが必要とする配列を判別します。

   入力配列は、次の 3 つのタイプのいずれかです。
   - 大文字の Java プリミティブ型、またはこの型の配列
   - 通常の Java クラス、またはクラスの配列
   - サービス・データ・オブジェクト (SDO)
3. コンポーネントからの応答を収容する配列を定義します。

   応答配列は、入力配列と同じタイプでかまいません。
4. invoke() メソッドを使用して、必要なコンポーネントを呼び出し、配列オブジェクトをそのコンポーネントに渡します。
5. 結果を処理します。

## コンポーネントの動的呼び出しの例

以下の例では、モジュールは invoke() メソッドを使用して、大文字の Java プリミティブ・データ型を使用するコンポーネントを呼び出します。

```
Service service = (Service)serviceManager.locateService("multiParamInf");

  Reference reference = service.getReference();

  OperationType methodMultiType =
    reference.getOperationType("methodWithMultiParameter");

  Type t = methodMultiType.getInputType();

  BOFactory boFactory = (BOFactory)serviceManager.locateService
    ("com/ibm/websphere/bo/BOFactory");

  DataObject paramObject = boFactory.createbyType(t);

  paramObject.set(0,"input1")
  paramObject.set(1,"input2")
  paramObject.set(2,"input3")

  service.invoke("methodMultiParamater",paramObject);
```

次の例では、WSDL ポート・タイプ・インターフェースをターゲットとして持つ呼び出しメソッドを使用します。

```
Service serviceOne = (Service)serviceManager.locateService("multiParamInfWSDL");

 DataObject dob = factory.create("http://MultiCallWSServerOne/bos", "SameBO");
   dob.setString("attribute1", stringArg);

 DataObject wrapBo = factory.createByElement
  ("http://MultiCallWSServerOne/wsdl/ServerOneInf", "methodOne");
   wrapBo.set("input1", dob); //wrapBo encapsulates all the parameters of methodOne
   wrapBo.set("input2", "XXXX");
   wrapBo.set("input3", "yyyy");

 DataObject resBo= (DataObject)serviceOne.invoke("methodOne", wrapBo);
```

## モジュールとターゲットの分離の概要

モジュールを開発する際、複数のモジュールが使用できるサービスを識別します。このようにしてサービスにてこ入れすることにより、開発サイクルとコストを最小化します。多数のモジュールによって使用されるサービスがある場合は、ターゲットがアップグレードされた場合に新規サービスへの切り替えが呼び出しモジュールに対して透過的になるように、呼び出しモジュールをターゲットから分離する必要があります。このトピックでは、単純な呼び出しモデルと分離された呼び出しモデルを対比して、分離がどのように役立つかを示す例を提供します。特定の例について説明しますが、これが、ターゲットからモジュールを分離する唯一の方法というわけではありません。

### 単純な呼び出しモデル

モジュールを開発する際、その他のモジュールにあるサービスを使用することができます。これは、モジュールにサービスをインポートしてからそのサービスを呼び出すことによって実行します。インポートされたサービスは、WebSphere Integration Developer で、または管理コンソール内のサービスをバインディングすることによって、その他のモジュールによってエクスポートされたサービスに「関連付け」られます。『単純な呼び出しモデル』は、このモデルを示しています。

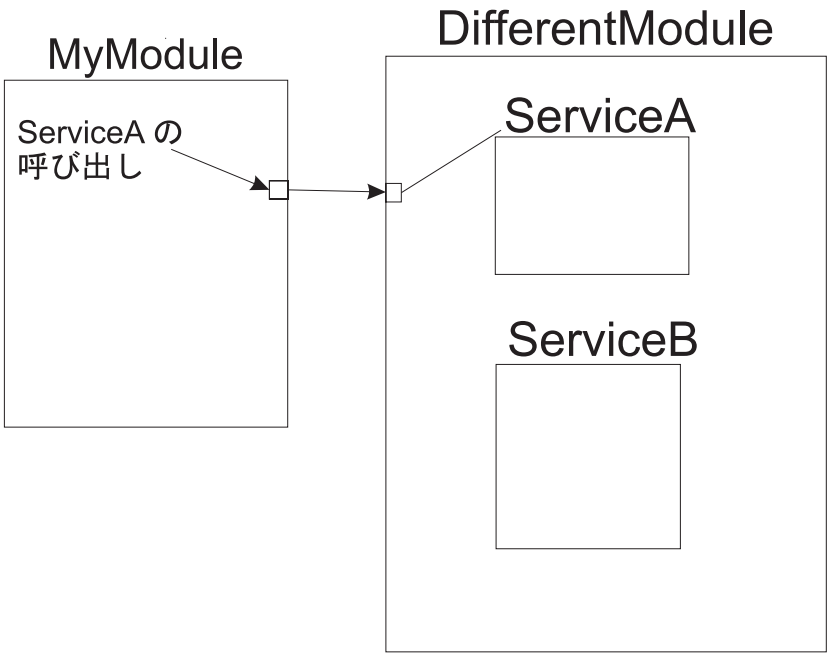



*図 5. 単純な呼び出しモデル*

## 分離された呼び出しモデル

呼び出しモジュールを停止せずに呼び出しのターゲットを変更するには、呼び出しのターゲットから呼び出しモジュールを分離します。この場合、モジュールそのものではなくダウンストリーム・ターゲットを変更しているので、ターゲットの変更中もモジュールが処理を続行できます。『アプリケーションの分離の例』は、分離によって、呼び出しモジュールの状況に影響を与えずにターゲットを変更する方法を示します。

## アプリケーションの分離の例

単純な呼び出しモデルを使用した場合、同一のサービスを呼び出す複数のモジュールは、『単一のサービスを呼び出す複数のアプリケーション』のようになります。MODA、MODB、および MODC はすべて CalculateFinalCost を呼び出します。

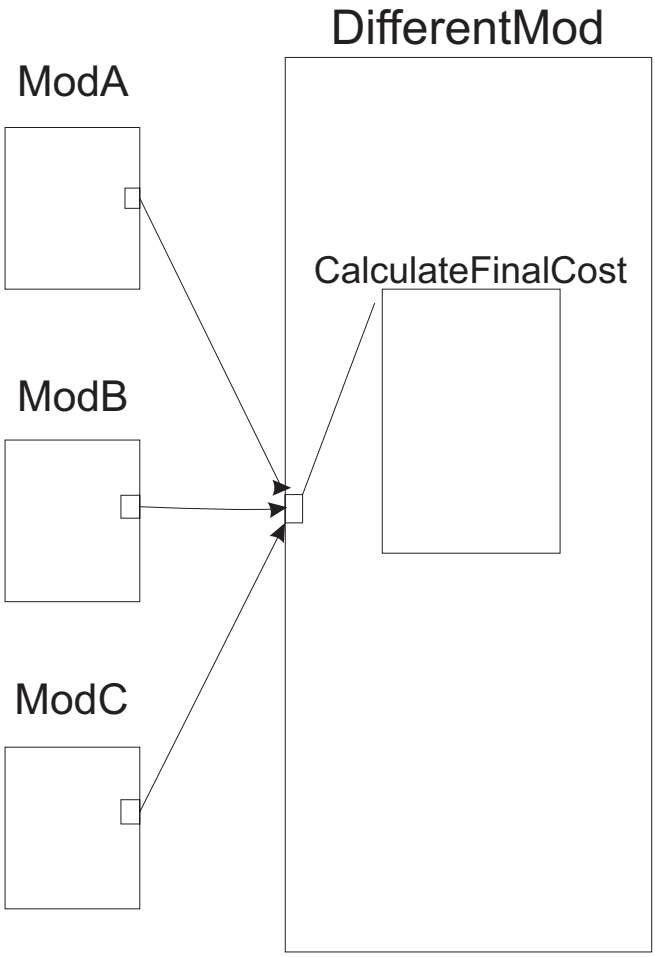



*図 6. 単一のサービスを呼び出す複数のアプリケーション*

CalculateFinalCost によって提供されるサービスは、そのサービスを使用するすべてのモジュールに新規コストが反映されるように、更新する必要があります。開発チームは、新規サービス UpdatedCalculateFinal を構築およびテストして、変更を取り込みます。新規サービスは実動に移す準備ができています。分離を使用しない場合は、UpdateCalculateFinal を呼び出すために、CalculateFinalCost を呼び出すモジュールをすべて更新する必要があります。分離を使用した場合、実行する必要があるのは、バッファー・モジュールをターゲットに接続するバインディングを変更することだけです。

**注:** このようにサービスを変更することにより、オリジナルのサービスを、それを必要とするその他のモジュールに提供し続けることができます。

分離を使用して、アプリケーションとターゲットの間でバッファー・モジュールを作成します (『UpdateCalculateFinal を呼び出す分離された呼び出しモデル』を参照してください)。

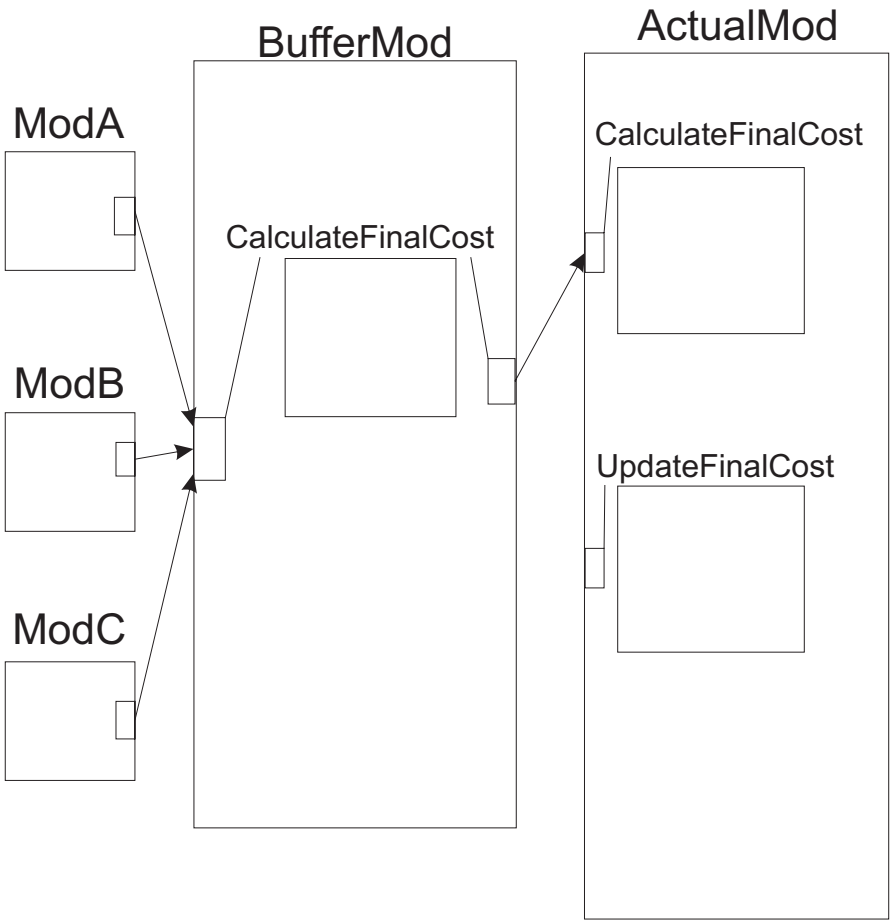



*図 7. UpdateCalculateFinal を呼び出す分離された呼び出しモデル*

このモデルで、呼び出しモジュールは変わりません。実行する必要があるのは、バインディングをバッファー・モジュール・インポートからターゲットへ変更することだけです (『UpdatedCalculateFinal を呼び出す分離された呼び出しモデル』を参照してください)。

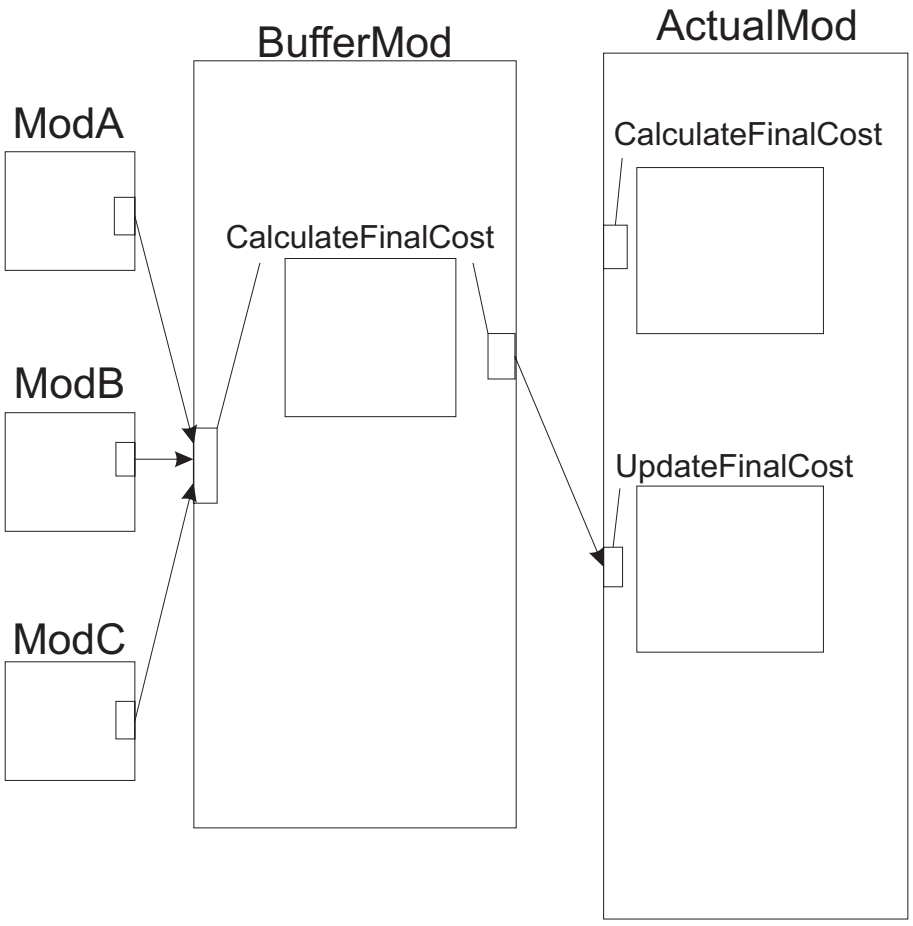



*図 8. UpdatedCalculateFinal を呼び出す分離された呼び出しモデル*

バッファー・モジュールがターゲットを同期で呼び出す場合、元のアプリケーションに戻される結果は、バッファー・モジュール (メディエーション・モジュール、またはビジネス・モジュール用のサービス) を再始動したときに、新規ターゲットから送信されます。バッファー・モジュールがターゲットを非同期で呼び出す場合、元のアプリケーションに戻される結果は、次の呼び出し時に新規ターゲットから送信されます。

## HTTP バインディング

HTTP バインディングは、Service Component Architecture (SCA) と HTTP の接続を提供する目的で設計されています。これにより、既存のまたは新規作成された HTTP アプリケーションをサービス指向アーキテクチャー (SOA) 環境内に組み込むことができます。

さらに、SCA ランタイム環境のネットワークは、既存の HTTP インフラストラクチャーを通して通信できます。

HTTP バインディングは、以下のような HTTP 機能を公開します。

- メッセージがメディエーション・コンポーネントに渡されるときに、HTTP フォーマットとメッセージ・ヘッダー情報が保持されます。これにより、HTTP アプリケーションのプログラマー、ユーザー、および管理者に、より分かりやすい表示が提供されます。
- 既存のデータ・バインディング・フレームワークが HTTP 規則に対応して拡張され、SCA メッセージと HTTP メッセージ・ヘッダーおよび本体との間のマッピングが提供されます。
- さまざまな共通 HTTP 機能をサポートするようにインポートおよびエクスポートを構成できます。
- HTTP インポートまたはエクスポートを含む SCA モジュールをインストールすると、HTTP 接続が可能になるように、適切なランタイム環境が自動的に構成されます。

HTTP インポートおよびエクスポートの詳しい作成手順は、 インフォメーション・センターの「**WebSphere Integration Developer**」>「**統合アプリケーションの開発**」>「**HTTP データ・バインディング**」を参照してください。

**関連タスク**

HTTP バインディングの表示
アプリケーションのデプロイ後に、HTTP バインディングを検査して、それが正しいことを確認する必要がある場合があります。

HTTP エクスポート・バインディングの変更
管理コンソールでは、HTTP エクスポート・バインディングの構成を変更するときに、元のソースを変更してからアプリケーションを再デプロイする必要はありません。

HTTP インポート・バインディングの変更
管理コンソールでは、HTTP インポート・バインディングの構成を変更するときに、元のソースを変更してからアプリケーションを再デプロイする必要はありません。

# 第 3 章 プログラミング・ガイドおよび手法

このセクションでは、プログラミング・ガイドと例を説明します。

以下のリンクに、プログラミングの各種コンポーネント、アプリケーション、およびビジネス・インテグレーション・ソリューションに関する情報ガイドが記載されています。

- ビジネス・オブジェクトのプログラミング: スキーマの強化およびインダストリー・スキーマのサポート
- ビジネス・ルール管理のプログラミング・ガイド
- XMS アプリケーションの開発
- ビジネス・プロセスおよびタスク用クライアント・アプリケーションの開発

**重要:** WebSphere Process Server および WebSphere Enterprise Service Bus でサポートされるアプリケーション・プログラミング・インターフェース (API) とシステム・プログラミング・インターフェース (SPI) について詳しくは、インフォメーション・センターの『参照』セクションを参照してください。

© Copyright IBM Corp. 2005, 2008

# 第 4 章 生成される Service Component Architecture 実装のオーバーライド

場合によっては、システムが作成した Java コードとサービス・データ・オブジェクト (SDO) 間の変換では、要件に合わないことがあります。デフォルトの Service Component Architecture (SCA) クラスの実装を独自の実装に置き換えるには、以下の手順を実行します。

## 始める前に

WebSphere Integration Developer または genMapper コマンドを使用して Java から Web サービス記述言語 (WSDL) 型への変換を生成していることを確認します。

## このタスクについて

生成されたコードを、要件に合うコードで置き換えることにより、Java 型から WSDL 型にマップする生成されたコンポーネントをオーバーライドします。独自の Java クラスを定義している場合は、独自のマップを使用することを考慮してください。以下の手順を使用して変更を行います。

### 手順

1. 生成されたコンポーネントを見つけます。 コンポーネントの名前は *java_class*Mapper.component です。
2. テキスト・エディターを使用してコンポーネントを編集します。
3. 生成されたコードをコメント化し、独自のメソッドを指定します。

   コンポーネントの実装を含むファイル名は変更しないでください。

## 例

以下は、置き換え対象の生成されたコンポーネントの例です。

```
private DataObject javatodata_setAccount_output(Object myAccount) {

  // このコードをカスタム・マッピングのためオーバーライドできます。
  // このコードをコメント化してカスタム・コードを書き込みます。

  // コンバーターに渡される Java 型は、コンバーターが作成を試行します。
  // この Java 型を変更することもできます。


 return SDOJavaObjectMediator.java2Data(myAccount);


}
```

## 次のタスク

コンポーネントおよびその他のファイルを、含んでいるモジュールが存在するディレクトリーにコピーし、WebSphere Integration Developer のコンポーネントをワイヤ

© Copyright IBM Corp. 2005, 2008

リングするか、serviceDeploy コマンドを使用してエンタープライズ・アーカイブ (EAR) ファイルを生成します。

**関連概念**

41 ページの『第 7 章 Java からサービス・データ・オブジェクトへの変換で使用されるランタイム・ルール』
生成されるコードを正しくオーバーライドする、または Java からサービス・データ・オブジェクト (SDO) への変換に関連して起こりうるランタイム例外を判別するには、関係のあるルールを理解することが重要です。変換の大部分は簡単なものですが、生成されたコードを変換するときに、ランタイムが最適な方法を提供するという、複雑なケースもあります。

**関連資料**


Java から XML への変換
システムは、事前定義のルールを使用して、Java 型に基づいて XML を生成します。


genMapper コマンド行ユーティリティー
genMapper コマンドは、Service Component Architecture (SCA) 参照を Java インターフェースにブリッジするコンポーネントを生成する場合に使用します。

# 第 5 章 サービス・データ・オブジェクトから Java への変換のオーバーライド

場合によっては、システムが作成するサービス・データ・オブジェクト (SDO) と Java 型オブジェクト間の変換では、要件に合わないことがあります。デフォルトの実装を独自の実装に置き換えるには、以下の手順を実行します。

## 始める前に

WebSphere Integration Developer または genMapper コマンドを使用して WSDL から Java 型への変換を生成していることを確認します。

## このタスクについて

生成されたコードを、要件に合うコードで置き換えることにより、WSDL 型から Java 型にマップする生成済みコンポーネントをオーバーライドします。独自の Java クラスを定義している場合は、独自のマップを使用することを考慮してください。以下の手順を使用して変更を行います。

**手順**

1. 生成されたコンポーネントを見つけます。 コンポーネントの名前は *java_class*Mapper.component です。
2. テキスト・エディターを使用してコンポーネントを編集します。
3. 生成されたコードをコメント化し、独自のメソッドを指定します。

   コンポーネントの実装を含むファイル名は変更しないでください。

## 例

以下は、置き換え対象の生成されたコンポーネントの例です。

```
 private Object datatojava_get_customerAcct(DataObject myCustomerID,
    String integer)
{

  // このコードをカスタム・マッピングのためオーバーライドできます。
  // このコードをコメント化してカスタム・コードを書き込みます。

  // コンバーターに渡される Java 型は、コンバーターが作成を試行します。
  // この Java 型を変更することもできます。


  return SDOJavaObjectMediator.data2Java(customerID, integer) ;

}
```

## 次のタスク

コンポーネントおよびその他のファイルを、含んでいるモジュールが存在するディレクトリーにコピーし、WebSphere Integration Developer のコンポーネントをワイヤ

© Copyright IBM Corp. 2005, 2008

リングするか、serviceDeploy コマンドを使用してエンタープライズ・アーカイブ (EAR) ファイルを生成します。

**関連概念**

41 ページの『第 7 章 Java からサービス・データ・オブジェクトへの変換で使用されるランタイム・ルール』
生成されるコードを正しくオーバーライドする、または Java からサービス・データ・オブジェクト (SDO) への変換に関連して起こりうるランタイム例外を判別するには、関係のあるルールを理解することが重要です。変換の大部分は簡単なものですが、生成されたコードを変換するときに、ランタイムが最適な方法を提供するという、複雑なケースもあります。

**関連資料**


Java から XML への変換
システムは、事前定義のルールを使用して、Java 型に基づいて XML を生成します。


genMapper コマンド行ユーティリティー
genMapper コマンドは、Service Component Architecture (SCA) 参照を Java インターフェースにブリッジするコンポーネントを生成する場合に使用します。

# 第 6 章 非 SCA エクスポート・バインディングからのプロトコル・ヘッダー伝搬

コンテキスト・サービスは、コンテキスト (JMS ヘッダーなどのプロトコル・ヘッダーや、アカウント ID などのユーザー・コンテキストを含む) を Service Component Architecture (SCA) 呼び出しパスに沿って伝搬します。コンテキスト・サービスは、一連の API と構成可能な設定を提供します。

コンテキスト・サービス伝搬が双方向の場合、現行のコンテキストは常に応答コンテキストによって上書きされます。SCA コンポーネントから別の SCA コンポーネントへの呼び出しを実行している場合、応答には異なるコンテキストが含まれます。サービス・コンポーネントには着信コンテキストがありますが、別のサービスを呼び出すと、元の発信コンテキストがもう一方のサービスによって上書きされます。応答コンテキストは新しいコンテキストになります。

コンテキスト・サービス伝搬が片方向の場合、元のコンテキストがそのまま維持されます。

コンテキスト・サービスのライフ・サイクルは、呼び出しに関連付けられます。要求には関連コンテキストがあり、そのコンテキストのライフ・サイクルがその特定の要求の処理にバインドされます。要求が処理を終了すると、そのコンテキストのライフ・サイクルが終了します。

実行時間が短い Business Process Execution Language (BPEL) プロセスの場合、応答コンテキストが要求コンテキストを上書きします。最初の要求から応答コンテキストが取り戻されて、次の要求にプッシュされます。長期間実行される BPEL プロセスの場合は、応答コンテキストが BPEL フレームワークによって廃棄されます。元のコンテキストは保管され、別の発呼を行う際にはそのコンテキストが使用されます。

## 例

例えば、プロトコル・ヘッダーが含まれるコンテキストが、SOAP Web サービスから BPEL に着信する要求で始まる呼び出しパスで伝搬されるとします。BPEL がこの要求を処理すると、続いて BPEL の呼び出しが Web サービスがバインドするアウトバウンドに対して行われ、次に別の Web サービスがバインドするアウトバウンドに対して行われます。SOAP Web サービスからの要求は、コンテキスト・サービスを使用してプロトコル・ヘッダーを渡します。コンテキスト・サービスをインバウンド要求から取得し、それをプロトコル・ヘッダーのアウトバウンドにプッシュします。

この例の BPEL を別の SCA コンポーネントに置き換えても、同じような振る舞いが行われることが分かるはずです。

© Copyright IBM Corp. 2005, 2008

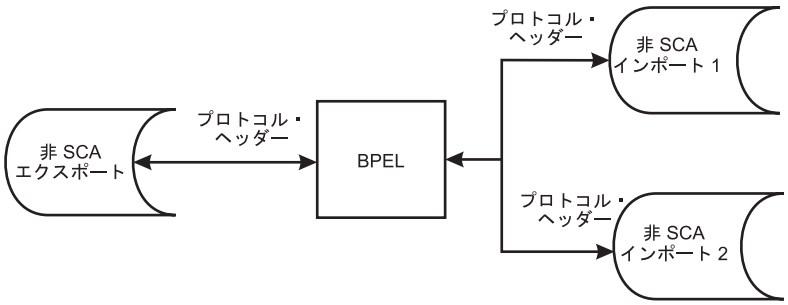



*図 9. プロトコル・ヘッダーが含まれるコンテキストの伝搬*

以下にコード例を示します。

```
//必要なクラスをインポート
import com.ibm.bpm.context.ContextService;
import com.ibm.websphere.sca.ServiceManager;
import com.ibm.bpm.context.cobo.ContextObject;
import com.ibm.bpm.context.cobo.ContextObjectFactory;
import com.ibm.bpm.context.cobo.HeaderInfoType;
import com.ibm.bpm.context.cobo.UserDefinedContextType;

//ContextService の場所を指定
ContextService contextService =
 (ContextService)ServiceManager.INSTANCE.locateService("com/ibm/bpm/context/ContextService");

// ヘッダー情報を取得
HeaderInfo headerInfo = contextService.getHeaderInfo();
// 現行の実行コンテキストでユーザー定義コンテキストを取得
UserDefinedContextType userDefinedContext = contextService.getUserDefinedContext();
if(userDefinedContext == null){ // create a new context if context is null
userDefinedContext = ContextObjectFactory.eINSTANCE.createUserDefinedContextType()
}

// ヘッダー情報および userDefinedContext を変更

// ユーザー定義コンテキストを現行の実行コンテキストに再び設定
contextService.setUserDefinedContext(userDefinedContext);

// ヘッダー情報を現行の実行コンテキストに再び設定
contextService.setHeaderInfo(headerInfo);
```

**注:** メディエーション・フロー・コンポーネントでは、ContextService API を使用しないでください。コンテキストにアクセスするには SMO プログラミング・モデルを使用します。

コンテキスト・サービスには、バインディング振る舞いを指定する構成可能ルールと表があります。詳しくは、『参照』セクションに記載されている生成済み API および SPI の資料を参照してください。コンテキスト・サービスは、WebSphere® Integration Developer の開発中に、インポートおよびエクスポート・プロパティーに設定できます。詳しくは、WebSphere Integration Developer インフォメーション・センターのインポートおよびエクスポート・バインディングに関する情報を参照してください。

# 第 7 章 Java からサービス・データ・オブジェクトへの変換で使用されるランタイム・ルール

生成されるコードを正しくオーバーライドする、または Java からサービス・データ・オブジェクト (SDO) への変換に関連して起こりうるランタイム例外を判別するには、関係のあるルールを理解することが重要です。変換の大部分は簡単なものですが、生成されたコードを変換するときに、ランタイムが最適な方法を提供するという、複雑なケースもあります。

## 基本の型およびクラス

ランタイムは、サービス・データ・オブジェクトと基本の Java 型およびクラス間で単純な変換を実行します。基本の型およびクラスは以下のとおりです。

- Char または java.lang.Character
- Boolean
- Java.lang.Boolean
- Byte または java.lang.Byte
- Short または java.lang.Short
- Int または java.lang.Integer
- Long または java.lang.Long
- Float または java.lang.Float
- Double または java.lang.Double
- Java.lang.String
- Java.math.BigInteger
- Java.math.BigDecimal
- Java.util.Calendar
- Java.util.Date
- Java.xml.namespace.QName
- Java.net.URI
- Byte[]

## ユーザー定義の Java クラスおよび配列

Java クラスまたは配列から SDO に変換する場合、ランタイムが作成するデータ・オブジェクトは、Java 型のパッケージ名を反転して生成される URI を持ち、Java クラス名と同じ型を持ちます。例えば、Java クラス com.ibm.xsd.Customer が SDO に変換されると、URI は http://xsd.ibm.com であり、Customer 型を持ちます。次にランタイムは Java クラス・メンバーのコンテンツを検査し、値を SDO 内のプロパティーに割り当てます。

SDO から Java 型に変換する場合、ランタイムは URI を反転させてパッケージ名を生成し、SDO の型と同じ型の名前を生成します。例えば、型 Customer と、URI http://xsd.ibm.com を持つデータ・オブジェクトは、Java パッケージ com.ibm.xsd.Customer のインスタンスを生成します。次にランタイムは、SDO のプロパティーから値を抽出し、それらのプロパティーを Java クラスのインスタンス内のフィールドに割り当てます。

© Copyright IBM Corp. 2005, 2008

Java クラスがユーザー定義のインターフェースである場合、生成されるコードをオーバーライドして、ランタイムがインスタンス化できる具象クラスを作成する必要があります。ランタイムが具象クラスを作成できない場合、例外が発生します。

## Java.lang.Object

Java 型が java.lang.Object の場合、生成される型は xsd:anyType です。モジュールは、すべての SDO で、このインターフェースを呼び出すことができます。ランタイムは、具象クラスを見つけることができる場合は、ユーザー定義の Java クラスと配列の場合と同様にその具象クラスをインスタンス化しようとします。見つからない場合、ランタイムは SDO を Java インターフェースに渡します。

メソッドが java.lang.Object 型を戻しても、ランタイムは、メソッドが具象型を戻す場合のみ、SDO に変換します。ランタイムは、ユーザー定義の Java クラスおよび配列から SDO への変換でも同様の変換を使用します (次の段落を参照)。

Java クラスまたは配列から SDO に変換する場合、ランタイムが作成するデータ・オブジェクトは、Java 型のパッケージ名を反転して生成される URI を持ち、Java クラス名と同じ型を持ちます。例えば、Java クラス com.ibm.xsd.Customer が SDO に変換されると、URI は http://xsd.ibm.com であり、Customer 型を持ちます。次にランタイムは Java クラス・メンバーのコンテンツを検査し、値を SDO 内のプロパティーに割り当てます。

いずれの場合でも、ランタイムが変換を完了できない場合は例外が発生します。

## Java.util コンテナー・クラス

Vector、HashMap、HashSet などのような具象 Java コンテナー・クラスに変換する場合、ランタイムは該当するコンテナー・クラスをインスタンス化します。ランタイムは、ユーザー定義の Java クラスおよび配列で使用したのと同様の方法で、コンテナー・クラスにデータを取り込みます。ランタイムが具象 Java クラスを見つけられない場合、ランタイムはコンテナー・クラスに SDO を取り込みます。

具象 Java コンテナー・クラスから SDO に変換する場合、ランタイムは、『Java から XML への変換』に示される、生成されたスキーマを使用します。

## Java.util インターフェース

java.util パッケージ内の特定のコンテナー・インターフェースの場合、ランタイムは以下の具象クラスをインスタンス化します。

*表 3. WSDL 型から Java クラスへの変換*

| インターフェース | デフォルトの具象クラス |
|---|---|
| コレクション | HashSet |
| マップ | HashMap |
| リスト | ArrayList |
| セット | HashSet |

**関連タスク**

35 ページの『第 4 章 生成される Service Component Architecture 実装のオーバーライド』
場合によっては、システムが作成した Java コードとサービス・データ・オブジェクト (SDO) 間の変換では、要件に合わないことがあります。デフォルトの Service Component Architecture (SCA) クラスの実装を独自の実装に置き換えるには、以下の手順を実行します。

37 ページの『第 5 章 サービス・データ・オブジェクトから Java への変換のオーバーライド』
場合によっては、システムが作成するサービス・データ・オブジェクト (SDO) と Java 型オブジェクト間の変換では、要件に合わないことがあります。デフォルトの実装を独自の実装に置き換えるには、以下の手順を実行します。

**関連資料**


Java から XML への変換
システムは、事前定義のルールを使用して、Java 型に基づいて XML を生成します。


genMapper コマンド行ユーティリティー
genMapper コマンドは、Service Component Architecture (SCA) 参照を Java インターフェースにブリッジするコンポーネントを生成する場合に使用します。

# 第 8 章 ビジネス・オブジェクト: スキーマの強化およびインダストリー・スキーマのサポート

サービス・データ・オブジェクト (SDO) フレームワークは、WebSphere Process Server によって使用されるビジネス・オブジェクト・データの基盤です。

このガイドでは、いくつかの機能でスキーマ構成を処理する場合の問題領域についての情報を提供します。ビジネス・オブジェクトの定義方法、ビジネス・オブジェクトの開発ガイドライン、およびビジネス・オブジェクト・プログラミング API の使用方法について詳しくは、以下の『関連情報』セクションの記事を参照してください。

**関連情報**

Web Services Description Language (WSDL) 1.1

Introduction to Service Data Objects

Examining business objects in WebSphere Process Server

## 同じ名前がついたエレメントの差別化

ビジネス・オブジェクトのエレメントと属性には、固有の名前を指定する必要があります。

サービス・データ・オブジェクト (SDO) フレームワークでは、エレメントと属性はプロパティーとして作成されます。以下のコード例では、XSD で foo という名前の 1 つのプロパティーを持つタイプを作成します。

```
<xsd:complexType name="ElementFoo">
  <xsd:sequence>
    <xsd:element name="foo" type="xsd:string" default="elem_value"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="AttributeFoo">
  <xsd:attribute name="foo" type="xsd:string" default="attr_value"/>
</xsd:complexType>
```

これらの事例では、XML パス言語 (XPath) を使用してプロパティーにアクセスできます。ただし、有効なスキーマ・タイプは、次の例に示すように、同じ名前の属性とエレメントを持つことができます。

```
<xsd:complexType name="DuplicateNames">
  <xsd:sequence>
    <xsd:element name="foo" type="xsd:string" default="elem_value"/>
  </xsd:sequence>
  <xsd:attribute name="foo" type="xsd:string" default="attr_value"/>
</xsd:complexType>
```

© Copyright IBM Corp. 2005, 2008

XPath では、同じ名前の付いたエレメントと属性を区別できなければなりません。そのためには、一方の名前をアットマーク (@) で始めます。次の断片コードは、同じ名前の付いたエレメントと属性にアクセスする方法を示しています。

```
1 DataObject duplicateNames = ...

2 // 「elem_value」を表示する
3 System.out.println(duplicateNames.get("foo"));

4 // 「attr_value」を表示する
5 System.out.println(duplicateNames.get("@foo"));
```

この命名方式は、SDO XPath である String 値を使用するすべてのメソッドに使用してください。

## モデル・グループ・サポート (all、choice、sequence、および group 参照)

SDO 仕様は、モデル・グループ (all、choice、sequence、および group 参照) を正しい位置に展開して、タイプやプロパティーを記述しないことを必要としています。

基本的に、これは同じ収容構造の中にある構造は、すべて「フラット化される」ことを意味しています。この「フラット化」は、それらの構造の子を同じレベルに置きます。このため、フラット化されたデータから構造が派生した SDO 内で、名前の重複の問題が生じる可能性があります。 XSD がグループをフラット化しない場合でも、さまざまな親に含まれている重複が個別に存在します。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://MultipleGroup">
  <xsd:complexType name="MultipleGroup">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
      <xsd:element name="separator" type="xsd:string"/>
      <xsd:choice>
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

option1 と option2 の複数のオカレンスは異なる choice ブロックに含まれており、それらの間には分離するエレメントさえあるため、XSD および XML は、それらを区別するのに何の問題もありません。しかし、SDO によってこれらのグループがフラット化されると、すべてのオプション・プロパティーは MultipleGroup という同じコンテナーに含まれます。

たとえ重複する名前がなくても、これらのグループをフラット化したことによる意味構造の問題もあります。例えば、以下の XSD を考えてみましょう。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://SimpleChoice">
  <xsd:complexType name="SimpleChoice">
    <xsd:sequence>
      <xsd:choice>
```

```
        <xsd:element name="option1" type="xsd:string"/>
        <xsd:element name="option2" type="xsd:string"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

重複する名前の変更や XSD への特別な注釈の追加をユーザーに求めるのは実際的ではありません。なぜなら、多くの場合、規格や業界のスキーマのように、ユーザーは作業に使用している XSD を制御できないからです。

すべてのプロパティーに一貫性を持たせるために、ビジネス・オブジェクトは、重複した名前を持つプロパティーの個々のオカレンスに XPath を通じてアクセスするメソッドを含んでいます。EMF 命名規則によると、重複するプロパティー名が検出された場合、それらの名前に次の未使用の数字が付加されます。したがって、例えば次の XSD の場合:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://TieredGroup">
  <xsd:complexType name="TieredGroup">
    <xsd:sequence>
      <xsd:choice minOccurs="0">
        <xsd:sequence>
          <xsd:element name="low" minOccurs="1"
     maxOccurs="1" type="xsd:string"/>
          <xsd:choice minOccurs="0">
            <xsd:element name="width" minOccurs="0"
      maxOccurs="1" type="xsd:string"/>
            <xsd:element name="high" minOccurs="0"
      maxOccurs="1" type="xsd:string"/>
          </xsd:choice>
        </xsd:sequence>
        <xsd:element name="high" minOccurs="1"
    maxOccurs="1" type="xsd:string"/>
          <xsd:sequence>
            <xsd:element name="width" minOccurs="1"
     maxOccurs="1" type="xsd:string"/>
            <xsd:element name="high" minOccurs="0"
     maxOccurs="1" type="xsd:string"/>
        </xsd:sequence>
        <xsd:sequence>
          <xsd:element name="center" minOccurs="1"
     maxOccurs="1" type="xsd:string"/>
          <xsd:element name="width" minOccurs="0"
     maxOccurs="1" type="xsd:string"/>
        </xsd:sequence>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

先行する XSD は次の DataObject モデルを生成します。

```
DataObject - TieredGroup
 Property[0] - low - string
 Property[1] - width - string
 Property[2] - high - string
 Property[3] - high1 - string
 Property[4] - width1 - string
 Property[5] - high2 - string
 Property[6] - center - string
 Property[7] - width2 - string
```

ここで、**width**、**width1**、および **width2** は width というプロパティーの名前で、この XSD 内の最初のものから始まります。**high**、**high1**、**high2** も同様です。

これらの新しいプロパティー名は、参照と XPath に使用される名前にすぎず、シリアライズされた内容に影響しません。シリアライズされた XML 内に現れるこれらの各プロパティーの「真の」名前は、XSD 内で指定された値です。したがって、次の XML インスタンスの場合:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:TieredGroup xsi:type="p:TieredGroup"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://TieredGroup">
  <width>foo</width>
  <high>bar</high>
</p:TieredGroup>
```

これらのプロパティーにアクセスするためには、以下のコードを使用します。

```
DataObject tieredGroup = ...

// 「foo」を表示する
System.out.println(tieredGroup.get("width1"));

// 「bar」を表示する
System.out.println(tieredGroup.get("high2"));
```

**関連概念**

67 ページの『第 9 章 ビジネス・オブジェクト内の配列』
ビジネス・オブジェクト内のエレメントに配列を定義して、エレメントに複数のデータ・インスタンスを含めることができます。

**関連タスク**

73 ページの『モデル・グループ内のビジネス・オブジェクトの使用』
モデル・グループの一部であるネストされたビジネス・オブジェクトを処理するときは、モデル・グループのパス・パターンを使用する必要があります。

## 同じ名前がついたプロパティーの差別化

同じ名前空間を持つ複数の XSD で同じ名前のタイプが定義された場合、偶然に誤ったタイプが参照される可能性があります。

**Address1.xsd:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="city" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

Address2.xsd:
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="Address">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="state" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

ビジネス・オブジェクトは BOFactory.create() API を通じて、グローバル XSD 構造 (complexType、simpleType、element、attribute など) の重複名をサポートしていません。それでも、これらの重複したグローバル構造は、以下の例に示すように、適正な API が使用された場合でも、他の構造に対する子として作成される可能性があります。

**Customer1.xsd:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://Customer1"
 targetNamespace="http://Customer1">
  <xsd:import schemaLocation="./Address1.xsd"/>
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="address" type="Address"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

**Customer2.xsd:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://Customer2"
 targetNamespace="http://Customer2">
  <xsd:import schemaLocation="./Address2.xsd"/>
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element minOccurs="0" name="address" type="Address"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

両方の Customer address フィールドにデータを取り込んだ後、Address を作成するために BOFactory.create() を呼び出すと、結果としての子ビジネス・オブジェクトが誤って設定される可能性があります。これを避けるには、Customer DataObject に対して createDataObject(″address″) API を呼び出します。これにより、ビジネス・オブジェクトは import の schemaLocation に従うため、正しいタイプの子が生成されます。

```
DataObject customer1 = ...

// Address の子を作成する誤った方法
// これは Address1.xsd Address または Address2.xsd Address のタイプを
// 作成する場合がある
DataObject incorrect = boFactory.create("", "Address");
customer1.set("address", incorrect);

// Address の子を作成する正しい方法
// これにより Address1.xsd Address のタイプが作成されることが保証される
customer1.createDataObject("address");
```

## ピリオドを含んでいるプロパティー名の解決

XSD 内のプロパティー名には、多数の有効な文字の 1 つとしてピリオド (.) を含めることができますが、SDO では、ピリオドは複数カーディナリティーのプロパティー内で索引付けを示すためにも使用されます。このため、特定のシチュエーションで解決の問題が起きることがあります。

サービス・データ・オブジェクト (SDO) 内のプロパティー名は、XSD 内で生成されたエレメントと属性の名前に基づいています。ビジネス・オブジェクトは「.」文字を正しく処理しますが、1 つ例外があります。それは、XSD に「<name>.<#>」という名前の単一カーディナリティー・プロパティーと、「<name>」という名前の複数カーディナリティー・プロパティーがある場合です。

「foo.0」などの XPath は、「foo.0」という名前の単一カーディナリティー・プロパティーと「foo」という名前の複数カーディナリティー・プロパティーが存在する場合、正しく解決されません。その場合、「foo.0」という名前の単一カーディナリティー ・プロパティーに解決されます。これはまれにしか起きませんが、複数カーディナリティー・プロパティーにアクセスする場合に「foo[1]」の構文を使用すると、完全に回避できます。SDO は、索引付けに「.」構文をサポートしないので、索引付けには「[]」を使用してください。

## xsi:type での共用体のシリアライズとデシリアライズ

XSD では、共用体とは、メンバーと呼ばれるいくつかの単純データ型の字句スペースをマージする方法のことです。

以下の XSD の例は、整数と日付をメンバーとする共用体を示しています。

```
<xsd:simpleType name="integerOrDate">
 <xsd:union memberTypes="xsd:integer xsd:date"/>
</xsd:simpleType>
```

この複数のタイプ指定により、デシリアライゼーション時およびデータの操作時に混乱が起きる可能性があります。

ビジネス・オブジェクトは SDO でシリアライゼーションに xsi:type を使用することをサポートしており、デシリアライゼーションで XML データ内に xsi:type が存在しない場合は、同じアルゴリズムに従ってタイプを決定します。

したがって、データ (この例では数値の「42」) が整数としてデシリアライズされることを保証するために、入力 XML で指定された xsi:type を使用できます。また、整数がストリングの前に来るよう、XSD 内で共用体のメンバー・リストを順序付けることもできます。以下の例では、両方の方式の実装方法を示します。

```
<integerOrString xsi:type="xsd:integer">42</integerOrString>

<xsd:simpleType name="integerOrString">
 <xsd:union memberTypes="xsd:integer xsd:string"/>
</xsd:simpleType>
```

同様に、ユーザーがデータをストリングとしてデシリアライズさせる場合は、以下のいずれかの変更によって、そのような動作を起こさせることができます。

```
<integerOrString xsi:type="xsd:string">42</integerOrString>

<xsd:simpleType name="integerOrString">
 <xsd:union memberTypes="xsd:string xsd:integer"/>
</xsd:simpleType>
```

string タイプがこの共用体の最初のメンバーである場合は、決して情報が損失しないことに注意してください。また、この共用体は、xsi:type 以外のアルゴリズムで常に選択されるすべてのデータも保持できます。string 以外のタイプを使用する場合は、

XML の中で xsi:type を使用するか、XSD でメンバー・タイプの順序を変更して、他のメンバーがデータを受け入れられるようにする必要があります。

## Sequence オブジェクトを使用したデータの順序の設定

一部の XSD は、XML 内でのデータの出現順序が特別な重要性を持つような方法で定義されます。

XSD での順序の重要度を示す例の 1 つは、混合内容です。テキスト・データが、あるエレメントの前か後に出現する場合、そのデータは、別の場所で出現した場合とは異なる意味を持つことがあります。そのようなシチュエーションでは、SDO は Sequence というオブジェクトを生成します。このオブジェクトは、順序を付けてデータを設定するために使用されます。

SDO Sequence を XSD シーケンスと混同しないでください。XSD シーケンスは、SDO モデルの生成の前にフラット化される単なるモデル・グループです。XSD シーケンスの存在は、SDO Sequence の存在とは関係ありません。

XSD 内の以下の条件により、SDO Sequence が生成されます。

**混合内容を持つ complexType:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://MixedContent"
 targetNamespace="http://MixedContent">
  <xsd:complexType name="MixedContent" mixed="true">
    <xsd:sequence>
      <xsd:element name="element1" type="xsd:string" minOccurs="0"/>
      <xsd:element name="element2" type="xsd:string" minOccurs="0"/>
      <xsd:element name="element3" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="MixedContent" type="tns:MixedContent"/>
</xsd:schema>
```

**1 つ以上の <any/> タグを持つスキーマ:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://AnyElemAny"
 targetNamespace="http://AnyElemAny">
  <xsd:complexType name="AnyElemAny">
    <xsd:sequence>
      <xsd:any/>
      <xsd:element name="marker1" type="xsd:string"/>
      <xsd:any/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

**モデル・グループ配列 (maxOccurs > 1 の all、choice、sequence、または group 参照):**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://ModelGroupArray">
  <xsd:complexType name="ModelGroupArray">
    <xsd:sequence maxOccurs="3">
      <xsd:element name="element1" type="xsd:string"/>
```

```
      <xsd:element name="element2" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

**複数のエレメントを含んでいる maxOccurs <= 1 の <all/> タグ:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://All">
  <xsd:complexType name="All">
    <xsd:all>
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
</xsd:schema>
```

<any/> と sequence を一緒に使用することについての詳細は、このページの下部にリストしたトピックの中で説明します。このセクションで以下の残りの部分に示す一般情報は、その他の Sequence 条件の処理方法を説明したものですが、<any/> にも当てはまります。

**関連概念**

67 ページの『第 9 章 ビジネス・オブジェクト内の配列』
ビジネス・オブジェクト内のエレメントに配列を定義して、エレメントに複数のデータ・インスタンスを含めることができます。

## どうすれば DataObject にシーケンスがあるかどうかが分かるか

DataObject にシーケンスがあるかどうかを判別できる API は 2 つのうちから選択できます。DataObject noSequence と DataObject withSequence です。

DataObject noSequence および DataObject withSequence は、次の例のように使用します。

```
DataObject noSequence = ...
DataObject withSequence = ...

// false を表示する
System.out.println(noSequence.getType().isSequenced());

// true を表示する
System.out.println(withSequence.getType().isSequenced());

// true を表示する
System.out.println(noSequence.getSequence() == null);

// false を表示する
System.out.println(withSequence.getSequence() == null);
```

## なぜ DataObject に Sequence があることを知っていなければならないのか

Sequence がある DataObject に対して作業をする場合、データが設定される順序を知ることが重要です。そのため、値が設定される順序に注意する必要があります。

順序付けされていない DataObject では、ランダムな順序での設定アクセスができます。これは、すべてのキーが同じ値に設定される Map のように機能します。キー

が設定された順序は問題ではなく、マップ内のデータは同じものであり、XML へも完全に同じようにシリアライズされます。

DataObject が順序付けされている場合、データが設定された順序は、List へのデータの追加と同様に、Sequence 内に記録されます。これは、データへの 2 とおりのアクセス方法を提供します。名前/値のペアによるアクセス (DataObject API) と、設定された順序によるアクセス (Sequence API) です。DataObject set(...) API または Sequence add(...) API を使用して、構造を保存できます。この順序付けは、XML がシリアライズされる方法に影響を及ぼします。

例として、以下の <all/> タグを考えてみます。set メソッドが以下の順序で呼び出された場合、シリアライズされると、以下の XML が生成されます。

```
DataObject all = ...
all.set("element1", "foo");
all.set("element2", "bar");

<?xml version="1.0" encoding="UTF-8"?>
<p:All xsi:type="p:All"
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:p="http://All">
  <element1>foo</element1>
  <element2>bar</element2>
</p:All>
```

代わりに、set メソッドが逆の順序で呼び出された場合は、このビジネス・オブジェクトがシリアライズされると、以下の XML が生成されます。

```
DataObject all = ...
all.set("element2", "bar");
all.set("element1", "foo");

<?xml version="1.0" encoding="UTF-8"?>
<p:All xsi:type="p:All"
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:p="http://All">
  <element2>bar</element2>
  <element1>foo</element1>
</p:All>
```

Sequence の順序を変更する必要がある場合、Sequence クラスにはユーザーが Sequence の順序を変更できるよう、基本的な add、remove、および move メソッドがあります。

## どのようにして混合内容を処理するか

混合内容の場合、Sequence にはテキストを追加するための固有の API である addText(...) があります。

それ以外のすべての API は、テキストに対して Property の場合と同じように機能します。getProperty(int) API は、混合内容テキスト・データについて、ヌルを返します。以下の混合内容コードの例は、DataObject からのすべての混合内容テキストを印刷するために使用できます。

```
DataObject mixedContent = ...
Sequence seq = mixedContent.getSequence();

for (int i=0; i < seq.size(); i++)
{
    Property prop = seq.getProperty(i);
```

```
    Object value = seq.getValue(i);

    if (prop == null)
    {
        System.out.println("Found mixed content text: "+value);
    }
    else
    {
        System.out.println("Found Property "+prop.getName()+": "+value);
    }
}
```

## どのようにしてモデル・グループ配列を処理するか

モデル・グループ配列は、モデル・グループが maxOccurs > 1 の値を持っているときに作成されます。

モデル・グループはフラット化され、DataObject の中で表現されないため、モデル・グループの内部のプロパティーは複数カーディナリティー・プロパティーになり、これにより isMany() メソッドは、まだ true でなくても true を返します。それらの minOccurs および maxOccurs ファセットは、収容しているモデル・グループのファセットによって乗算されます。Choice は、他のモデル・グループと同じように maxOccurs ファセットを乗算しますが、minOccurs の乗算値として常に 0 を使用します。なぜなら、choice 内のいずれかのデータは選択されない場合があるからです。

例えば、以下の XSD にはモデル・グループ配列があります。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ModelGroupArray">
  <xsd:complexType name="ModelGroupArray">
    <xsd:sequence minOccurs="2" maxOccurs="5">
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"
   minOccurs="0" maxOccurs="3"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

前に述べたように、**element1** と **element2** は get(...) アクセサーが List を返すよう、この時点で複数カーディナリティーになります。**Element1** の minOccurs はデフォルトの 1 で、maxOccurs もデフォルトの 1 です。**Element2** の minOccurs は 0 で、maxOccurs は 3 です。次の例では、それらの新しい minOccurs と maxOccurs は、以下のようになります。

```
DataObject - ModelGroupArray
 Property[0] - element1 - minOccurs=(2*1)=2 - maxOccurs=(5*1)=5
 Property[1] - element2 - minOccurs=(2*0)=0 - maxOccurs=(5*3)=15
```

タイプが Choice だとすると:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://ModelGroupArray">
  <xsd:complexType name="ModelGroupArray">
    <xsd:choice minOccurs="2" maxOccurs="5">
      <xsd:element name="element1" type="xsd:string"/>
      <xsd:element name="element2" type="xsd:string"
```

```
   minOccurs="0" maxOccurs="3"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

は、以下の minOccurs を生成します。これは、**element1** だけが毎回選択されるか **element2** だけが毎回選択される choice の除外によるもので、したがって、妥当性検査をパスするためには、どちらも 0 個のオカレンスを持つことができなければなりません。

```
DataObject - ModelGroupArray
 Property[0] - element1 - minOccurs=(0*1)=0 - maxOccurs=(5*1)=5
 Property[1] - element2 - minOccurs=(0*0)=0 - maxOccurs=(5*3)=15
```

**関連タスク**

73 ページの『モデル・グループ内のビジネス・オブジェクトの使用』
モデル・グループの一部であるネストされたビジネス・オブジェクトを処理するときは、モデル・グループのパス・パターンを使用する必要があります。

## 単純タイプへの AnySimpleType の使用

SDO API による AnySimpleType の処理は、他のいずれの単純タイプ (string、int、boolean など) とも異なる点はありません。

anySimpleType が他の単純タイプと異なるのは、そのインスタンス・データとシリアライゼーション/デシリアライゼーションだけです。これらはビジネス・オブジェクトの内部だけの概念にすべきであり、フィールドへまたはフィールドからマップされるデータが有効であるかどうかを判別するために使用してください。string タイプに対して set(...) メソッドを呼び出す場合は、最初にデータがストリングに変換され、元のデータ・タイプは失われます。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://StringType">
  <xsd:complexType name="StringType">
    <xsd:sequence>
      <xsd:element name="foo" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

 DataObject stringType = ...

 // データを String に設定する
 stringType.set("foo", "bar");

 // インスタンス・データは、データ・セットに関係なく、常に String タイプになる
 // 「java.lang.String」を表示する
 System.out.println(stringType.get("foo").getClass().getName());

 // データを Integer に設定する
 stringType.set("foo", new Integer(42));

 // インスタンス・データは、データ・セットに関係なく、常に String タイプになる
 // 「java.lang.String」を表示する
 System.out.println(stringType.get("foo").getClass().getName());
```

その代わり、anySimpleType は設定される元のデータ・タイプを失いません。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://AnySimpleType">
  <xsd:complexType name="AnySimpleType">
    <xsd:sequence>
      <xsd:element name="foo" type="xsd:anySimpleType"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

 DataObject anySimpleType = ...

 // データを String に設定する
 stringType.set("foo", "bar");

 // インスタンス・データは常に、set 内で使用される日付のタイプになる
 // 「java.lang.String」を表示する
 System.out.println(stringType.get("foo").getClass().getName());

 // データを Integer に設定する
 stringType.set("foo", new Integer(42));

 // インスタンス・データは常に、set 内で使用される日付のタイプになる
 // 「java.lang.Integer」を表示する
 System.out.println(stringType.get("foo").getClass().getName());
```

このデータ・タイプも、シリアライゼーションおよびデシリアライゼーションをまたがっても、xsi:type によって保存されます。したがって、anySimpleType エレメントをシリアライズするときはいつでも、その Java タイプに基づいた SDO 仕様で定義されているものに一致する xsi:type が存在します。

次の例では、データが以下のようになるよう、上記のビジネス・オブジェクトをシリアライズします。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:StringType xsi:type="p:StringType"
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xmlns:xsd=http://www.w3.org/2001/XMLSchema
 xmlns:p="http://StringType">
  <foo xsi:type="xsd:int">42</foo>
<p:StringType></p:StringType>
```

xsi:type は、デシリアライゼーションのときに、データを適切な Java インスタンス・クラスとしてロードするために使用されます。xsi:type が指定されていない場合、デフォルトのデシリアライゼーション・タイプは string になります。

他の単純タイプの場合、マップ可能性の判別は常に一定です。例えば、boolean は常に string にマップできます。しかし、AnySimpleType には任意の単純タイプを含めることができるので、フィールド内のインスタンス・データによって、マッピングが可能な場合とそうでない場合があります。

プロパティーが anySimpleType タイプであるかどうかを判別するには、プロパティー Type の URI と Name を使用します。それらは、「commonj.sdo」と「Object」になります。anySimpleType に挿入しても有効なデータかどうかを判別するには、それが DataObject のインスタンスでないかどうかを調べます。String として表現でき、DataObject でないすべてのデータは、anySimpleType フィールド内に設定できます。

したがって、以下のマッピング・ルールが導出されます。

- anySimpleType は、常に anySimpleType にマップできます。
- それ以外のすべての単純タイプは、常に anySimpleType にマップできます。
- anySimpleType は、常に string にマップできます。すべての単純タイプは string に変換できなければならないからです。
- anySimpleType は、ビジネス・オブジェクト内のその値に応じて、他のいずれかの単純タイプにマップできる場合とそうでない場合があります。したがって、このマッピングは設計時には判別できず、実行時にのみ判別できます。

**関連情報**


Assigning from and to xs:any

## 複合タイプへの AnyType の使用

SDO API による anyType タグの処理は、他のいずれの複合タイプとも異なる点はありません。

anyType が他の複合タイプと異なるのは、そのインスタンス・データとシリアライゼーション/デシリアライゼーションだけです。これらはビジネス・オブジェクトのみに対する内部概念であり、フィールドへまたはフィールドからマップされるデータが valid.Complex タイプであるかどうかの判別は、単一タイプ (Customer、Address など) に限られます。しかし、anyType は、タイプに関係なく、すべての DataObject を許容します。maxOccurs > 1 の場合、リスト内の各 DataObject は異なるタイプであってもかまいません。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://AnyType">
  <xsd:complexType name="AnyType">
    <xsd:sequence>
      <xsd:element name="person" type="xsd:anyType"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://Customer">
  <xsd:complexType name="Customer">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://Employee" targetNamespace="http://Employee">
  <xsd:complexType name="Employee">
    <xsd:sequence>
      <xsd:element name="id" type="xsd:string"/>
 </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

 DataObject anyType = ...
 DataObject customer = ...
 DataObject employee = ...
```

```
// person を Customer に設定する
anyType.set("person", customer);

// インスタンス・データは Customer になる
// 「Customer」を表示する
System.out.println(anyType.getDataObject("person").getName());

// person を Employee に設定する
anyType.set("person", employee);

// インスタンス・データは Employee になる
// 「Employee」を表示する
System.out.println(anyType.getDataObject("person").getName());
```

anySimpleType とまったく同じように、anyType はシリアライゼーション時に xsi:type を使用して、DataObject の意図されたタイプがデシリアライゼーション時に維持されるようにします。したがって、「Customer」に設定した場合、XML は次のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:customer="http://Customer"
    xmlns:p="http://AnyType">
  <person xsi:type="customer:Customer">
    <name>foo</name>
  </person>
</p:AnyType>
```

また、「Employee」に設定した場合は、以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:employee="http://Employee"
    xmlns:p="http://AnyType">
  <person xsi:type="employee:Employee">
    <id>foo</id>
  </person>
</p:AnyType>
```

また、AnyType では、ラッパー DataObject を通じて単純タイプの値を設定できます。それらのラッパー DataObjects は、「value」(エレメント) という、単純タイプ値を保持する単一プロパティーを持ちます。SDO API は、get<Type>/set<Type> API を使用した場合、これらの単純タイプとラッパー DataObject を自動的にラップおよびアンラップするためにオーバーライドされています。型キャストのない get/set API は、このラッピングを行いません。

```
DataObject anyType = ...

// set<Type> API を anyType Property について呼び出すと、自動的に
// ラッパー DataObject が作成される
anyType.setString("person", "foo");

// 通常の get/set API は、オーバーライドされないので、
// ラッパー DataObject を返す
DataObject wrapped = anyType.get("person");

// ラップされた DataObject は「value」Property を持つ
// 「foo」を表示する
System.out.println(wrapped.getString("value"));
```

```
// get<Type> API は自動的に DataObject をアンラップする
// 「foo」を表示する
System.out.println(anyType.getString("person"));
```

ラッパー DataObject は、シリアライズされる場合、Java インスタンス・クラスの anySimpleType マッピングとまったく同じように、xsi:type フィールドで XSD タイプへシリアライズされます。したがって、この設定は、次のようにシリアライズされます。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:AnyType xsi:type="p:AnyType"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:p="http://AnyType">
  <person xsi:type="xsd:string">foo</person>
</p:AnyType>
```

xsi:type が指定されていないか、指定された xsi:type が正しくない場合は、例外がスローされます。自動ラッピングのほかに、BOFactory createDataTypeWrapper(Type, Object) により、set() API で使用するラッパーを手動で作成できます。ここで、Type はラップされるデータの SDO 単純タイプで、Object は、ラップされるデータです。

```
Type stringType = boType.getType("http://www.w3.org/2001/XMLSchema", "string");
DataObject stringType = boFactory.createByMessage(stringType, "foo");
```

DataObject がラッパー・タイプであるかどうかを判別するには、BOType isDataTypeWrapper(Type) を呼び出すことができます。

```
DataObject stringType = ...
boolean isWrapper = boType.isDataTypeWrapper(stringType.getType());
```

他の複合タイプの場合、フィールド間でデータを移動するには、データが同じタイプである必要があります。しかし、AnyType には任意の複合タイプを含めることができるので、フィールド内のインスタンス・データによって、マッピングなしの直接の移動が可能な場合と可能でない場合があります。

プロパティー Type の URI と Name を使用して、プロパティーが anyType タイプであるかどうかを判別できます。それらは、「commonj.sdo」と「DataObject」になります。すべてのデータは、anyType に挿入するのに有効です。したがって、以下のマッピング・ルールが導出されます。

- anyType は、常に anyType にマップできます。
- すべての複合タイプは、常に anyType にマップできます。
- すべての単純タイプは、常に anyType にマップできます。
- anyType は、BO インスタンス内のその値に応じて、他のいずれかの単純タイプまたは複合タイプにマップできる場合とそうでない場合があります。したがって、このマッピングは設計時には判別できず、実行時にのみ判別できます。

## 複合タイプのグローバル・エレメントを設定するための Any の使用

<any/> タグを使用して、複合タイプにグローバル・エレメントを設定できます。

any タグがあると、DataObject Type isOpen() メソッドおよび isSequenced() メソッドは true を返します。any タグ上で maxOccurs の値が > 1 の場合、DataObject の構造に影響はありません。そのタグは、妥当性検査時の情報としてのみ使用されます。同様に、type 内に複数の any タグがあっても、DataObject の構造は変更されません。それらのタグは、設定されたオープン・データの場所の妥当性検査にのみ使用されます。

**関連タスク**

72 ページの『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』
タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

## どうすれば DataObject に any タグがあるかどうかが分かるか

DataObject のインスタンス内に any 値が設定されているかどうかは、インスタンスのプロパティーを調べて、いずれかのオープン・プロパティーが属性であるかどうかを確認すれば、簡単に判別できます。

DataObject は DataObject Type に any タグがあるかどうかを判別するメカニズムを備えていません。DataObject には、any と anyAttribute の両方に適用され、any のプロパティーを自由に追加できる「オープン」の概念だけがあります。any タグが存在すると DataObject は isOpen() = true で isSequenced() = true になりますが、単にその DataObject に anyAttribute タグと、Sequence のトピックで述べた順序付けされる理由の 1 つがあるにすぎない場合も考えられます。次の例は、それらの概念を実際に示したものです。

```
DataObject dobj = ...

// タイプがオープンであるかどうかを検査し、そうでなければ
// 中に any 値が設定されていることはありえない。
boolean isOpen = dobj.getType().isOpen();

if (!isOpen) return false;  // any 値は設定されていない

// オープン・プロパティーはインスタンス・プロパティー・リストに追加されるが
// プロパティー・リストには追加されないため、それらのサイズを比較すれば
// 設定されたオープン・データがあるかどうかを簡単に判別できる
int instancePropertyCount = dobj.getInstanceProperties().size();
int definedPropertyCount = dobj.getType().getProperties().size();

// 同じであれば、オープン・コンテンツは設定されていない
if (instancePropertyCount == definedPropertyCount) return false;

// オープン・コンテンツの Property を調べて、any が Element であるかどうかを判別する
for (int i=definedPropertyCount; i < instancePropertyCount; i++)
{
    Property prop = (Property)dobj.getInstanceProperties().get(i);
    if (boXsdHelper.isElement(prop))
    {
        return true;  // any 値が検出された
    }
}

return false;  // any 値は設定されていない
```

## どのようにして any の値を取得/設定するか

any フィールドに設定されたデータの取得は、名前が分かっている場合、他のすべてのエレメント値と同じように行うことができます。

XPath の「<name>」で get を行うことができ、それは解決されます。名前が不明の場合は、上記のようにインスタンス・プロパティーを調べることにより、値を見つけることができます。複数の any タグが存在するか、maxOccurs > 1 の any タグが存在する場合、データの発生元である any タグを特定することが重要であるなら、DataObject sequence を代わりに使用する必要があります。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://AnyElemAny"
 targetNamespace="http://AnyElemAny">
  <xsd:complexType name="AnyElemAny">
    <xsd:sequence>
      <!-- Handle all these any one way -->
      <xsd:any maxOccurs="3"/>
      <xsd:element name="marker1" type="xsd:string"/>
      <!-- Handle this any in another -->
      <xsd:any/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

<any/> タグは DataObject を順序付けするので、どの any 値が設定されたかの判別は、Sequence 内の any プロパティーの位置を調べることによって行うことができます。

以下の XSD のインスタンス・データが属する any タグを判別するには、以下のコードを使用します。

```
DataObject anyElemAny = ...
Seqeuence seq = anyElemAny.getSequence();

// marker1 エレメントを検出するまで、見つかったすべてのオープン・データは
// 最初の any タグに属する
boolean foundMarker1 = false;

for (int i=0; i<seq.size(); i++)
{
    Property prop = seq.getProperty(i);

    // プロパティーがオープン・プロパティーであるかどうかをチェックする
    if (prop.isOpenContent())
    {
        if (!foundMarker1)
        {
            // 最初の any でなければならない
            // marker1 エレメントの前にあるからである
            System.out.println("Found first any data: "+seq.getValue(i));
        }
        else
        {
            // 2 番目の any でなければならない
            // marker1 エレメントの後にあるからである
            System.out.println("Found second any data: "+seq.getValue(i));
         }
    }
    else
    {
        // marker1 エレメントでなければならない
```

```
            System.out.println("Found marker1 data: "+seq.getValue(i));
            foundMarker1 = true;
        }
    }
```

<any/> 値の設定は、グローバル・エレメント・プロパティーを作成し、その値をシーケンスに追加することによって行います。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://GlobalElems"
 targetNamespace="http://GlobalElems">
  <xsd:element name="globalElement1" type="xsd:string"/>
  <xsd:element name="globalElement2" type="xsd:string"/>
</xsd:schema>

 DataObject anyElemAny = ...
 Seqeuence seq = anyElemAny.getSequence();

 // globalElement1 のグローバル・エレメント Property を取得する
 Property globalProp1 = boXsdHelper.getGlobalProperty(http://GlobalElems,
 "globalElement1", true);

 // globalElement2 のグローバル・エレメント Property を取得する
 Property globalProp2 = boXsdHelper.getGlobalProperty(http://GlobalElems,
 "globalElement2", true);

 // データを最初の any のシーケンスに追加する
 seq.add(globalProp1, "foo");
 seq.add(globalProp1, "bar");

 // marker1 のデータを追加する
 seq.add("marker1", "separator");  // または anyElemAny.set("marker1", "separator")

 // データを 2 番目の any のシーケンスに追加する
 seq.add(globalProp2, "baz");

 // これで、get 呼び出しによってデータにアクセスできる
 System.out.println(dobj.get("globalElement1"); //  「[foo, bar]」を表示する
 System.out.println(dobj.get("marker1");        //  「separator」を表示する
 System.out.println(dobj.get("globalElement2"); //  「baz」を表示する
```

**関連タスク**

72 ページの『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』

タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

## any でのデータの有効なマッピングはどのようなものか

<any/> タグは、名前/値のペアの集合です。設計時に <any/> に決定できる唯一の有効なマッピングは、別の <any/> か、同じ maxOccurs 値を持つ anyType です。

any の DataObject のインスタンスに含まれている個々の値は、複合タイプ・マッピングのすべてのルールに従う基本複合タイプです。それらの複合タイプの一部は、ラップされた単純タイプの場合があり、したがって、それらは単純タイプ・マッピングのルールに従います。

## 複合タイプのグローバル属性を設定するための AnyAttribute の使用

<anyAttribute/> タグを使用すると、複合タイプに任意の数のグローバル属性を設定できます。

<any/> タグと同様に、<anyAttribute/> タグがあると、DataObject Type isOpen() メソッドは true を返します。しかし、<any/> タグとは異なり、<anyAttribute/> タグは DataObject を順序付きにしません。XSD 内の属性は順序付きの構成体ではないからです。

**関連タスク**

72 ページの『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』
タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

## どうすれば DataObject に anyAttribute タグがあるかどうかが分かるか

DataObject のインスタンス内に anyAttribute 値が設定されているかどうかは、インスタンスのプロパティーを調べて、いずれかのオープン・プロパティーが属性であるかどうかを確認すれば、簡単に判別できます。

DataObject は DataObject Type に anyAttribute タグがあるかどうかを判別するメカニズムを備えていません。DataObject には、any と <anyAttribute/> の両方に適用され、any の Property を自由に追加できる「オープン」の概念だけがあります。DataObject has isOpen() = true で isSequenced() = false の場合には anyAttribute タグがなければならないということは正しいのですが、isOpen() = true で isSequenced() = true の場合には、DataObject Type は anyAttribute タグを持つ場合と持たない場合があります。

DataObject は、その DataObject を生成するために使用された XSD 構造に関するこれらの疑問にプログラマチックに回答する、メタデータ照会メソッドを備えています。必要であれば、InfoSet モデルに照会して、anyAttribute タグが存在するかどうかを知ることができます。anyAttribute は単数形で、true かそうでないかであるため、ビジネス・オブジェクトは BOXSDHelper hasAnyAttribute(Type) メソッドも提供して、この DataObject にオープン属性を設定しても有効な結果が得られるかどうかを判別できるようにします。次のコード例は、それらの概念を実際に示したものです。

```
DataObject dobj = ...

// タイプがオープンであるかどうかを検査し、そうでなければ
// 中に anyAttribute 値が設定されていることはありえない。
boolean isOpen = dobj.getType().isOpen();

if (!isOpen) return false;  // anyAttribute 値は設定されていない

// オープン・プロパティーはインスタンス・プロパティー・リストに追加されるが
// プロパティー・リストには追加されないため、それらのサイズを比較すれば
// 設定されたオープン・データがあるかどうかを簡単に判別できる
int instancePropertyCount = dobj.getInstanceProperties().size();
int definedPropertyCount = dobj.getType().getProperties().size();

// 同じであれば、オープン・コンテンツは設定されていない
```

```
if (instancePropertyCount == definedPropertyCount) return false;

// オープン・コンテンツの Property を調べて、any が Attribute であるかどうかを判別する
for (int i=definedPropertyCount; i<instancePropertyCount; i++)
{
    Property prop = (Property)dobj.getInstanceProperties().get(i);
    if (boXsdHelper.isAttribute(prop))
    {
        return true;  // anyAttribute 値が検出された
    }
}

return false;  // anyAttribute 値は設定されていない
```

**関連タスク**

72 ページの『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』
タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

## どのようにして anyAttribute の値を取得/設定するか

<anyAttribute/> 値の設定は、<any/> と同じ方法で行いますが、グローバル・エレメントの代わりにグローバル属性を使用します。

anyAttribute フィールドに設定されたデータの取得は、名前が分かっている場合、他のすべての属性値と同じように行うことができます。XPath の「@<name>」で get を行うことができ、それは解決されます。名前が不明の場合は、上記のコードを使用すると、値に 1 つずつ反復してアクセスできます。次のコード例は、それを示しています。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://AnyAttrOnlyMixed"
 targetNamespace="http://AnyAttrOnly">
  <xsd:complexType name="AnyAttrOnly">
    <xsd:sequence>
      <xsd:element name="element" type="xsd:string"/>
    </xsd:sequence>
    <xsd:anyAttribute/>
  </xsd:complexType>
</xsd:schema>

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://GlobalAttrs">
  <xsd:attribute name="globalAttribute" type="xsd:string"/>
</xsd:schema>

 DataObject dobj = ...

 // 設定しようとしているグローバル属性 Property を取得する
 Property globalProp = boXsdHelper.getGlobalProperty(http://GlobalAttrs,
 "globalAttribute", false);

 // 他のすべてのデータとまったく同じように、dobj に値を設定する
 dobj.set(globalProp, "foo");

 // これで、get 呼び出しによってデータにアクセスできる
 System.out.println(dobj.get("@globalAttribute")); // 「foo」を表示する
```

**関連タスク**

72 ページの『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』
タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

## anyAttribute でのデータの有効なマッピングはどのようなものか

AnyAttribute タグは、すべてのタグと同様に、一連の名前/値のペアから構成されています。したがって、anyAttribute の唯一の有効なマッピングは、別の anyAttribute です。

anyAttribute データに含まれている個々の値は、単純タイプ・マッピングのすべてのルールに従う基本単純タイプです。

# 第 9 章 ビジネス・オブジェクト内の配列

ビジネス・オブジェクト内のエレメントに配列を定義して、エレメントに複数のデータ・インスタンスを含めることができます。

リスト型を使用して、ビジネス・オブジェクト内の単一の名前付きエレメントに配列を作成することができます。これによって、このエレメントに複数のデータ・インスタンスを格納することができます。例えば、ビジネス・オブジェクト・ラッパー内にストリングとして定義した telephone という名前のエレメント内に、配列を使用して複数の電話番号を保管することができます。また、maxOccurs の値を使用してデータ・インスタンスの数を指定することによって、配列のサイズを定義することもできます。次のコード例は、エレメントに 3 つのデータ・インスタンスを保持する配列を作成する方法を示したものです。

```
<xsd:element name="telephone" type="xsd:string" maxOccurs="3"/>
```

これは、最大 3 つのデータ・インスタンスを保持できるエレメント telephone のリスト指標を作成します。項目が 1 つだけの配列を計画している場合は、値 minOccurs も使用できます。

結果の配列は、以下の 2 つの項目で構成されます。

- 配列の内容
- 配列そのもの

ただし、この配列を作成するには、ラッパーを定義して中間ステップを実行する必要があります。このラッパーは実際には、エレメントのプロパティーを配列オブジェクトで置換します。上記の例では、ArrayOfTelephone オブジェクトを作成して、エレメント telephone を配列として定義することができます。次のコード例は、このタスクを行う方法を示したものです。

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xsd:element name="Customer">
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="name" type="xsd:string"/>
                    <xsd:element name="ArrayOfTelephone" type="ArrayOfTelephone"/>
         </xsd:sequence>
       </xsd:complexType>
    </xsd:element>

        <xsd:complexType name="ArrayOfTelephone">
       <xsd:sequence maxOccurs="3">
                <xsd:element name="telephone" type="xsd:string" nillable="true"/>
       </xsd:sequence>
    </xsd:complexType>
 </xsd:schema>
```

telephone エレメントは、ArrayOfTelephone ラッパー・オブジェクトの子として表現されています。

© Copyright IBM Corp. 2005, 2008

上の例では、telephone エレメントには nillable という名前のプロパティーが組み込まれています。配列指標の特定項目にデータを含めたくない場合は、このプロパティーを true に設定することができます。次のコード例は、配列内でのデータの表現方法を示したものです。

```
<Customer>
      <name>Bob</name>
      <ArrayOfTelephone>
          <telephone>111-1111</telephone>
          <telephone xsi:nil="true"/>
          <telephone>333-3333</telephone>
      </ArrayOfTelephone>
  </Customer>
```

このケースでは、telephone エレメントの配列指標の 1 番目と 3 番目の項目にはデータが格納されていますが、2 番目の項目にはデータが格納されていません。nillable プロパティーを telephone エレメントで使用しなかった場合、最初の 2 つのエレメントにデータを格納する必要があります。

WebSphere Process Server のサービス・データ・オブジェクト (SDO) の Sequence API を、ビジネス・オブジェクト配列内のシーケンスを処理する代替の方法として使用することができます。以下のコード例によって、上記で示したものと同じデータを持つ telephone エレメントの配列が作成されます。

```
DataObject customer = ...
  customer.setString("name", "Bob");

  DataObject tele_array = customer.createDataObject("ArrayOfTelephone");
  Sequence seq = tele_array.getSequence();  // 配列は順序付けされます
  seq.add("telephone", "111-1111");
  seq.add("telephone", null);
  seq.add("telephone", "333-3333");
```

以下の例のようなコードを使用して、特定のエレメント配列指標のデータを戻すことができます。

```
String tele3 = tele_array.get("telephone[3]");  // tele3 = "333-3333"
```

この例では、tele3 という名前のストリングによって、データ「333-3333」が戻されます。

リスト指標の配列のデータ項目には、JMS または MQ メッセージ・キュー内に配置されている固定幅または区切り文字で区切られているデータを使用して入力できます。また、適切にフォーマット設定されたデータが格納されているフラット・テキスト・ファイルを使用して、このタスクを行うこともできます。

**関連概念**

51 ページの『Sequence オブジェクトを使用したデータの順序の設定』
一部の XSD は、XML 内でのデータの出現順序が特別な重要性を持つような方法で定義されます。

46 ページの『モデル・グループ・サポート (all、choice、sequence、および group 参照)』
SDO 仕様は、モデル・グループ (all、choice、sequence、および group 参照) を正しい位置に展開して、タイプやプロパティーを記述しないことを必要としています。

# 第 10 章 ネストされたビジネス・オブジェクトの作成

setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトを親ビジネス・オブジェクト内に作成することができます。

中間の子オブジェクトの詳細を記載したコードを記述せずに、ネストされたビジネス・オブジェクトを親ビジネス・オブジェクトから作成できます。例えば、親ビジネス・オブジェクトの 1 レベル下に依存ビジネス・オブジェクトを定義せずに、親ビジネス・オブジェクトの 2 レベル下にネストされたビジネス・オブジェクトを設定できます。以下について、setWithCreate 機能を使用してこのタスクを実行します。

- 単一インスタンス
- 複数インスタンス
- ワイルドカード値
- モデル・グループ

以下のトピックでは、これらのそれぞれについて実行する方法を説明します。

## ネストされたビジネス・オブジェクトの単一インスタンス

setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの単一インスタンスを作成します。

### 始める前に

以下のコード例は、第 3 レベル (孫) オブジェクトを作成するために、高レベル (親) オブジェクトから中間 (子) オブジェクトのコードを通常であればどのように作成する必要があるのかを示したものです。XSD ファイルは以下のようになります。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="Child"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Child">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="grandChild" type="GrandChild"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="GrandChild">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
```

© Copyright IBM Corp. 2005, 2008

```
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

### このタスクについて

従来の「トップダウン」方式を使用してビジネス・オブジェクト・データを設定する場合、孫オブジェクトにデータを設定する前に、子オブジェクトおよび孫オブジェクトを指定する以下のコードを処理する必要があります。

```
DataObject parent = ...
DataObject child = parent.createDataObject("child");
DataObject grandchild = child.createDataObject("grandChild");
grandchild.setString("name", "Bob");
```

setWithCreate 機能の使用によって、より効率的な方式を使用することができます。この方式では、孫オブジェクトの定義およびオブジェクトのデータの設定を同時に実行でき、中間の子オブジェクトの指定は不要です。次のコード例は、このタスクを行う方法を示したものです。

```
DataObject parent = ...
parent.setString("child/grandchild/name", "Bob");
```

### タスクの結果

中間レベルのビジネス・オブジェクトを参照することなく、下位のビジネス・オブジェクト・データが設定されます。パスが無効な場合は例外が発生します。

**関連タスク**

『ネストされたビジネス・オブジェクトの複数インスタンスの作成』
setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの複数インスタンスを作成します。

72 ページの『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』
タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

73 ページの『モデル・グループ内のビジネス・オブジェクトの使用』
モデル・グループの一部であるネストされたビジネス・オブジェクトを処理するときは、モデル・グループのパス・パターンを使用する必要があります。

## ネストされたビジネス・オブジェクトの複数インスタンスの作成

setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの複数インスタンスを作成します。

### 始める前に

以下の XSD ファイルの例には、トップ (親) のビジネス・オブジェクトの 1 レベル下 (子) および 2 レベル下 (孫) のネストされたオブジェクトが含まれています。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
```

```
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="Child" maxOccurs="5"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="Child">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="grandChild" type="GrandChild"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="GrandChild">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

maxOccurs の値で指定するように、親オブジェクトは最大 5 つの子オブジェクトを持つことができることに注意してください。

## このタスクについて

配列内で欠落したシーケンスを許可しない、より厳密なポリシーでリストを作成することができます。setWithGet メソッドを使用して、同時に特定のリスト指標項目に現れるデータを指定します。

```
DataObject parent = ...
parent.setString("child[3]/grandchild/name", "Bob");
```

この事例では、結果の配列のサイズは 3 ですが、child[1] および child[2] のリスト指標項目の値は未定義です。項目は、ヌル値にするか、または関連したデータ値を持たせることが必要な場合があります。上記のシナリオでは、最初の 2 つの配列指標項目が未定義であるため、例外がスローされます。

リストの指標の値を定義することで、この状況に対処することができます。指標項目が配列内の既存のエレメントを参照しており、その参照先エレメントがヌル以外である場合 (つまりデータを含む場合)、それが使用されます。参照先エレメントがヌルである場合、そのエレメントが作成されて使用されます。リストの指標がリストのサイズよりも 1 つ大きい場合、新規の値が作成されて追加されます。以下のコード例では、サイズが 2 のリストで、child[1] をヌルに指定し、child[2] にデータを格納した場合に何が起こるかを示したものです。

```
DataObject parent = ...
// child[1] = ヌル
// child[2] = 既存の子
// child[1] はヌルであり、作成されることから、このコードは機能します。
parent.setString("child[1]/grandchild/name", "Bob");

// child[2] が存在して使用されるため、このコードは機能します。
parent.setString("child[2]/grandchild/name", "Dan");

// 子のリストのサイズは 2 であり、
// リスト項目を 1 つ追加するとリストのサイズが増加するため、このコードは機能します。
parent.setString("child[3]/grandchild/name", "Sam");
```

### タスクの結果

2 つの既存項目の値を指定変更して、3 つ目の項目をリスト指標に追加しました。ただし、次にサイズが 4 でない別の項目か、または maxOccurs で指定したサイズよりも大きな項目を追加すると、例外がスローされます。このメソッドのより厳密なポリシーについて、以下のコード例で示します。

**注:** 以下のコードは、上記の既存のコードに追加することを想定しています。

```
// リストはサイズが 3 であり、
// サイズを 4 に増加させる項目を作成していなかったため、このコードは例外をスローします。
parent.setString("child[5]/grandchild/name", "Billy");
```

**関連タスク**

69 ページの『ネストされたビジネス・オブジェクトの単一インスタンス』
setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの単一インスタンスを作成します。

『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』
タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

73 ページの『モデル・グループ内のビジネス・オブジェクトの使用』
モデル・グループの一部であるネストされたビジネス・オブジェクトを処理するときは、モデル・グループのパス・パターンを使用する必要があります。

## ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用

タイプ xsd:any を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

### このタスクについて

サービス・データ・オブジェクト内でワイルドカードの値 xsd:any を使用している場合、単一および複数のインスタンスのためにネストされたビジネス・オブジェクトを定義する目的で使用される setWithCreate 機能は作用しません。これについて、以下のコード例で説明します。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:complexType name="Parent">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="child" type="xsd:anyType"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

### タスクの結果

子データ・オブジェクトが存在しない場合、例外がスローされます。

**関連概念**

63 ページの『どうすれば DataObject に anyAttribute タグがあるかどうかが分かるか』

DataObject のインスタンス内に anyAttribute 値が設定されているかどうかは、インスタンスのプロパティーを調べて、いずれかのオープン・プロパティーが属性であるかどうかを確認すれば、簡単に判別できます。

63 ページの『複合タイプのグローバル属性を設定するための AnyAttribute の使用』
<anyAttribute/> タグを使用すると、複合タイプに任意の数のグローバル属性を設定できます。

61 ページの『どのようにして any の値を取得/設定するか』
any フィールドに設定されたデータの取得は、名前が分かっている場合、他のすべてのエレメント値と同じように行うことができます。

64 ページの『どのようにして anyAttribute の値を取得/設定するか』
<anyAttribute/> 値の設定は、<any/> と同じ方法で行いますが、グローバル・エレメントの代わりにグローバル属性を使用します。

59 ページの『複合タイプのグローバル・エレメントを設定するための Any の使用』
<any/> タグを使用して、複合タイプにグローバル・エレメントを設定できます。

**関連タスク**

69 ページの『ネストされたビジネス・オブジェクトの単一インスタンス』
setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの単一インスタンスを作成します。

70 ページの『ネストされたビジネス・オブジェクトの複数インスタンスの作成』
setWithCreate 機能を使用して、ネストされたビジネス・オブジェクトの複数インスタンスを作成します。

『モデル・グループ内のビジネス・オブジェクトの使用』
モデル・グループの一部であるネストされたビジネス・オブジェクトを処理するときは、モデル・グループのパス・パターンを使用する必要があります。

## モデル・グループ内のビジネス・オブジェクトの使用

モデル・グループの一部であるネストされたビジネス・オブジェクトを処理するときは、モデル・グループのパス・パターンを使用する必要があります。

### このタスクについて

モデル・グループでは、親ビジネス・オブジェクトからビジネス・オブジェクトを作成するのに使用できるタグ xsd:choice を使用します。しかし、Eclipse Modeling Framework (EMF) では、例外を生成する可能性がある名前の競合を起こす場合があります。以下のコード例は、これがどのように発生するかを示したものです。

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://MultipleGroup">
  <xsd:complexType name="MultipleGroup">
    <xsd:sequence>
      <xsd:choice>
        <xsd:element name="child1" type="Child"/>
        <xsd:element name="child2" type="Child"/>
      </xsd:choice>
```

```
      <xsd:element name="separator" type="xsd:string"/>
      <xsd:choice>
        <xsd:element name="child1" type="Child"/>
        <xsd:element name="child2" type="Child"/>
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

「child1」および「child2」という名前のエレメントは、複数インスタンスが可能であることに注意してください。

これらの競合を解決するには、モデル・グループ用にサービス・データ・オブジェクト (SDO) パス・パターンを使用する必要があります。

## タスクの結果

以下のコード例で示すように、モデル・グループの処理に使用される SDO パス・パターンを使用した配列が取得されます。

```
set("child1/grandchild/name", "Bob");

set("child11/grandchild/name", "Joe");
```

**関連概念**

46 ページの『モデル・グループ・サポート (all、choice、sequence、および group 参照)』
SDO 仕様は、モデル・グループ (all、choice、sequence、および group 参照) を正しい位置に展開して、タイプやプロパティーを記述しないことを必要としています。

54 ページの『どのようにしてモデル・グループ配列を処理するか』
モデル・グループ配列は、モデル・グループが maxOccurs > 1 の値を持っているときに作成されます。

**関連タスク**

69 ページの『ネストされたビジネス・オブジェクトの単一インスタンス』
`setWithCreate` 機能を使用して、ネストされたビジネス・オブジェクトの単一インスタンスを作成します。

70 ページの『ネストされたビジネス・オブジェクトの複数インスタンスの作成』
`setWithCreate` 機能を使用して、ネストされたビジネス・オブジェクトの複数インスタンスを作成します。

72 ページの『ワイルドカードで定義されたネストされたビジネス・オブジェクトの使用』
タイプ `xsd:any` を親オブジェクト内で指定して、子オブジェクトを指定できますが、子オブジェクトが既に存在する場合に限られます。

# 第 11 章 XML 文書の妥当性検査

XML 文書およびビジネス・オブジェクトは、検証サービスを使用して検証することができます。

また、他のサービスでは一定の最低基準が必要で、満たさなければランタイム例外をスローします。これらの 1 つが BOXMLSerializer です。

BOXMLSerializer を使用すれば、XML 文書がサービス要求によって処理される前に検証することができます。BOXMLSerializer は XML 文書の構造を検証して、以下の種類のエラーがあるか判断します。

- 無効な XML 文書 (エレメント・タグが欠落しているものなど)。
- 整形式 XML 文書でないもの (クローズ・タグが欠落しているものなど)。
- 構文解析エラー (エンティティー宣言のエラーなど) を含む文書。

エラーが BOXMLSerializer によって検出されると、問題の詳細情報とともに例外がスローされます。

妥当性検査は、以下のサービスについての XML 文書のインポートまたはエクスポート、あるいはその両方に対して実行できます。

- HTTP
- JAXRPC Web サービス
- JAX-WS Web サービス
- JMS サービス
- MQ サービス

HTTP、JAXRPC、および JAX-WS サービスの場合、BOXMLSerializer は以下の方法で例外を生成します。

- インポート –
  1. SCA コンポーネントはサービスを呼び出します。
  2. サービスは宛先 URL を呼び出します。
  3. 宛先 URL は無効な XML 例外で応答します。
  4. サービスは失敗して、ランタイム例外およびメッセージが出されます。
- エクスポート –
  1. サービス・クライアントはサービス・エクスポートを呼び出します。
  2. サービス・クライアントは無効な XML を送信します。
  3. エクスポートのサービスが失敗して、例外およびメッセージが生成されます。

JMS および MQ メッセージング・サービスの場合、例外は以下の方法で生成されます。

- インポート –
  1. インポートは JMS または MQ サービスを呼び出します。
  2. サービスは応答を戻します。

© Copyright IBM Corp. 2005, 2008

3. サービスは無効な XML 例外を戻します。
4. インポートが失敗して、メッセージが生成されます。

- エクスポート –
  1. MQ または JMS クライアントはエクスポートを呼び出します。
  2. クライアントは無効な XML を送信します。
  3. エクスポートが失敗して、例外およびメッセージが生成されます。

XML 検証の例外によって生成されたメッセージは、ログを表示して確認できます。以下の例は、BOXMLSerializer によって検証され、XML コーディングが正しくないために生成されたメッセージです。

- JAXWS インポート

```
javax.xml.ws.WebServiceException: org.apache.axiom.om.OMException:
 javax.xml.stream.XMLStreamException: Element type "TestResponse" must be
 followed by either attribute specifications, ">" or "/>".

javax.xml.ws.WebServiceException: org.apache.axiom.soap.SOAPProcessingException:
  First Element must contain the local name, Envelope
```

- JAXRPC インポート

```
[9/11/08 15:16:27:417 CDT] 0000003e ExceptionUtil E
  CNTR0020E: EJB threw an unexpected (non-declared)
    exception during invocation of method
  "transactionNotSupportedActivitySessionNotSupported" on bean
    "BeanId(WSXMLValidationApp#WSXMLValidationEJB.jar#Module, null)".
  Exception data: WebServicesFault
 faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
 faultString: org.xml.sax.SAXParseException: Element type "TestResponse"
   must be followed by either
    attribute specifications, ">" or "/>". Message being parsed:
    <?xml version="1.0"?><TestResponse
    xmlns="http://WSXMLValidation"<firstName>Bob</firstName>
    <lastName>Smith</lastName></TestResponse>
 faultActor: null
 faultDetail:
[9/11/08 15:16:35:135 CDT] 0000003f ExceptionUtil E   CNTR0020E: EJB threw an
  unexpected (non-declared) exception during invocation of method
  "transactionNotSupportedActivitySessionNotSupported" on bean
    "BeanId(WSXMLValidationApp#WSXMLValidationEJB.jar#Module, null)".
 Exception data: WebServicesFault
 faultCode: {http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
 faultString: org.xml.sax.SAXException: WSWS3066E: Error: Expected 'envelope'
   but found  TestResponse
    Message being parsed: <?xml version="1.0"?><TestResponse
   xmlns="http://WSXMLValidation">
    <firstName>Bob</firstName><middleName>John</middleName>
   <lastName>Smith</lastName>
   </TestResponse>
 faultActor: null
 faultDetail:
```

- JAXRPC/JAXWS エクスポート

```
[9/11/08 15:35:13:401 CDT] 00000064 WebServicesSe E
  com.ibm.ws.webservices.engine.transport.http.WebServicesServlet
  getSoapAction WSWS3112E:
      Error: Generating WebServicesFault due to missing SOAPAction.
                    WebServicesFault
 faultCode: Client.NoSOAPAction
 faultString: WSWS3147E: Error: no SOAPAction header!
 faultActor: null
 faultDetail:
```

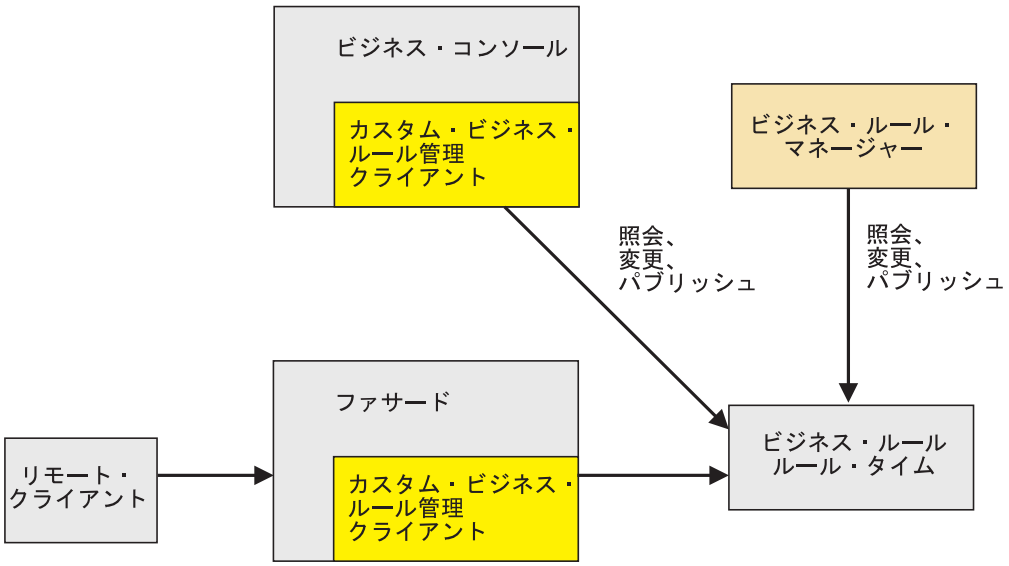
検証サービスについて詳しくは、『Reference』セクションの『Generated API and SPI』の資料にある `BOInstanceValidator` インターフェースの項目を参照してください。

# 第 12 章 ビジネス・ルールの管理

カスタム管理クライアントを作成したり、ビジネス・ルールの変更を自動化したりするための共通ビジネス・ルール管理クラスが用意されています。

ビジネス・ルール管理クラスを Web アプリケーションで使用し、ビジネス・プロセスやヒューマン・タスクなどを対象とした他の管理機能と組み合わせて、単一のクライアントからすべてのコンポーネントを管理することができます。 WebSphere Process Server に組み込まれたビジネス・ルール・マネージャー Web アプリケーションと併せて、あらゆるカスタム管理クライアントを使用できます。これらのクラスは、アプリケーション内のビジネス・ルールに対する変更を自動化するためにも使用できます。例えば、ビジネス・ルールを使用するビジネス・プロセスが一定のしきい値または制限を超えた場合、その結果としてビジネス・ルールを変更することができます。

ビジネス・ルール管理クラスは、WebSphere Process Server にインストールされたアプリケーションで使用する必要があります。これらのクラスはリモート・インターフェースを提供しませんが、クラスをファサードにラップして、リモート実行用に特定のプロトコルで公開することが可能です。




このプログラミング・ガイドは、2 つのメイン・セクションと付録で構成されています。最初のセクションでは、プログラミング・モデルと、さまざまなクラスの使用方法について説明します。クラス間の関係を示すために、クラス・ダイアグラムが記載されています。 2 番目のセクションでは、クラスを使用して、ビジネス・ルール・グループの検索、新規ルール宛先のスケジューリング、およびルール・セットまたはデシジョン・テーブルの変更などのアクションを実行する例を提供します。付録には、共通操作を単純化するために例の中で使用した追加クラスと、ワイルドカードを使用してビジネス・ルール・グループを検索する複雑な照会を作成する追加の例を記載します。

© Copyright IBM Corp. 2005, 2008

クラスに関する情報は、このプログラミング・ガイドのほか、WebSphere Process Server v6.1 および WebSphere Integration Developer v6.1 付属のテスト環境の両方にも Javadoc HTML 形式で含まれています。この Javadoc 資料は、${WebSphere Process Server Install Directory}¥web¥apidocs または ${WebSphere Integration Developer Install Directory}¥runtimes¥bi_v61¥web¥apidocs にあります。パッケージ com.ibm.wbiserver.brules.mgmt.* に、すべての情報が記載されています。

## プログラミング・モデル

WebSphere Business Integration のビジネス・ルールは、2 種類のオーサリング・ツールで作成されており、ルール・ランタイムによって実行されます。この 3 つはすべて、ビジネス・ルール成果物に同じモデルを共用します。

このモデルを共用することが、将来の保守を容易にするためだけでなく、エンド・ユーザーに一貫したプログラム・モデルを提供するためにも重要だと判断されました。このモデルの共有には、デスクトップ・ツールの要件と、ランタイムの実行およびオーサリングの要件との間で妥協が必要でした。いずれも、それぞれの環境に対応した明確な一連の要件があり、時にはこれらの要件が競合する場合があったからです。全体的なプログラミング・モデルの一部として以下に説明する成果物は、これらの異なる環境の要件をバランス良く満たすことを表します。

ビジネス・ルールの変更は、ルール・セットとデシジョン・テーブルのテンプレートで定義された項目、および操作選択テーブルのテンプレートで定義された項目 (有効開始日付とターゲット) のみに制限されます。ルール・セットおよびデシジョン・テーブルの新規作成は、既存のルール・セットまたはデシジョン・テーブルのコピーによってのみサポートされます。ビジネス・ルール・グループ・コンポーネント自体は、ランタイムでの動的オーサリングに適格ではありませんが、ユーザー定義のプロパティーおよび記述値については例外となります。コンポーネントに必要な変更 (例えば、新規操作の追加など) は、WebSphere Integration Developer で行った後に、サーバーに再デプロイまたは再インストールする必要があります。

### ビジネス・ルール・グループ

BusinessRuleGroup クラスは、ビジネス・ルール・グループ・コンポーネントを表します。 BusinessRuleGroup クラスは、ルール・セットとデシジョン・テーブルを含むルート・オブジェクトと考えることができます。

ルール・セットとデシジョン・テーブルは、関連付けられているビジネス・ルール・グループを通じてしかアクセスできません。このクラスには、ビジネス・ルール・グループに関する情報を取得するメソッドと、ルール・セットおよびデシジョン・テーブルにアクセスするためのメソッドが含まれます。これらのメソッドを使用すると、以下の情報を取得できます。

- ターゲット名前空間
- ビジネス・ルール・グループの名前
- 表示名
- 名前/表示名の同期
- 説明

- 表示される時間帯 (システム日付を UTC 形式とローカルのどちらで表示するのかを示します)
- ビジネス・ルール・グループに関連付けられているインターフェースに定義された操作
- ビジネス・ルール・グループで定義されたカスタム・プロパティー

ビジネス・ルール・グループに関連付けられている各種のルール・セットおよびデシジョン・テーブルには、ビジネス・ルール・グループの操作を通じてアクセスできます。

ビジネス・ルール・グループの情報を更新できるメソッドも用意されています。このメソッドを使用すると、以下の情報を更新できます。

- 説明
- 表示名
- 名前/表示名の同期
- ビジネス・ルール・グループで定義されたカスタム・プロパティー

ビジネス・ルール・グループの表示名は、明示的に設定することもできますし、setDisplayNameIsSynchronizedToName メソッドを使用して Name の値に設定することもできます。

その他の値は、ビジネス・ルール・グループのコンポーネント定義の一部であるため変更できません。これらの値を変更するには、再デプロイと再インストールが必要です。

ビジネス・ルール・グループ・クラスは、最新表示メソッドも備えています。このメソッドは、ビジネス・ルールが保管されている永続ストレージまたはリポジトリーを呼び出して、ビジネス・ルール・グループおよび関連するすべてのルール・セットとデシジョン・テーブルを持続情報とともに戻します。戻されるビジネス・ルール・グループは最新のコピーであり、直前のオブジェクトは廃止になります。

isShell メソッドを使用すると、ビジネス・ルール・グループ・インスタンスが現在のランタイムでサポートされないバージョンであるかどうかを確認できます。例えば、現在のビジネス・ルール管理クラスで Web クライアントを作成しており、将来、このクラスでサポートされない新機能をビジネス・ルール・グループに追加する場合は、ビジネス・ルール・グループを取得するときに、シェル・ビジネス・ルール・グループが作成されます。これにより、Web クライアントは、サポートされるビジネス・ルールの操作を続行することができ、属性と機能が制限されたビジネス・ルール・グループを依然として取得することができます。 isShell が true の場合、値を戻すメソッドは、getName、 getTargetNameSpace、 getProperties、 getPropertyValue、および getProperty のみです。それ以外のすべてのメソッドは、UnsupportedOperationException を生成します。 isShell メソッドを使用することに加えて、BusinessRuleGroup のタイプも検査すれば、これが BusinessRuleGroupShell のインスタンスであるかどうかを確認して、サポートされるバージョンであるかどうかを判断することができます。

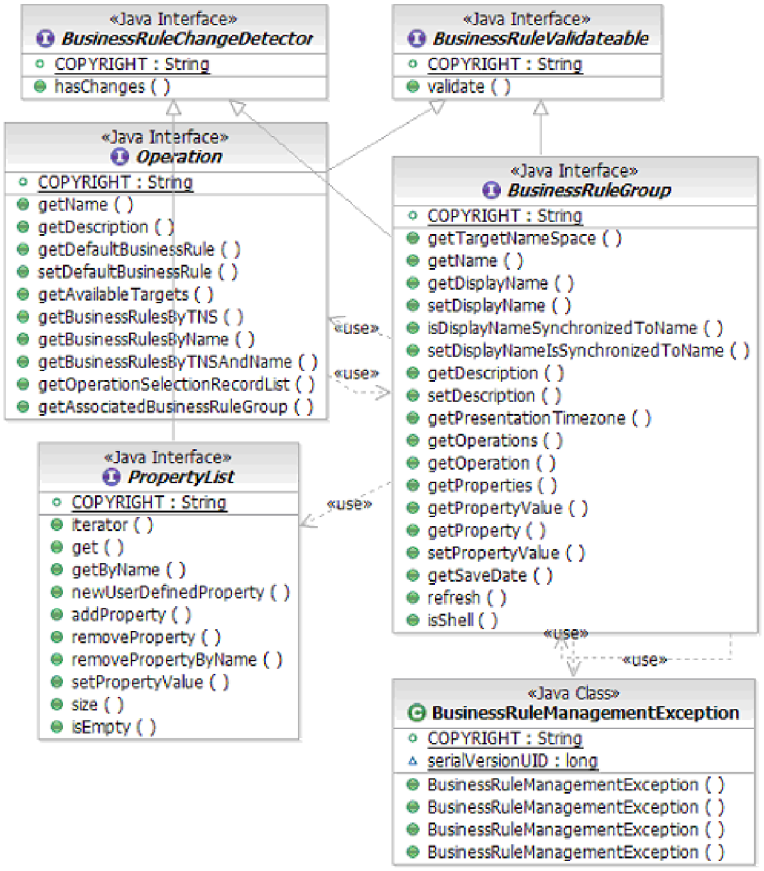



*図 10. BusinessRuleGroup および関連するクラスのクラス・ダイアグラム*

## ビジネス・ルール・グループのプロパティー

ビジネス・ルール・グループのプロパティーは、ビジネス・ルール・グループを管理するために使用されます。ビジネス・ルール・グループのプロパティー・セットを照会で使用すると、表示して変更する必要のあるビジネス・ルール・グループのサブセットのみを戻すことができます。

プロパティーのタイプはすべてストリングであり、名前と値のペアとして定義されます。各プロパティーは、ビジネス・ルール・グループで 1 回のみ定義できます。定義されている各プロパティーには、値も定義されている必要があります。プロパティーの値は、空ストリングにすることも、長さをゼロにすることもできますが、ヌルにすることはできません。プロパティーをヌルに設定することは、プロパティーを削除することと同じです。

ビジネス・ルール・グループのプロパティーは、実行時にルール・セットまたはデシジョン・テーブルでアクセスすることもできます。これにより、ビジネス・ルール・グループの複数のルール・セットまたはデシジョン・テーブル内で使用する単一値をビジネス・ルール・グループで設定することができます。含まれているルール・セットおよびデシジョン・テーブルで使用できるのは、ビジネス・ルール・グループで定義されているプロパティーに限られます。

プロパティーには、システム定義プロパティーとユーザー定義プロパティーの 2 種類があります。ビジネス・ルール・グループでは、システム定義プロパティーまたはユーザー定義プロパティーの数に制限はありません。システム・プロパティーは、特定のコンポーネント情報 (ルール・ロジックの定義時に使用するルール・モデルのバージョンなど) を保持するために使用されます。このシステム情報はプロパティーで公開され、これらのフィールド全体での照会が可能になります。システム・プロパティーは、接頭部 IBMSystem で始まり、ビジネス・ルール・グループおよびプロパティー・クラスを通じて、読み取り専用でアクセスされます。システム・プロパティーを追加、変更、または削除することはできません。システム・プロパティーの例を以下に示します。

| プロパティー名 | プロパティー値 |
| --- | --- |
| IBMSystemVersion | 6.2.0 |

**注:** ビジネス・ルール・グループの名前、名前空間、および表示名の値は、照会用のシステム・プロパティーとして扱われます。これらの値は、ビジネス・ルール・グループのプロパティーのリストに含められ、getProperties メソッドを使用して取得することができます。ただし、これらのプロパティーは、ビジネス・ルール・グループの独立した固有のエレメントで定義されており、ビジネス・ルール・グループ成果物の実際のプロパティー・エレメントとしては定義されていないため、WebSphere Integration Developer ではプロパティーとして表示されません。これらのプロパティーは、より多くの照会オプションを提供するためにのみに用意されています。

ユーザー定義プロパティーは、任意のユーザー固有の情報を保持するために使用します。ビジネス・ルール・グループの照会で使用することも可能です。ユーザー定義プロパティーは、読み取りと書き込みを行うことができます。

ビジネス・ルール・グループのプロパティーは、個々に取得することも、リスト (PropertyList オブジェクト) として取得することもできます。 PropertyList では、個々のプロパティーを取得するメソッドと、ユーザー定義プロパティーを追加および除去するメソッドを利用できます。

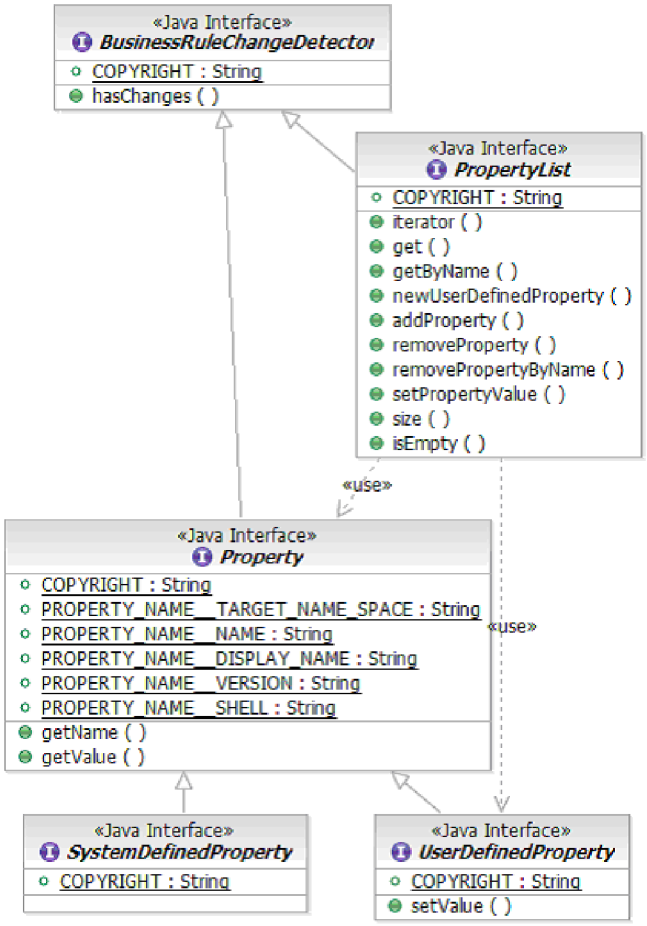



*図 11. Property および関連するクラスのクラス・ダイアグラム*

## 操作

操作は、変更する個々のルール・セットやデシジョン・テーブルに到達するための開始点となります。ビジネス・ルール・グループの操作は、そのビジネス・ルール・グループのコンポーネントに関連付けられた WSDL にリストされた操作と一致します。

各操作には、以下の異なるターゲットがあります。このターゲットのそれぞれが、ビジネス・ルール (ルール・セットまたはデシジョン・テーブル) です。

- デフォルト・ターゲット (オプション)
- 日付/時刻の範囲によってスケジュールされたターゲットのリスト (OperationSelectionRecord)
- 操作に使用可能なすべてのターゲットのリスト

各操作には、少なくとも 1 つのビジネス・ルール・ターゲットを指定する必要があります。このターゲットは、開始日と終了日でターゲットをアクティブにするスケジュールを指定した OperationSelectionRecord にできます。操作には単一のデフォルト・ターゲットを設定することもできます。この場合、一致するスケジュール済みビジネス・ルール・ターゲットが見つからない場合には、設定されたデフォルト・

ターゲットが実行中に使用されます。 Operation クラスには、デフォルト・ビジネス・ルール・ターゲットを取得および設定するためのメソッド、およびスケジュール済みビジネス・ルール・ターゲットのリスト (OperationSelectionRecordList) を取得するためのメソッドがあります。デフォルト・ビジネス・ルール・ターゲットとスケジュール済みビジネス・ルール・ターゲットとは別に、操作に使用可能なすべてのビジネス・ルール・ターゲットのリストがあります。このリストには、スケジュール済みおよびデフォルトのビジネス・ルール・ターゲットだけでなく、この操作にスケジュールされていない他のすべてのルール・セットまたはデシジョン・テーブルも含まれます。スケジュールされていないルール・セットまたはデシジョン・テーブルは、操作の情報を暗黙的に共用することから、使用可能なターゲット・リストによって操作に関連付けられています。すべてのビジネス・ルール・ターゲットは、それぞれに該当する操作の入力および出力メッセージをサポートする必要があります。各操作はインターフェース上で固有であるため、操作のルール・セットおよびデシジョン・テーブルは、別の操作のルール・セットおよびデシジョン・テーブルに対しても固有です。

使用可能なターゲット・リストに含まれるそれぞれのルール・セットおよびデシジョン・テーブルはいずれも、OperationSelectionRecord を作成することによって、アクティブになるようにスケジュールできます。使用可能なターゲット・リストから特定のルール・セットまたはデシジョン・テーブルを指定するのに加えて、開始日と終了日も指定する必要があります。開始日は、終了日よりも前でなければなりません。この 2 つの日付によって、現在、過去および将来の日付を範囲とする期間を指定できます。日付の範囲を OperationSelectionRecordList に追加して公開を行う時点で、その日付の範囲が他のどの OperationSelectionRecord ともオーバーラップしていないことが必要です。開始日と終了日の値の型は、java.util.Date です。指定する値はすべて、java.util.Date クラスに従った UTC 値として扱われます。作成した OperationSelectionRecord は、OperationSelectionRecordList に追加して他のビジネス・ルール・ターゲットと併せてスケジュールできます。異なる OperationSelectionRecord の間に、時間のギャップがある場合があります。実行中にギャップが検出された場合には、デフォルト・ターゲットが使用されます。デフォルト・ターゲットが指定されていない場合、例外がスローされます。ベスト・プラクティスとして、常にデフォルト・ビジネス・ルール・ターゲットを指定することをお勧めします。

スケジュール済みビジネス・ルール・ターゲットをスケジュール済みターゲットのリストから除去するには、OperationSelectionRecord を OperationSelectionRecordList から除去します。 OperationSelectionRecord を除去しても、使用可能なビジネス・ルール・ターゲットのリストからビジネス・ルール・ターゲットが除去されたり、同じビジネス・ルール・ターゲットがスケジュールされている他の OperationSelectionRecord が除去されたりすることはありません。

Operation クラスでは、OperationSelectionRecordList または使用可能なターゲット・リストを使用してルール・セットまたはデシジョン・テーブルを取得できるだけでなく、名前とターゲット名前空間のプロパティー値によってビジネス・ルール・ターゲットを取得することもできます。 Operation クラスのメソッドによって、該当する操作に使用可能なターゲット・リストに含まれるルール・セットおよびデシジョン・テーブルを照会できます。他の操作に使用可能なターゲット・リストに含まれるルール・セットおよびデシジョン・テーブルは、名前およびターゲット名前空間が一致していても、この結果セットには含まれません。特定のルール・セットお

よびデシジョン・テーブルの取得を単純化するために、
getBusinessRulesByName、getBusinessRulesByTNS、および
getBusinessRulesByTNSAndName メソッドが用意されています。

Operation クラスは、以下の操作をサポートするメソッドを提供します。

- 操作名の取得
- 操作についての説明の取得
- デフォルト・ビジネス・ルール・ターゲットの取得と設定
- スケジュール済みビジネス・ルール・ターゲット (OperationSelectionRecordList) の取得
- 使用可能なすべてのビジネス・ルール・ターゲットのリストの取得
- 名前またはターゲット名前空間を基準とした、使用可能なすべてのターゲット・リストからのルール・セットまたはデシジョン・テーブルの取得
- 操作が関連付けられたビジネス・ルール・グループの取得

OperationSelectionRecordList クラスは、以下の操作をサポートするメソッドを提供します。

- 索引値を基準とした、特定の OperationSelectionRecord の取得
- 索引値を基準とした、特定の OperationSelectionRecord の除去
- リストへの新規 OperationSelectionRecord の追加

OperationSelectionRecord クラスは、以下の操作をサポートするメソッドを提供します。

- 開始日の取得と設定
- 終了日の取得と設定
- ビジネス・ルール・ターゲットの取得と設定
- OperationSelectionRecord が関連付けられた操作の取得

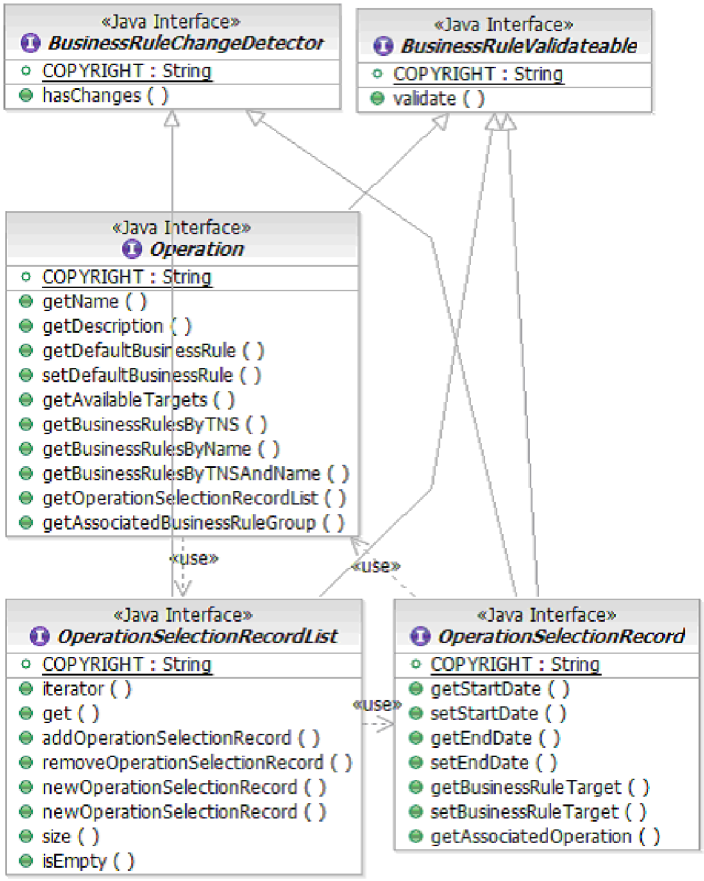



*図 12. Operation および関連クラスのクラス・ダイアグラム*

## ビジネス・ルール

RuleSet クラスと DecisionTable クラスは、ルール・セットとデシジョン・テーブルの両方で使用可能な情報を提供するメソッドを備えた汎用の BusinessRule クラスから派生しています。

ビジネス・ルール・グループの成果物と同様、ルール・セットとデシジョン・テーブルには、名前とターゲット名前空間があります。その他のルール・セットおよびデシジョン・テーブルと比較して、これらの値の組み合わせは固有でなければなりません。例えば、2 つのルール・セットは、同じターゲット名前空間の値を共用できますが、名前は別々でなければなりません。また、ルール・セットとデシジョン・テーブルは、同じ名前を持つことはできますが、ターゲット名前空間の値は異なります。

パラメーター値が異なる同等のルールを特定時刻にスケジュールする必要があるときは、ビジネス・ルールのコピーを既存のビジネス・ルールから作成できます (テンプレートからルールが作成されている場合)。新しいルールを最初から完全に作成することはできません。なぜなら、ビジネス・ルールの実装を提供するには、バッキング Java クラスが必要だからです。バッキング Java クラスは、デプロイ時にのみ作成されます。新しいルールを作成すると、そのルールは、元のルールに関連付

けられている操作の使用可能なターゲットのリストに追加されます。ただし、操作が関連付けられているビジネス・ルール・グループが公開されるまで、追加のルールは持続されません。

新しいビジネス・ルールは、元のルールとは異なるターゲット名前空間または名前を持つ必要があります。新しいビジネス・ルールの表示名は、元のルールと同じままにすることができます。なぜなら、名前と名前空間の組み合わせにより、ビジネス・ルールを識別するキー値が得られるからです。ビジネス・ルール内では、テンプレートを使用して定義した各種のパラメーター値を変更できます。ビジネス・ルールを特定時刻にスケジュールするには、OperationSelectionRecordList を使用するか、またはデフォルトの宛先として、ビジネス・ルールに関連付けられた Operation を使用します。

BusinessRule クラスは、以下の操作をサポートするメソッドを提供します。

- ターゲット名前空間の取得
- ルール・セットまたはデシジョン・テーブルの名前の取得
- ルール・セットまたはデシジョン・テーブルの表示名の取得および設定
- ビジネス・ルールのタイプ (ルール・セットまたはデシジョン・テーブル) の取得
- ビジネス・ルールの説明の取得および設定
- ビジネス・ルールが関連付けられている操作の取得
- 名前やターゲット名前空間が異なる、ビジネス・ルールのコピーの作成

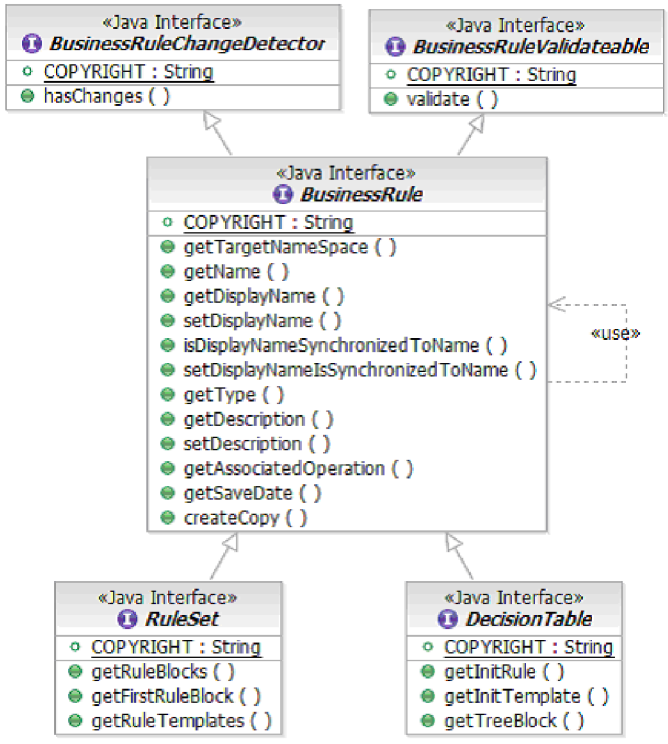



*図 13. BusinessRule および関連するクラスのクラス・ダイアグラム*

## ルール・セット

ルール・セットは、ビジネス・ルールの 1 つのタイプです。通常、ルール・セットは、異なる条件値に基づいて複数のルールを実行する必要がある場合に使用されます。ルール・セットはルール・ブロックおよびルール・テンプレートで構成されます。ルール・ブロック (RuleBlock) には、ルール・セットのロジックを構成するさまざまな if-then およびアクション・ルールが含まれます。

RuleSet クラスは、以下の操作をサポートするメソッドを提供します。

- ルール・セットのルール・ブロックのリストの取得
- ルール・セットに定義されたルール・テンプレートのリストの取得

現在、各ルール・セットに含めることのできるルール・ブロックは 1 つのみですが、ルール・セットには複数のルール・テンプレートを定義できます。ルール・ブロックには、ルール・セットが呼び出されると実行される一連のルールが含まれます。ルール・ブロックではルールの順序を変更できます。ルール・ブロックには少なくとも 1 つのルールが定義されていなければなりません。ルール (Rule) は、テンプレート・インスタンス・ルール (TemplateInstanceRule) として定義することも、ハードコーディングすることもできます。テンプレートで定義されている if-then またはアクション・ルールは、ルール・ブロックから除去できます。テンプレートで作成されたルールの新しいインスタンスは、ルール・ブロックに追加できます。

ルールがハードコーディングされていて、テンプレートでは定義されていない場合、そのルールは変更することも、ルール・ブロックから除去することもできません。これらのルールは、常にルール・セット・ロジックの一部となるように設計されており、ロジック内で変更されたり繰り返されたりすることは想定されていません。

新規ルールをテンプレートで作成するときには、そのルールに固有の名前が必要です。ルールを作成する前に、既存のルールのリストを取得して確認できます。

ハードコーディングされた if-then およびアクション・ルールについては、その名前と表示のみを取得できます。この表示は、クライアント・アプリケーションでルールに関する情報を表示するために使用できるストリングです。テンプレートで定義された if-then またはアクション・ルールについては、その名前と表示の他に、追加情報も取得できます。特定のパラメーター値を取得して変更することができます。ルール・セットに定義されたテンプレート (RuleSetRuleTemplate) を使用して、ルール・セット内にルールの別のインスタンスを作成し、パラメーター値を設定できます。例えば、特定の状況レベルの顧客が特定の量のディスカウントを受けると指定しているルールがあるとします。このロジックは、単一のルール・テンプレートで定義してから、該当する状況レベル (gold、silver、bronze など) およびディスカウント量 (15%、10%、5% など) ごとにパラメーター値を変更して繰り返すことができます。

テンプレートで定義されたルールのパラメーターは、ルールのインスタンスに固有です。テンプレートがルールについて定義するのは、標準の表示とパラメーターの数だけです。異なる顧客の状況に応じたディスカウントの例で説明したように、テンプレートで定義されたルールごとに異なる値を設定できます。

RuleBlock クラスは、以下の操作をサポートするメソッドを提供します。

- 索引によるルールの取得
- テンプレートで定義されたルールの追加
- テンプレートで定義されたルールの除去
- ルールの順序の変更 (1 つ上または下、もしくは特定の索引ロケーションへの移動)

`RuleSetRule` クラスは、以下の操作をサポートするメソッドを提供します。

- ルールの名前の取得
- ルールの表示名の取得
- ユーザー表示の取得
- ルール・ブロックの取得

`RuleSetRuleTemplate` クラスは、以下の操作をサポートするメソッドを提供します。

- このテンプレート定義からのルール・テンプレート・インスタンスの作成
- 親ルール・セットの取得

`TemplateInstanceRule` クラスは、以下の操作をサポートするメソッドを提供します。

- ルールのパラメーターの取得
- ルールを定義したテンプレート定義の取得

`Template` クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート ID の取得
- 名前の取得
- 表示名の取得と設定
- 説明の取得と設定
- このテンプレートのパラメーターの取得
- ユーザー表示の取得

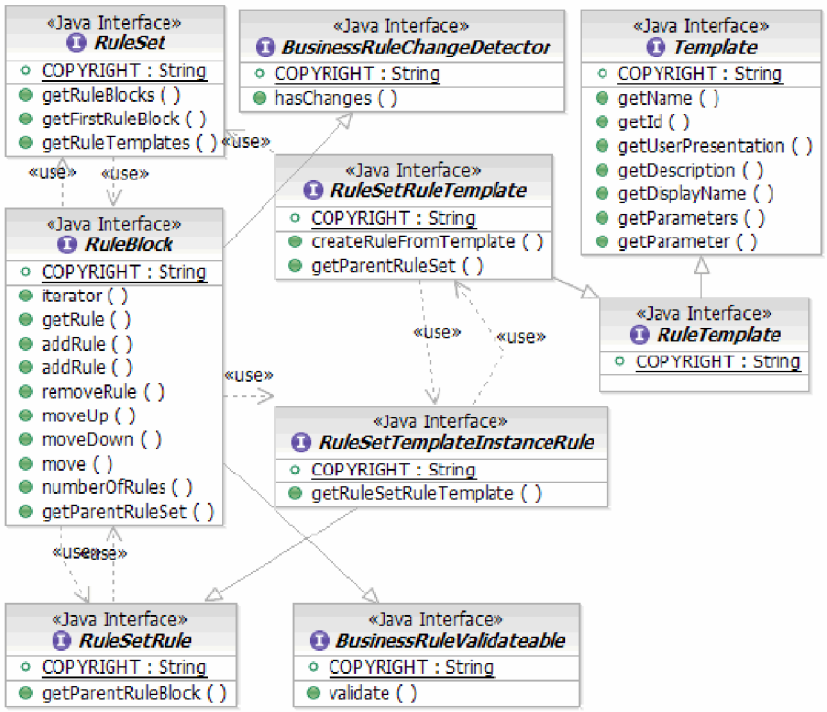


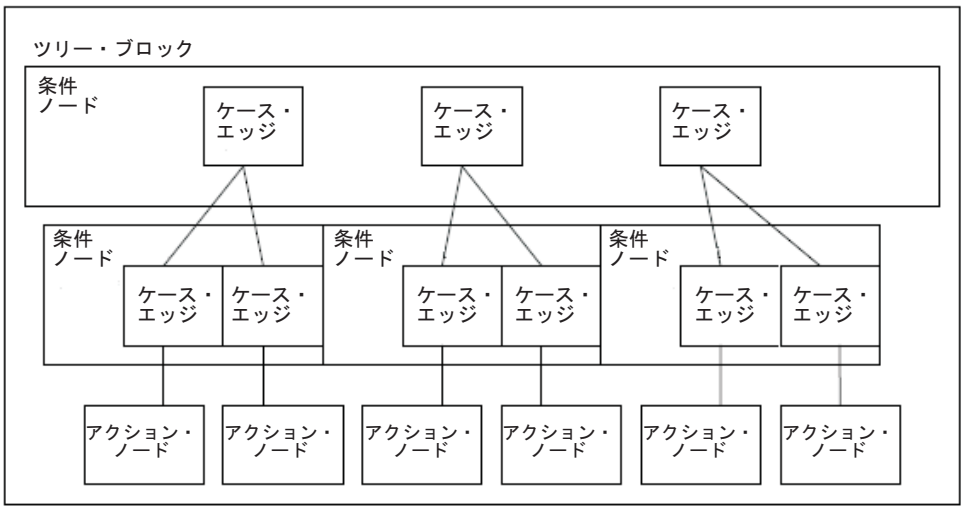

*図 14. BusinessRule および関連クラスのクラス・ダイアグラム*

## デシジョン・テーブル

デシジョン・テーブルは、管理および変更することのできるもう一つのタイプのビジネス・ルールです。通常、デシジョン・テーブルが使用されるのは、評価すべき条件の数に矛盾がなく、条件を満足した時点で発行する特定のアクション・セットが存在するときです。

デシジョン・テーブルは決定木と似ていますが、平衡型であるという特徴があります。どのブランチ・セットが真に解決されるかに関係なく、デシジョン・テーブルには常に同じ数の評価条件と実行アクションがあります。決定木は、別のブランチよりも評価条件を多く持つブランチを 1 つ持つことができます。

デシジョン・テーブルは、ノードのツリーとして作成され、`TreeBlock` によって定義されます。 `TreeBlock` は、各種の `TreeNodes` から構成されます。 `TreeNodes` は、条件ノードまたはアクション・ノードにすることができます。条件ノードは、評価ブランチです。条件がすべて真に評価された場合、ブランチの最後には、発行する該当のツリー・アクションを持つアクション・ノードがあります。条件ノードのレベルはいくつにでもできますが、アクション・ノードのレベルは 1 つにしかできません。




デシジョン・テーブルは、初期化ルール (init ルール) を持つこともできます。このルールは、テーブルの条件を検査する前に発行できます。

DecisionTable クラスは、以下の操作をサポートするメソッドを提供します。

- ツリー・ノード (条件ノードとアクション・ノード) のツリー・ブロックの取得
- init ルール・インスタンスの取得
- init ルール・テンプレートの取得 (定義されている場合)

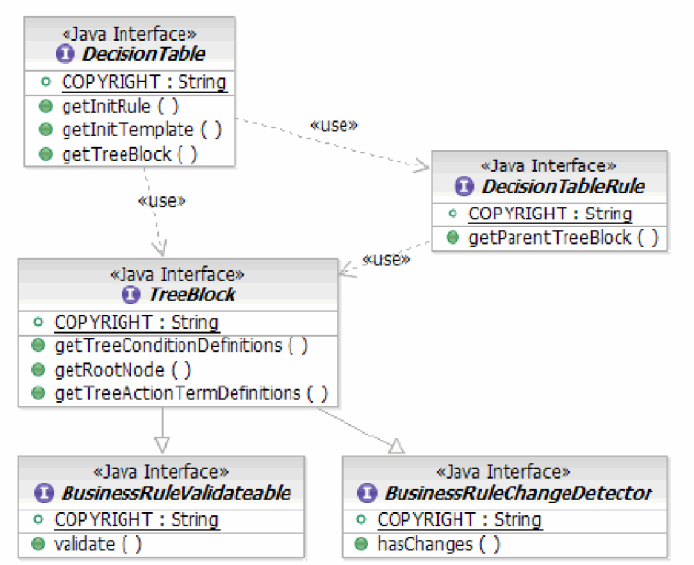
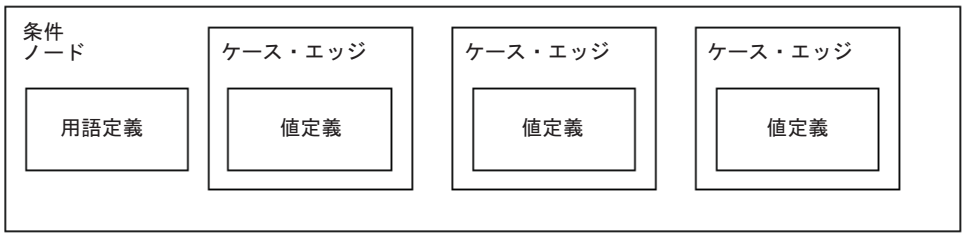



*図 15. DecisionTable および関連するクラスのクラス・ダイアグラム*

デシジョン・テーブルの TreeBlock には、さまざまな条件ノードとアクション・ノードが含まれています。各条件ノード (ConditionNode) には、条件定義 (TreeConditionTermDefinition) と、 1 対 n のケース・エッジ (CaseEdge) があります。条件定義には、条件式の左側のオペランドが含まれます。ケース・エッジには、値定義 (条件式の右側で使用する各種のオペランド) が含まれます。例えば、式

(status == “gold”) の場合、条件定義は「status」であり、「gold」はケース・エッジの値定義です。条件ノードのすべてのケース・エッジでは、条件定義が共有されており、値 (TreeConditionValueDefinition) のみが異なっています。例を続けます。条件ノードの別のケース・エッジは、値「silver」を持つことができます。この値は、式でも使用されます (status == “silver”)。この動作の唯一の例外は、otherwise が条件ノードに定義されている場合です。 otherwise では、値定義が存在しません。なぜなら、それは、条件ノード内のその他のすべてのケース・エッジが偽に評価された場合に使用されるからです。 otherwise は、ケース・エッジではありませんが、取得可能な TreeNode を持ちます。




条件定義の場合は、クライアント・アプリケーションでユーザー表示を取得して使用できます。通常、条件定義の表示は、左側のオペランド (この例では status) の表示のみであり、プレースホルダーは含まれません。ケース・エッジの場合は、テンプレートを使用して、値定義を定義できます (TreeConditionValueTemplate)。テンプレート値定義インスタンス (TemplateInstanceExpression) は、実行用に使用されるパラメーター値を実際に保持しています。このインスタンスは、変更することが可能です。テンプレートで定義されていない TreeConditionValueDefinition の値テンプレート定義の取得を試みると、ヌル値が戻されます。値の条件を定義するときにテンプレートを使用しなかった場合でも、オーサリング時に指定されていれば、クライアント・アプリケーションでユーザー表示を取得して使用することができます。

TreeBlock クラスは、以下の操作をサポートするメソッドを提供します。

- ツリーのルート・ノードの取得
- ツリー・ブロックの条件項の定義の取得
- ツリー・ブロックのアクション項の定義の取得

ツリーのルート・ノードのタイプは、TreeNode です。ここからは、デシジョン・テーブルのナビゲーションを使用する場合があります。 TreeNode クラスは、以下の操作をサポートするメソッドを提供します。

- ノードが otherwise 節かどうかの確認
- 現在のツリー・ノード (条件ノードまたはアクション・ノード) の親ノードの取得
- 現在のツリー・ノードを含むツリーのルート・ノードの取得

ConditionNode クラスは、以下の操作をサポートするメソッドを提供します。

- ケース・エッジの取得
- 条件定義の取得
- otherwise ケースの取得

- 条件ノード用のケース・エッジの値条件のテンプレートの取得
- テンプレートに基づく条件値のノードへの追加
- テンプレートに基づく条件値の除去

CaseEdge クラスは、以下の操作をサポートするメソッドを提供します。
- 値定義で使用できる値テンプレートのリストの取得
- 子ノード (条件ノードまたはアクション・ノード) の取得
- 値定義に関連付けられたテンプレート定義のインスタンスの取得
- テンプレートを取得せずに値定義を直接取得
- 特定のテンプレート・インスタンス定義を使用するための定義の値の設定

TreeConditionTermDefinition クラスは、以下の操作をサポートするメソッドを提供します。
- 条件ノード用に定義された値定義テンプレートの取得
- 条件項のユーザー表示の取得

TreeConditionDefinition クラスは、以下の操作をサポートするメソッドを提供します。
- 条件ノードの条件定義の取得
- 条件ノードの条件値の定義をすべてのケース・エッジから取得
- 方向 (行または列) の取得

TreeConditionValueDefinition クラスは、以下の操作をサポートするメソッドを提供します。
- 値用に定義された特定のテンプレート・インスタンス式の取得
- ユーザーの取得

Template クラスは、以下の操作をサポートするメソッドを提供します。
- テンプレートのシステム ID の取得
- テンプレートの名前の取得
- テンプレート用に定義されたパラメーターの取得
- テンプレートの表示の取得

TreeConditionValueTemplate クラスは、以下の操作をサポートするメソッドを提供します。
- テンプレート条件値インスタンスの新規作成

TemplateInstanceExpression クラスは、以下の操作をサポートするメソッドを提供します。
- テンプレート・インスタンスのパラメーターの取得
- インスタンスを定義するときに使用したテンプレート (デシジョン・テーブルのケース・エッジの場合は TreeConditionValueTemplate) の取得

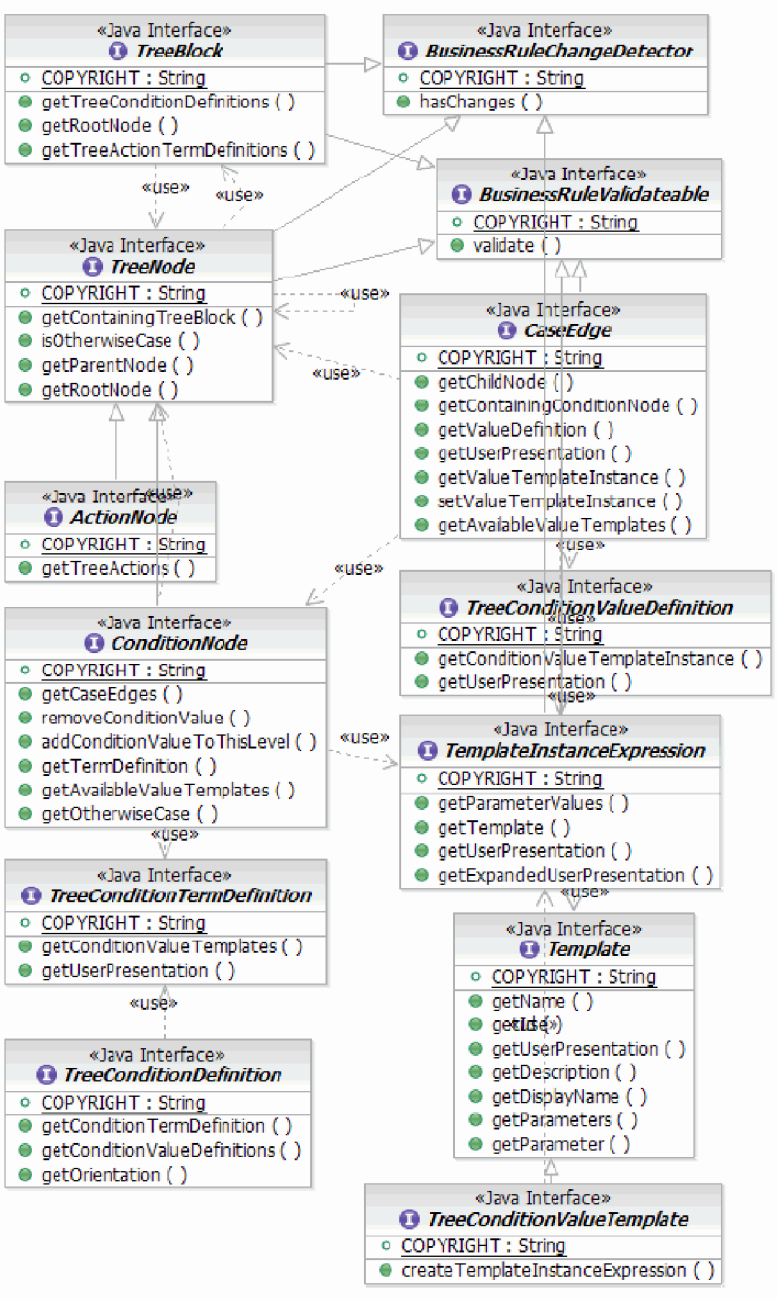



*図 16. TreeNode および関連するクラスのクラス・ダイアグラム*

新しいケース・エッジを条件ノードに追加する場合、そのケース・エッジでは、値を定義するためにテンプレートを使用する必要があります。例えば、「status」を検査するために「bronze」の新しいケース・エッジを追加する必要がある場合、パラメーター値に「bronze」を設定して新しい TemplateInstanceExpression を作成するには、該当するテンプレート (`TreeConditionValueTemplate`) を使用する必要があります。

新しいケース・エッジを追加する場合、そのケース・エッジには、子条件ノードが自動的に追加されます。この子条件ノードには、その同じレベルで条件ノード用に定義されたケース・エッジ定義に基づくケース・エッジが含まれます。テンプレートまたはハードコーディングした値をケース・エッジで使用する場合、これらは、

子条件ノードのケース・エッジでも使用されます。また、自動的に追加される子条件ノードでは、独自の子条件ノードが自動的に作成されます。すべてのレベルの条件ノードが再作成されるまで、子条件ノードにもその子条件ノードが繰り返し作成されます。

条件ノードに加えて、デシジョン・テーブル、より具体的に言えばツリー・ブロックにも、あるレベルのアクション・ノード (ActionNode) が含まれます。アクション・ノードは、リーフ・ノードであり、条件ノードのブランチの末端とケース・エッジに存在します。ケース・エッジの行で条件値がすべて真に解決された場合は、アクション・ノードに到達します。アクション・ノードには、少なくとも 1 つのアクション (TreeAction) が定義されます。このアクションには、条件定義と値定義が存在します。条件ノードと同様、条件定義 (TreeActionTermDefinition) は式の左側であり、値定義 (TemplateInstanceExpression) は式の右側です。例えば、状況を検査している各種の条件ノードにおいて、割引を定義するアクションが存在します。条件が (status == “gold”) の場合、アクションは、(discountValue = 0.90) になります。このアクションでは、「discountValue」が条件定義であり、「= 0.90」が値定義です。

ツリー・アクションの条件定義は、実際には、別のアクション・ノードのその他のツリー・アクションと共用されます。ケース・エッジのすべてのブランチがアクションに到達するので、同じ条件定義が使用されます。ただし、値定義は、ツリー・アクションおよびアクション・ノードごとに異なる場合があります。例えば、状況が「gold」の discountValue は「0.90」になる場合がありますが、状況が「silver」の「discountValue」は「0.95」になる場合があります。

アクション・ノードは、独立した条件定義と独立した値定義を含むツリー・アクションを複数持つことができます。例えば、レンタカーの割引を決定する場合は、discountValue を設定するほかに、特定のレベルの車を割り当てることもできます。「carSize」条件に「full size」を設定し (状況が「gold」の場合)、「discountValue」に「0.90」を設定する別のツリー・アクションを作成することができます。

ツリー・アクションの値定義は、テンプレート (TreeActionValueTemplate) から作成できます。テンプレート定義には、パラメーターを持つ式 (TemplateInstanceExpression) が含まれます。

パラメーターを変更することのほかに、値定義全体を変更できます。値定義全体を変更するには、ツリー・アクション用に定義された別のテンプレートで作成した新しい値定義インスタンスを使用します。

値定義は、テンプレートを使用せずに作成した場合は、変更することはできません。クライアント・アプリケーションの場合、オーサリング時にユーザー表示を指定した場合は、そのユーザー表示を表示内で使用できます。

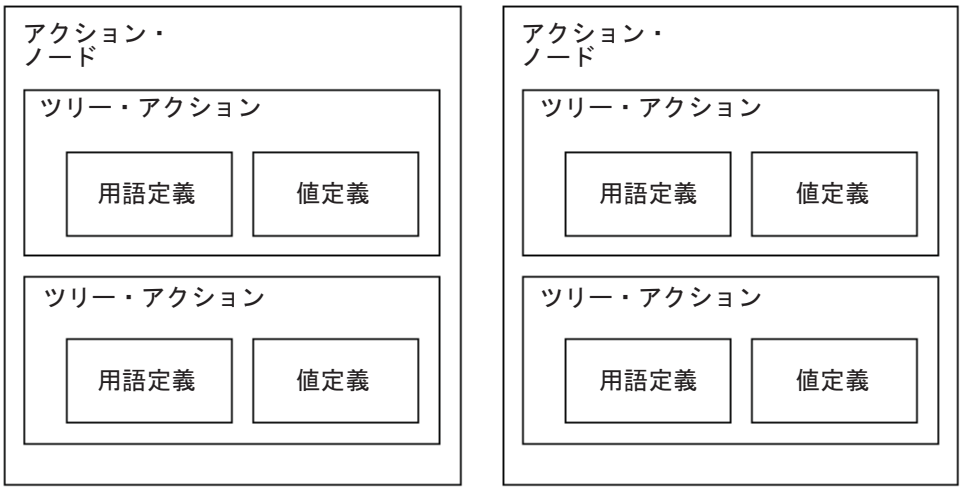
ツリー・アクションの条件定義の場合、ユーザー表示を指定すると、そのユーザー表示をクライアント・アプリケーションでも使用できます。

新しいケース・エッジを条件ノードに追加して、別の子条件ノードを作成すると、アクション・ノードも作成されます。そのレベル用に定義済みのケース・エッジの定義に基づいて作成されたケース・エッジおよび子条件ノードとは異なり、アクション・ノードでは、既存の設計が自動的に継承されません。アクション・ノードで

は、空のプレースホルダー TreeActions のみが作成されます。アクション・ノードの 1 つ以上の条件定義用に TemplateInstanceExpression を作成してアクション定義を完了するには、テンプレート (TreeActionValueTemplate) を使用する必要があります。 TemplateInstanceExpression を使用してツリー・アクションを設定するまで、ツリー・アクションでは、ユーザー表示値とテンプレート・インスタンス値にヌル値が指定されます。

新規の ActionNodes を生成する新しい条件を作成すると、直接の親の条件ノードの既存のアクションの右側にアクション・ノードが追加されます。例えば、「ruby」の状況がデシジョン・テーブルに追加され、特定の割引を持つ必要がある場合は、状況を検査する条件が「gold」、「silver」、および「bronze」の右側に追加されます。「ruby」の割引のアクション・ノードは、「gold」、「silver」、および「bronze」ケース・エッジに対応するアクション・ノードの右側に追加されます。

アクション・ノードの新しいツリー・アクションを設定すると、最下位のケース・エッジの右端のアクション・ノードに注目するアルゴリズムが、空のツリー・アクションを持つアクション・ノードを戻します。ユーザー表示値およびテンプレート・インスタンス値がヌル値であるかどうかについて、ツリー・アクションを検査することもできます。ツリー・アクションを取得すると、TreeActionValueTemplate の正しいインスタンスを使用して、そのツリー・アクションを設定することができます。




ActionNode クラスは、以下の操作をサポートするメソッドを提供します。

- 定義済みのツリー・アクションのリストの取得

TreeAction クラスは、以下の操作をサポートするメソッドを提供します。

- ツリー・アクション用に定義された使用可能な値テンプレートのリストの取得
- 条件定義の取得
- ツリー・アクション用に定義された値テンプレート・インスタンスの取得
- 値テンプレートを使用しなかった場合における値のユーザー表示の取得

- アクションが SCA サービス呼び出し (isValueNotApplicable メソッド) であるかどうかの確認
- 値テンプレート・インスタンスの新しいインスタンスでの置換

TreeActionTermDefinition クラスは、以下の操作をサポートするメソッドを提供します。

- 条件値の定義のユーザー表示の取得
- ツリー・アクションで使用可能な値テンプレートのリストの取得
- アクションが SCA サービス呼び出し (isTermNotApplicable メソッド) であるかどうかの確認

Template クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレートのシステム ID の取得
- テンプレートの名前の取得
- テンプレート用に定義されたパラメーターの取得
- テンプレートの表示の取得

TreeActionValueTemplate クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート定義からの新しい値テンプレート・インスタンスの作成

TemplateInstanceExpression クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート・インスタンスのパラメーターの取得
- インスタンスを定義するときに使用したテンプレート (デシジョン・テーブルのツリー・アクションの場合は TreeActionValueTemplate) の取得

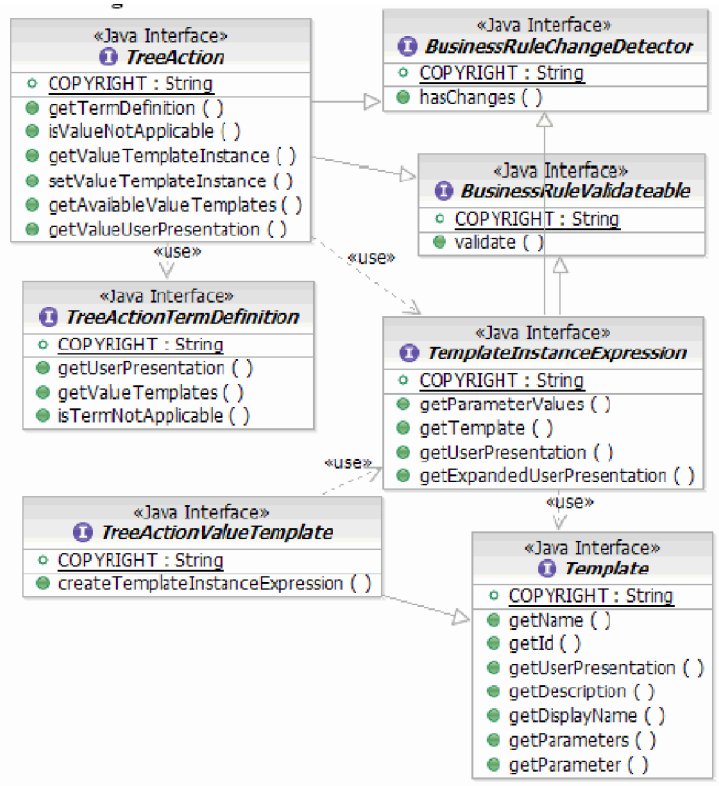



*図 17. TreeAction および関連するクラスのクラス・ダイアグラム*

デシジョン・テーブルの init ルールの定義は、ルール・セットのルールと同じ構造に従います。 init ルールは、テンプレート (`DecisionTableRuleTemplate`) で定義できます。

init ルールがオーサリング時に作成されなかった場合は、ルールをデプロイしたときにそれを追加することはできません。

`Rule` クラスは、以下の操作をサポートするメソッドを提供します。

- ルールの名前の取得
- ルールのユーザー表示の取得
- 各種のルール・パラメーターが設定されたルールのユーザー表示の取得

`DecisionTableRule` クラスは、以下の操作をサポートするメソッドを提供します。

- init ルールを含むツリー・ブロックの取得

`DecisionTableRuleTemplate` クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレートを含むデシジョン・テーブルの取得

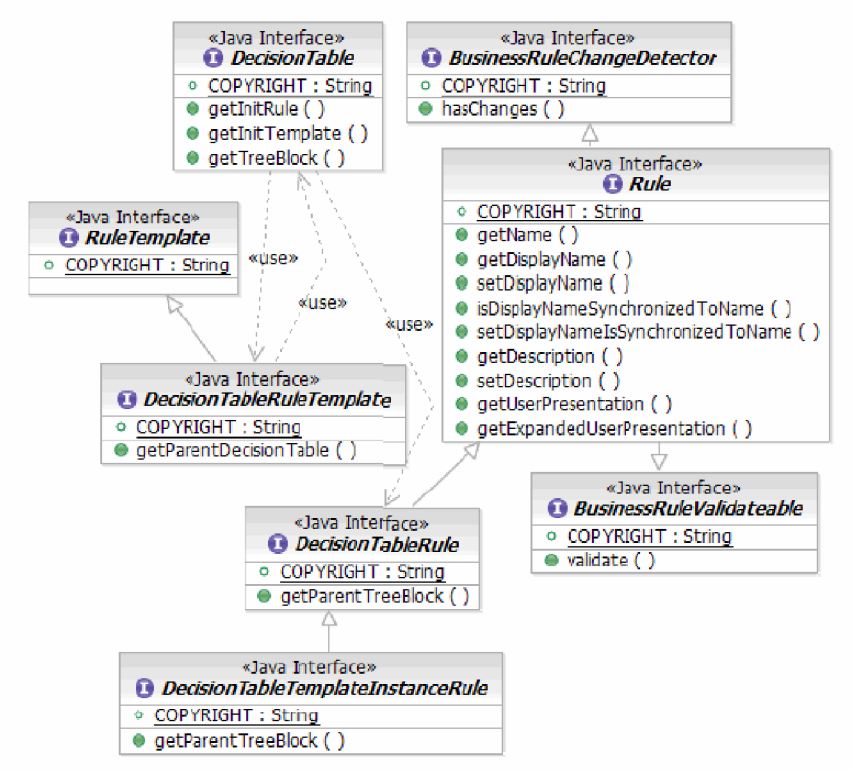



*図 18. DecisionTableRule および関連するクラスのクラス・ダイアグラム*

## テンプレートとパラメーター

ルール・セットおよびデシジョン・テーブルのテンプレートは、共通の定義に基づきます。テンプレートには、パラメーターとユーザー表示があります。テンプレートのパラメーター値は、ルールがデプロイされた後にそれを変更できるように定義されます。

ユーザー表示の値は、ルールやさまざまなパラメーターをユーザーに分かりやすく表示するために使用できるストリング値を定義します。ストリングであるユーザー表示には、さまざまなパラメーター値に置き換えられて正しく表示されるプレースホルダーが含まれています。プレースホルダーのフォーマットは {<parameter index>} です。例えば、初期ルールの表示ストリングが「Base discount is {0} %」の場合、プレースホルダー {0} をこのパラメーター値に置換できます。表示ストリングは、ルールまたはテンプレート定義に応じて変更することはできません。ただし、プレースホルダーの値は、テンプレートの定義に応じたクライアント・アプリケーションのパラメーター値で変更できます。それぞれのテンプレートには、すべてのパラメーター値が正しく配置されたストリングを返す便利メソッド (`getExpandedUserPresentation`) が組み込まれています。

すべてのパラメーター値には固有のデータ型がありますが、パラメーター値を取得および設定するときには、ストリング・オブジェクトが使用されます。パラメーター値は、値をユーザー表示に代入するときや、パラメーターに新しい値を設定するときにも、ストリングとして処理できます。実行時には、ルールを正しく発行するために、正しいデータ型にパラメーターが変換されます。検証時には、パラメータ

一値がそのデータ型と比較されて、正しいパラメーター値であることが確認されます。例えば、パラメーターが boolean 型で、値が「T」に設定されている場合、検証ではこの値が認識されないため、問題として返されます。

テンプレート定義では、パラメーター値を制約によって制限できます。制約は、範囲または列挙として定義できます。パラメーターに対する制約は、ルールが検証されるときに適用されます。値定義がテンプレートを使用して定義されていない場合、ユーザー表示のみが使用可能になります。値定義には、テンプレートとユーザー表示の両方を使用することはできません。テンプレートを使用した場合にはテンプレート定義の表示が使用可能な唯一の表示となります。

Template クラスは、以下の操作をサポートするメソッドを提供します。

- テンプレート ID の取得
- 名前の取得
- パラメーターの取得
- ユーザー表示の取得

Parameter クラスは、以下の操作をサポートするメソッドを提供します。

- パラメーター名の取得
- パラメーターのデータ型の取得
- パラメーターに対する制約の取得
- パラメーターを定義するテンプレートの取得
- パラメーター値の作成

ParameterValue クラスは、以下の操作をサポートするメソッドを提供します。

- パラメーター名の取得
- パラメーター値の取得
- パラメーター値の設定

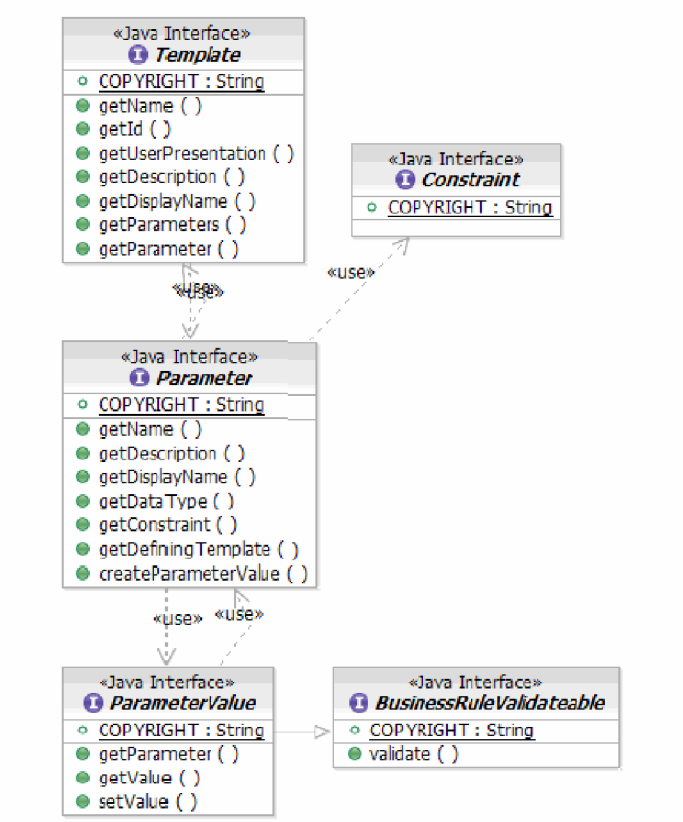



*図 19. Template、Parameter、および関連クラスのクラス・ダイアグラム*

## 検証

メイン・オブジェクトの多くには、成果物を公開する前に、その成果物の正確さと完全性を確認できる検証メソッドがあります。

API クラスを使用して変更するときに行われる検証は、serviceDeploy の実行中、または WebSphere Integration Developer で成果物を編集するときに行われる検証全体のなかのサブセットでしかありません。これは、ビジネス・ルール・グループにすでに設定された、実行時に編集可能な側面を制限するための制約によるものです。これらのクラスのユーザーは、必要に応じて常にビジネス・ルール・グループの選択テーブル、ルール・セットまたはデシジョン・テーブルを検証できます (ルール・グループのコンポーネント自体は、実行時に編集できません)。ビジネス・ルール・グループが公開されるときには、そのルール・グループの選択テーブル、ルール・セットおよびデシジョン・テーブルが検証されてから、リポジトリーに公開されます。

成果物が無効な場合、ValidationException がスローされ、検証問題のリストが提供されます。『例外処理』セクションに、さまざまな検証問題が文書化されています。

## 変更の追跡

すべてのオブジェクトで、そのオブジェクトや収容オブジェクトに変更が行われたかどうかを確認するために `hasChanges` メソッドを使用できます。

このメソッドは、変更を確認して、項目が変更されているビジネス・ルール・グループのみを公開するために使用できます。

## BusinessRuleManager

BusinessRuleManager クラスは、ビジネス・ルール・グループ、ルール・セット、およびデシジョン・テーブルを処理するためのメイン・クラスです。

BusinessRuleManager は、名前、ターゲット名前空間、またはカスタム・プロパティーによってビジネス・ルール・グループを取得できるメソッドを備えています。また、このクラスは、ビジネス・ルール・グループ、ルール・セット、またはデシジョン・テーブルに対して行われた変更を公開するためのメソッドも備えています。

BusinessRuleManager クラスは、以下の操作をサポートするメソッドを提供します。

- すべてのビジネス・ルール・グループの取得
- 特定のターゲット名前空間のビジネス・ルール・グループの取得
- 特定の名前のビジネス・ルール・グループの取得
- 特定の名前およびターゲット名前空間のビジネス・ルール・グループの取得
- 特定のプロパティーを 1 つ含むビジネス・ルール・グループの取得
- 特定のプロパティーを複数含むビジネス・ルール・グループの取得
- ビジネス・ルール・グループの公開

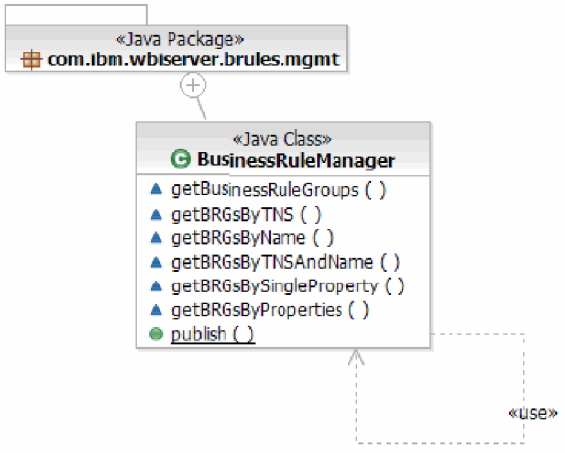



*図 20. BusinessRuleManager およびパッケージのクラス・ダイアグラム*

### ルール・グループ・コンポーネントの照会

ルール・グループ・コンポーネントは、ユーザー定義プロパティー (名前/値のペア) を持つことができます。このプロパティーを使用すると、クラスから戻されるビジネス・ルール・グループのリストを絞り込むことができます。照会で任意の組み合わせで使用できるフィールドを以下に示します。

- ビジネス・ルール・グループ・コンポーネントのターゲット名前空間

- ビジネス・ルール・グループ・コンポーネント名
- プロパティー名
- プロパティー値

各プロパティー名は、ビジネス・ルール・グループ・コンポーネントごとに 1 回だけ定義できます。

このクラスでサポートされる照会機能は、完全な SQL 言語の小さなサブセットです。ユーザーは、SQL ステートメントを指定するのではなく、単一プロパティーのパラメーターとして、またはノード形式の複数プロパティー照会用の情報を含むツリー構造のパラメーターとして値を指定します。論理演算子ノードとプロパティー照会ノードが存在しますが、これらのノードはすべて、`QueryNode` インターフェースを実装しています。論理演算子ノードでは、ブール演算子 (AND、OR、NOT) を指定します。これらは、`QueryNodeFactory` を通じて作成します。これらの論理演算子ノードを作成する一環として、追加の `QueryNode` クラスとともに左側と右側の演算子を指定する必要があります。これらのノードは、プロパティー照会ノードまたは別の論理演算子ノードにすることができます。プロパティー照会ノードが渡される場合、このノードには、プロパティーの名前、値、および演算子 (EQUAL (==)、NOT_EQUAL (!=)、LIKE、または NOTLIKE) が含まれます。 `QueryNode` 全体がクラスで構文解析され、永続ストレージ内の基盤となるデータに対して照会が実行されます。

LIKE 演算子または NOTLIKE 演算子を使用するときは、ワイルドカード検索がサポートされます。ワイルドカード検索では、「%」文字と「_」文字の両方がサポートされます。「%」文字を使用するのは、不明の文字、または検索時に考慮してはならない文字が不定の数だけ存在するときです。例えば、プロパティーの名前が「Department」であり、その値が「North」で始まるすべてのビジネス・ルール・グループを検索する場合は、値を「North%」として指定します。別の例として、値が「Region」で終了する「Department」がすべて必要であると想定します。この場合、値は「%Region」になります。「%」文字は、ストリングの中間で使用することもできます。例えば、ビジネス・ルール・グループのプロパティーの値が「NorthCentralRegion」、「NorthEastRegion」、および「NorthWestRegion」の場合は、値として「North%Region」を指定することができます。

不明の文字、または検索時に考慮すべきでない文字が 1 つだけ存在するときは、「_」文字を使用します。例えば、「Department」プロパティーの値が「Dept1North」、「Dept2North」、「Dept3North」、および「Dept4North」であるビジネス・ルール・グループをすべて検索する必要がある場合、値として「Dept_North」を指定すると、これらのプロパティーを持つ 4 つのビジネス・ルール・グループがすべて戻されます。「_」文字は、検索値で複数回使用できます。各インスタンスは、無視する 1 文字を示します。「_」文字は、値の先頭でも末尾でも使用することができます。例えば、値の中で 2 文字を無視する必要がある場合は、「_」を 2 つ使用します (例:「Dept__outh」)。

「%」と「_」をワイルドカードではなく、リテラル文字として扱うためには、「%」または「_」の前に「¥」エスケープ文字を指定する必要があります。例えば、プロパティー名が「%Discount」の場合、この名前を照会で使用するには、「¥%Discount」を指定する必要があります。「¥」文字をリテラル文字として使用する場合は、「¥」エスケープ文字をもう一つ使用する必要があります (例:

「Orders¥¥Customer」)。単一の「¥」文字があり、この後に「%」、「_」、または「¥」が続いていない場合は、IllegalArgumentException がスローされます。

ワイルドカード文字は、左側 (プロパティー値) でのみ使用できます。プロパティー名にワイルドカード文字を使用することはできません。

特定プロパティーの値の検索時、またはプロパティーに一致しない値の検索時、プロパティーが存在しないと、検索の考慮事項から成果物が無視されます。例えば、3 つのビジネス・ルール・グループ (A、B、C) があります。このうち、「Department」プロパティーを持つのは 2 つのみ (A と B) であり、このプロパティーの値はそれぞれ「Accounting」と「Shipping」になっています。この場合、「Department」プロパティーが「Accounting」でないビジネス・ルール・グループをすべて検索すると、「Department」プロパティーは定義されているが、その値が「Accounting」に等しくないビジネス・ルール・グループのみが戻されます (ビジネス・ルール・グループ B)。「Department」プロパティーを持たないビジネス・ルール・グループ (C) は戻されません。なぜなら、このプロパティーが定義されていないからです。

プロパティーを使用して検索する場合、*IBMSystemName* および *IBMSystemTargetNameSpace* という 2 つの特別なプロパティーを使用すると、成果物の名前および名前空間に基づいて検索を行うことができます。これらの値は、`getName` メソッドと `getTargetNameSpace` メソッドで取得することもできます。

クラスでは、照会用として以下のメソッドがサポートされます。

```
List getBRGsByTNS (string tNSName, Operator op, int skip, int threshold)
List getBRGByName(string Name, Operator op, int skip, int threshold)
List getBRGsByTNSAndName (string tNSName, Operator, tNSOp, string
  name, Operator nameOp, int skip, int threshold)
List getBRGsBySingleProperty (string propertyName, string propertyValue,
  Operator op, int skip, int threshold)
List getBRGsByProperties (QueryNode queryTree, int skip, int threshold)
```

「skip」および「threshold」パラメーターを使用すると、指定したしきい値までの部分的な結果リストを取り出すことができます。これらの両方のパラメーターにゼロの値を指定すると、完全な結果リストが戻されます。カーソルは、照会呼び出しからの結果セットに保持されません。 skip 値を使用する場合は、以前の結果セット内にあったビジネス・ルール・グループが以降の要求で戻されるように、結果セットに対して追加または削除を行うことができます。

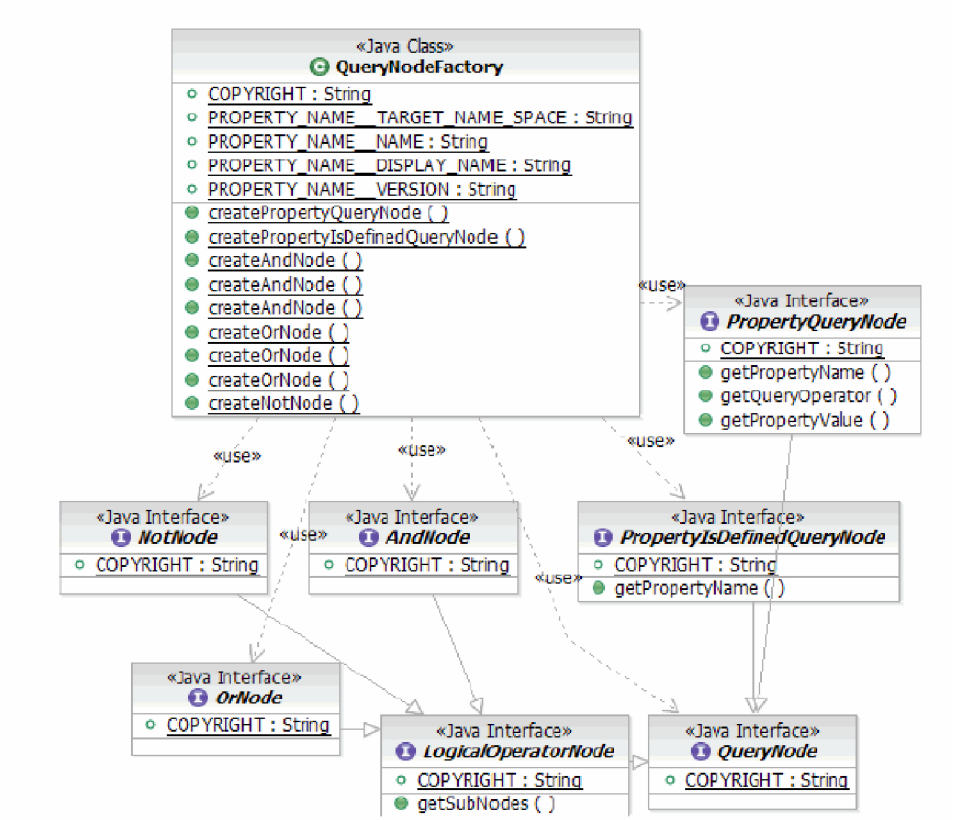



*図 21. QueryNodeFactory および関連するクラスのクラス・ダイアグラム*

ツリーのノードでは、検索式を指定するときにブール演算子、ワイルドカード (% およびエスケープ)、プロパティー/値のペアを使用できます。演算子は、値に対してのみ有効です。プロパティーの演算子は、常に等号 (==) です。

## 公開

ビジネス・ルールの変更の公開は、ビジネス・ルール・グループ・コンポーネント・レベルで行います。ユーザーは、1...n のビジネス・ルール・グループ・コンポーネントを公開できます。公開操作を実行する前に、ビジネス・ルール・グループとそれに含まれる別のオブジェクト (操作選択テーブル、ルール・セット、デシジョン・テーブルなど) に対して検証アクションが実行されます。各公開要求は、単一トランザクション内で行われます。検証時またはデータベース公開時に何らかの例外が検出された場合は、トランザクションがロールバックされ、どのビジネス・ルール・グループの変更もリポジトリーに公開されません。これにより、単一コンポーネント (例: 操作選択テーブルやルール・セット) 内で相互に依存関係を持つ変更、またはコンポーネント間の依存関係が 1 つのアトミック操作内で発生します。

公開時は検査が実行され、公開する項目が別のトランザクションによって変更されていないことが確認されます。競合の可能性を低減するために、公開メソッドでは、すべての成果物をその変更の有無にかかわらず公開すること、またはビジネス・ルール・グループ内で変更された成果物のみを公開することを選択できます。デフォルトの動作では、すべての成果物が公開されます。すべての成果物を公開するようにオプションを設定したが、別のトランザクションが成果物を変更した場合は、`ChangeConflictException` がスローされます。変更された成果物のみを公開す

ることを指定すると、競合の可能性が低減されます。変更された成果物のみを公開するようにすると、ビジネス・ルール・グループ内の 2 つの異なる成果物 (例: 2 つのルール・セット) の変更を 2 人のユーザーがリポジトリーにプッシュする場合があります。これにより、ビジネス・ルール・グループ内に互換性のない変更が生じる場合があります。こうした状況が発生する恐れがあるので、このオプションを使用するときは、注意を払う必要があります。

## 例外処理

例外は、成果物で検証が呼び出されるとき、または成果物が公開されるときに発生する可能性があります。検証エラーが発生すると、`ValidationException` が問題のリストとともにスローされます。成果物の公開中に、別のトランザクションが同じ成果物を公開することによって問題が発生すると、`ChangeConflictException` がスローされます。 `ChangeConflictException` 例外は、成果物の変更中に別のトランザクションが検出された時点で常にスローされます。

`SystemPropertyNotChangeableException` は、システム・プロパティー名と重複するプロパティーの変更が試行された場合にスローされます。システム・プロパティーは変更できません。

`ChangesNotAllowedException` は、公開中の成果物で設定操作が試行された場合にスローされます。

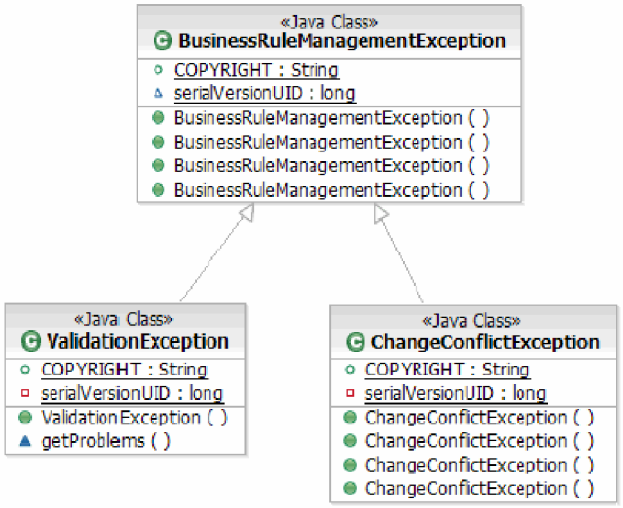



*図 22. BusinessRuleManagementException および関連クラスのクラス・ダイアグラム*

## ビジネス・ルール・グループでの問題

ビジネス・ルール・グループの検証時、またはビジネス・ルール・グループを公開しようとするときに、ビジネス・ルール・グループの一部が有効でないと、問題が発生する可能性があります。

*表 4. ビジネス・ルール・グループでの問題*

| 例外 | 説明 |
|---|---|
| ProblemBusRuleNotInAvailTargetList | この問題は、ルールが操作選択テーブルのデフォルト・ビジネス・ルールとして指定されているにもかかわらず、そのルール成果物がその操作の使用可能なターゲットのリストに含まれていないと発生します。この問題を回避するには、操作に使用可能なターゲットのリストから有効なビジネス・ルールを指定してください。 |
| ProblemDuplicatePropertyName | この問題は、ビジネス・ルール・グループのシステム・プロパティーまたはユーザー定義プロパティーと重複するプロパティーを作成しようとすると発生します。この問題を回避するには、固有のプロパティー名を使用してください。 |
| ProblemOperationContainsNoTargets | この問題は、操作にデフォルト・ルール宛先またはスケジュールされたルール宛先が設定されていないと発生します。この問題を回避するには、操作に少なくとも 1 つのルール宛先をデフォルトとして設定するか、またはスケジュールされた時間のルール宛先を設定してください。 |
| ProblemOverlappingRanges | この問題は、操作選択レコードの開始日または終了日が、別の操作選択レコードの開始日と終了日の範囲にオーバーラップしていると発生します。日付範囲がオーバーラップしていると、呼び出すべき正しいルール宛先をビジネス・ルール・ランタイムが見つけられません。この問題を回避するには、操作の他の操作選択レコードの開始日または終了日をチェックして、オーバーラップしていないことを確認してください。 |
| ProblemStartDateAfterEndDate | この問題は、操作選択レコードの開始日が、その選択レコードの終了日よりも後に指定されていると発生します。この問題は、開始日または終了日のないデフォルト・レコードを除くすべての操作選択レコードで発生する可能性があります。この問題を回避するには、操作選択レコードの開始日を終了日より前に指定してください。 |
| ProblemTargetBusRuleNotSet | この問題は、操作選択レコードに指定されたルールが、使用可能なターゲット・ルールのリストに含まれていないと発生します。この問題を回避するには、使用可能なターゲット・リストに含まれているルールを指定してください。 |
| ProblemTNSAndNameAlreadyInUse | この問題は、新規に作成するビジネス・ルールのターゲット名前空間と名前が、ルール・セットまたはデシジョン・テーブルですでに使用されていると発生します。このチェックは、現行のビジネス・ルール・グループに関連付けられたすべてのルール・セットとデシジョン・テーブルだけでなく、リポジトリーに保管されたルール成果物でも行われます。この問題を回避するには、異なるターゲット名前空間または名前を使用してください。 |
| ProblemWrongOperationForOpSelectionRecord | この問題は、新しい操作選択レコードを操作選択レコード・リストに追加するときに、新規レコードの操作がリスト内のレコードの操作と一致しないと発生します。この問題を回避するには、正しい操作選択レコード・リスト・オブジェクトで `newOperationSelectionRecord` メソッドを使用して新しい操作を作成してください。 |

## ルール・セットおよびデシジョン・テーブルでの問題

*表 5. ルール・セットおよびデシジョン・テーブルでの問題*

| 例外 | 説明 |
|---|---|
| ProblemInvalidBooleanValue | この問題は、ルール・セットに含まれるルール・テンプレートのパラメーター、またはデシジョン・テーブルに含まれるアクション値または条件値が、ブール型のパラメーターとして「true」または「false」とは異なる値を受け取ると発生します。例えば、「T」や「F」は不正なパラメーター値です。この問題を回避するには、ブール型のパラメーターを処理するときには「true」または「false」の値を使用してください。 |
| ProblemParmNotDefinedInTemplate | この問題は、値を指定するテンプレート・パラメーターが、そのテンプレートの有効なパラメーターのリストに定義されていないと発生します。パラメーターは、テンプレートで設定する前に確認してください。この問題は、RuleTemplate、 TreeActionValueTemplate、または TreeConditionValueTemplate テンプレートで発生する可能性があります。 |
| ProblemParmValueListContainsUnexpectedValue | この問題は、テンプレートに渡されたパラメーターは有効であるものの、そのテンプレートのパラメーターの数としては多過ぎる場合に発生します。パラメーターの数を削減する必要があります。この問題は、RuleTemplate、 TreeActionValueTemplate、または TreeConditionValueTemplate テンプレートで発生する可能性があります。 |
| ProblemRuleBlockContainsNoRules | この問題は、ルール・セットに含まれるルール・ブロック内のすべてのルールが除去されているときに、ルール・セットの検証または公開が試行されると発生します。ルール・セットに含まれるルール・ブロックには、少なくとも 1 つのルールがなければなりません。 |
| ProblemTemplateNotAssociatedWithRuleSet | この問題は、ルール・セットに追加しようとしているルールが、そのルール・セットで定義されていないテンプレートで作成されている場合に発生します。この問題を回避するには、新しいルールを作成するときに、ルール・セットで定義されたテンプレートを使用してください。 |
| ProblemRuleNameAlreadyInUse | この問題は、ルール・セットのルール・ブロックに追加しようとしているルールの名前が、そのルール・ブロック内にある既存のルールの名前と同じ場合に発生します。この問題を回避するには、新しいルールを追加する前に、そのルールの名前を確認してください。 |
| ProblemTemplateParameterNotSpecified | この問題は、ルール・セットのルールまたはデシジョン・テーブルに含まれるアクション値あるいは条件値に対してテンプレートを更新するときに、パラメーターが含まれていないと発生します。この問題を回避するには、テンプレートのすべてのパラメーターを指定してください。 |
| ProblemTypeConversionError | この問題は、テンプレートのパラメーターを適切な型に変換できない場合に発生します。すべてのパラメーターはストリング・オブジェクトとして処理されてから、パラメーターの型 (boolean、 byte、 short、 int、 long、 float、および double) に変換されます。パラメーター値のストリングを、そのパラメーターに指定された型に変換できないと、このエラーが発生します。この問題を回避するには、パラメーターの型 (boolean、 byte、 short、 int、 long、 float、および double) に変換可能なストリングを指定してください。 |

*表 5. ルール・セットおよびデシジョン・テーブルでの問題 (続き)*

| 例外 | 説明 |
|---|---|
| ProblemValueViolatesParmConstraints | この問題は、パラメーターが、そのパラメーターに対してテンプレートで定義されている列挙または値の範囲内にない場合に発生します。この問題は、ルール・セットのルール・テンプレート、あるいはデシジョン・テーブルのアクション値または条件値のテンプレートで列挙または範囲によって制限されているパラメーターで発生する可能性があります。この問題を回避するには、列挙に含まれる値を使用してください。 |
| ProblemInvalidActionValueTemplate | この問題は、テンプレート・インスタンスをツリー・アクションの値定義に設定しようとしたけれども、対応するテンプレートがそのツリー・アクションには使用できない場合に発生します。この問題を回避するには、正しいテンプレートを使用してツリー・アクションの値定義を作成してください。 |
| ProblemInvalidConditionValueTemplate | この問題は、テンプレート・インスタンスをケース・エッジの条件定義に設定しようとしたけれども、対応するテンプレートがそのケース・エッジには使用できない場合に発生します。この問題を回避するには、正しいテンプレートを使用してケース・エッジの条件定義を作成してください。 |
| ProblemTreeActionIsNull | この問題は、新しい条件値を作成するときに、アクションがテンプレート・インスタンスで設定されていない場合に発生します。ActionNode のテンプレートを使用して新しいテンプレート・インスタンスを作成し、そのインスタンスを `TreeActions` のリストに設定してください。 |

## 許可

クラスでは、どのレベルの許可もサポートされません。独自の形式の許可を追加するのは、クラスを使用するクライアント・アプリケーションに任されています。

## 例

さまざまなクラスを使用してビジネス・ルール・グループを取得する方法を示す例や、ルール・セットとデシジョン・テーブルを変更する方法を示す例が数多く提供されています。これらの例は、プロジェクト交換ファイル (ZIP) で提供されています。このファイルは WebSphere Integration Developer にインポートして参照したり、再使用したりできます。

プロジェクト交換には、以下の多数のプロジェクトがあります。

- **BRMgmtExamples** – さまざまな例で使用するビジネス・ルール成果物が含まれるモジュール・プロジェクト。
- **BRMgmt** – com.ibm.websphere.sample.brules.mgmt パッケージに置かれている例が含まれる Java プロジェクト。
- **BRMgmtDriverWeb** – サンプルを実行するためのインターフェースが含まれる Web プロジェクト。

これらの例は EAR ファイル (BRMgmtExamples.ear) としても提供されています。この EAR ファイルは、WebSphere Process Server にインストールしてから実行し

ます。例には Web インターフェースが提供されています。これらの例では、クラスを使用して成果物を取得し、変更を加え、変更を公開する方法を示すことに重点を置いているため、この Web インターフェースは意図的に単純なものになっています。機能性に優れた Web インターフェースとなるようには意図されていません。ただし、これらのクラスは、堅固な Web サービスを作成するために簡単に使用でき、ビジネス・ルールの変更を主な目的とする他の Java アプリケーションで使用できます。

サンプル・アプリケーションは、WebSphere Process Server v6.1 にインストールできます。索引ページにアクセスするには、以下を指定します。

http://<hostname>:<port>/BRMgmtDriverWeb/

例えば、http://localhost:9080/BRMgmtDriverWeb/ となります。

例を実行すると、ルール成果物が変更されることになります。すべての例を実行した場合、再びすべての例で同じ結果を得るためには、アプリケーションを再インストールする必要があります。

それぞれの例について、完全なサンプル・コードと Web ブラウザーに表示される結果を記載して詳しく説明します。

共通操作を実行し、サンプル Web アプリケーション内に情報を表示できるように、多数の追加クラスが作成されています。 `Formatter` クラスおよび `RuleArtifactUtility` クラスについて詳しくは、付録を参照してください。

これらの例を十分に理解するには、WebSphere Integration Developer 内のさまざまな成果物について調査することが大いに役立ちます。

## 例 1: すべてのビジネス・ルール・グループを取得してプリントする

この例では、すべてのビジネス・ルール・グループを取得して、各ビジネス・ルール・グループの属性、プロパティー、および操作をプリントします。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;
```

ビジネス・ルール管理クラスとしては、必ず com.ibm.wbiserver.brules.mgmt パッケージに含まれるクラスを使用してください。com.ibm.wbiserver.brules パッケージやその他のパッケージに含まれるクラスは使用できません。それらの他のパッケージは、IBM 内部クラスで使用されます。

```
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import
com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;

public class Example1 {
static Formatter out = new Formatter();
static public String executeExample1()
```

```
{
try
{
out.clear();
```

BusinessRuleManager クラスは、ビジネス・ルール・グループを取得したり、ビジネス・ルール・グループの変更を公開したりする際に使用するメイン・クラスです。このクラスの用途には、ルール・セットやデシジョン・テーブルなどのルール成果物の操作および変更も含まれます。 BusinessRuleManager クラスには、名前と名前空間、およびプロパティーを渡すことで特定のビジネス・ルール・グループを容易に取得できるメソッドが多数用意されています。

```
// すべてのビジネス・ルール・グループを取得します。
List<BusinessRuleGroup> brgList = BusinessRuleManager
  .getBusinessRuleGroups(0, 0);

Iterator<BusinessRuleGroup> iterator = brgList.iterator();

BusinessRuleGroup brg = null;
// ビジネス・ルール・グループのリストを繰り返し処理します。
while (iterator.hasNext())
{
  brg = iterator.next();
  // 各ビジネス・ルール・グループの属性を出力します。
  out.println1nBold("Business Rule Group");
```

ビジネス・ルール・グループの基本属性を取得して表示することができます。

```
out.println("Name: " + brg.getName());
out.println("Namespace: " +
brg.getTargetNameSpace());
out.println("Display Name: " +
brg.getDisplayName());
out.println("Description: " + brg.getDescription());
out.println("Presentation Time zone: "
     + brg.getPresentationTimezone());
out.println("Save Date: " + brg.getSaveDate());
```

ビジネス・ルール・グループのプロパティーも取得して変更することができます。

```
PropertyList propList = brg.getProperties();

Iterator<Property> propIterator =
propList.iterator();
Property prop = null;
// プロパティーの名前と値を出力します。
while (propIterator.hasNext())
{
prop = propIterator.next();
out.println("Property Name: " +
prop.getName());
out.println("Property Value: " +
prop.getValue());
}
```

ビジネス・ルール・グループに対する操作も使用できます。これらの操作によって、ルール・セットやデシジョン・テーブルなどのビジネス・ルール成果物を取得できます。

```
List<Operation> opList = brg.getOperations();

Iteration<Operation> opIterator = opList.iterator();
Operation op = null;
// ビジネス・ルール・グループの操作を出力します。
```

```
while (opIterator.hasNext())
{
op = opIterator.next();
out.println("Operation: " + op.getName());
}
out.println("");}
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
}
return out.toString();
}
}
```

例 1 の Web ブラウザー出力

**例 1 の実行**

```
Business Rule Group
Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ApprovalValues
Property Name: IBMSystemDisplayName
Property Value: ApprovalValues
Operation: getApprover

Business Rule Group
Name: ConfigurationValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ConfigurationValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: General
Property Name: RuleType
Property Value: messages
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ConfigurationValues
Property Name: IBMSystemDisplayName
Property Value: ConfigurationValues
Operation: getMessages

Business Rule Group
Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
```

```
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules
Operation: calculateOrderDiscount
Operation: calculateShippingDiscount
```

## 例 2: ビジネス・ルール・グループ、ルール・セット、およびデシジョン・テーブルを取得してプリントする

この例では、例 1 の関数と同じ処理に加えて、各操作の選択テーブル、デフォルト・ビジネス・ルール宛先 (ルール・セットまたはデシジョン・テーブルのいずれか)、操作にスケジュールされたその他のビジネス・ルールをプリントします。ルール・セットとデシジョン・テーブルの両方をプリントします。

この例の大部分は同じですが、完全を期して記載しています。

```
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
public class Example2
{
 status Formatter out = new Formatter();
 static public String executeExample2()
 {
  try
  {
    out.clear();
```

この例では、名前を基準に特定のビジネス・ルール・グループを取得します。

```
  // すべてのビジネス・ルール・グループを取得します。
  List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByName("DiscountRules",
    QueryOperator.EQUAL, 0, 0);

  Iterator<BusinessRuleGroup> iterator = brgList.iterator();

  BusinessRuleGroup brg = null;
  // ビジネス・ルール・グループのリストを繰り返し処理します。
  while (iterator.hasNext())
   {
   brg = iterator.next();
   // 各ビジネス・ルール・グループの属性を出力します。
```

```
out.printlnBold("Business Rule Group");
out.println("Name: " + brg.getName());
out.println("Namespace: " +
 brg.getTargetNameSpace());
out.println("Display Name: " +
 brg.getDisplayName());
out.println("Description: " + brg.getDescription());
out.println("Presentation Time zone: "
 + brg.getPresentationTimezone());
out.println("Save Date: " + brg.getSaveDate());

PropertyList propList = brg.getProperties();

Iterator<Property> propIterator =
propList.iterator();
Property prop = null;
// プロパティーの名前と値を出力します。
while (propIterator.hasNext())
{
  prop = propIterator.next();
  out.println("Property Name: " +
  prop.getName());
  out.println("Property Value: " +
  prop.getValue());
}
```

各操作の選択テーブルには、さまざまなルール成果物と、そのルール成果物がアクティブになるスケジュールがリストされます。デフォルト・ビジネス・ルールは、操作ごとに指定できます。デフォルト・ビジネス・ルールの指定やビジネス・ルールのスケジューリングは必須ではありませんが、少なくとも 1 つのデフォルト・ビジネス・ルールまたは 1 つのスケジュールされたビジネス・ルールが必要です。これをサポートするため、デフォルト・ビジネス・ルールを使用する前にこれが null であるかどうかを確認するか、または OperationSelectionRecordList のサイズを確認するようにしてください。

```
List<Operation> opList = brg.getOperations();

Iterator<Operation> opIterator = opList.iterator();
Operation op = null;
out.println("");
out.printlnBold("Operations");
// ビジネス・ルール・グループの操作を出力します。
while (opIterator.hasNext())
 {
   op = opIterator.next();
   out.printBold("Operation: ");
   out.println(op.getName());

   // 操作のデフォルト・ビジネス・ルールを取得します。
   BusinessRule defaultRule =
   op.getDefaultBusinessRule();
   // デフォルト・ルールが検出された場合には、ルールのタイプに適した
   // メソッドを使用して、そのビジネス・ルールをプリントします。
   if (defaultRule != null)
   {
     out.printlnBold("Default Destination:");
```

デフォルト・ビジネス・ルールのタイプは、RuleSet または DecisionTable です。これは、ルール成果物を処理するために正しいタイプにキャストできます。

```
     if (defaultRule instanceof RuleSet)
        out.println(RuleArtifactUtility.
        intRuleSet(defaultRule));
     else
```

```
            out.print(RuleArtifactUtility.
            tDecisionTable(defaultRule));
        }
        OperationSelectionRecordList
        opSelectionRecordList = op
          .getOperationSelectionRecordList()
          ;

        Iterator<OperationSelectionRecord>
        opSelRecordIterator = opSelectionRecordList
          .iterator();
        OperationSelectionRecord record = null;
```

OperationSelectionRecord は、ルール成果物と、そのルール成果物をアクティブにするためのスケジュールで構成されます。

```
        while (opSelRecordIterator.hasNext())
        {
          out.printlnBold("Scheduled
          Destination:");
          record = opSelRecordIterator.next();

          out.println("Start Date: " +
          record.getStartDate()
            + " - End Date: " +
            record.getEndDate());
          BusinessRule ruleArtifact = record
            .getBusinessRuleTarget();

          if (ruleArtifact instanceof RuleSet)
            out.println(RuleArtifactUtility.pr
            intRuleSet(ruleArtifact));
          else
            out.print(RuleArtifactUtility.prin
            tDecisionTable(ruleArtifact));
         }
      }
    }
    out.println("");
   } catch (BusinessRuleManagementException e)
   {
   e.printStackTrace();
   out.println(e.getMessage());
   return out.toString();
   }
}
```

## 例

例 2 の Web ブラウザー出力

```
Business Rule Group
Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
```

Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules

**Operations**
**Operation**: calculateOrderDiscount
**Default Destination**:
Rule Set
Name: calculateOrderDiscount
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
**Rule**: CopyOrder
Display Name: CopyOrder
Description: null
Expanded User Presentation: null
User Presentation: null
**Rule**: FreeGiftInitialization
Display Name: FreeGiftInitialization
Description: null
Expanded User Presentation: Product ID for Free Gift = 5001AE80 Quantity = 1 Cost = 0.0 Description = Free gift for discounted order
User Presentation: Product ID for Free Gift = {0} Quantity = {1} Cost = {2} Description = {3}Parameter Name: param0
Parameter Value: 5001AE80
Parameter Name: param1
Parameter Value: 1
Parameter Name: param2
Parameter Value: 0.0
Parameter Name: param3
Parameter Value: Free gift for discounted order
**Rule**: Rule1
Display Name: Rule1
Description: null
Expanded User Presentation: If customer is gold status, then apply a discount of 20.0 and include a free gift
User Presentation: If customer is {0} status, then apply a discount of {1} and include a free gift
Parameter Name: param0
Parameter Value: gold
Parameter Name: param1
Parameter Value: 20.0
**Rule**: Rule2
Display Name: Rule2
Description: null
Expanded User Presentation: If customer.status == silver, then provide a discount of 15.0
User Presentation: If customer.status == {0}, then provide a discount of {1}
Parameter Name: param0
Parameter Value: silver
Parameter Name: param1
Parameter Value: 15.0
**Rule**: Rule3
Display Name: Rule3
Description: Template for non-gold customers
Expanded User Presentation: If customer.status == bronze, then provide a discount of 10.0
User Presentation: If customer.status == {0}, then provide a discount of {1}
Parameter Name: param0
Parameter Value: bronze
Parameter Name: param1
Parameter Value: 10.0

**Operation**: calculateShippingDiscount
**Default Destination**:
**Decision Table**
Name: calculateShippingDiscount
Namespace: http://BRSamples/com/ibm/websphere/sample/brules

```
Init Rule: Rule1
Display Name: Rule1
Description: null
Extended User Presentation: null
User Presentation: null
```

## 例 3: AND で結合した複数のプロパティーを基準にビジネス・ルール・グループを取得する

この例も例 1 と同様ですが、Department という名前のプロパティーの値が「accounting」に設定されていると同時に、RuleType という名前のプロパティーの値が「regulatory」に設定されているビジネス・ルール・グループのみを取得します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.AndNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example3
{
static Formatter out = new Formatter();
static public String executeExample3()
{
  try
  {
    out.clear();
```

ビジネス・ルール・グループのクエリーは、ツリー構造を形成する照会ノードで構成されます。それぞれの照会ノードは、左側の項、右側の項および条件で構成されます。左側の項と右側の項には、さらに別の照会ノードを指定できます。この例では、2 つのプロパティー値の組み合わせを基準にビジネス・ルール・グループを取得します。

```
   // 2 つの条件を基準にビジネス・ルール・グループを取得します。
   // それぞれの条件に対して PropertyQueryNode を作成します。
   PropertyQueryNode propertyNode1 = QueryNodeFactory
     .createPropertyQueryNode("Department",
     QueryOperator.EQUAL,"Accounting");
   PropertyQueryNode propertyNode2 = QueryNodeFactory
     .createPropertyQueryNode("RuleType", QueryOperator.EQUAL,
     "regulatory");
   // 2 つの PropertyQueryNode を結合して AND ノードを作成します。
   AndNode andNode =
   QueryNodeFactory.createAndNode(propertyNode1, propertyNode2);

   // andNode を使用してビジネス・ルール・グループを検索します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
     .getBRGsByProperties(andNode, 0, 0);

   Iterator<BusinessRuleGroup> iterator = brgList.iterator();
```

```
   BusinessRuleGroup brg = null;
   // ビジネス・ルール・グループのリストを繰り返し処理します。
   while (iterator.hasNext())
   {
     brg = iterator.next();
     // 各ビジネス・ルール・グループの属性を出力します。
     out.println1nBold("Business Rule Group");
     out.println("Name: " + brg.getName());
     out.println("Namespace: " +
     brg.getTargetNameSpace());
     out.println("Display Name: " + brg.getDisplayName());
     out.println("Description: " + brg.getDescription());
     out.println("Presentation Time zone: "
     + brg.getPresentationTimezone());
     out.println("Save Date: " + brg.getSaveDate());

     PropertyList propList = brg.getProperties();

     Iterator<Property> propIterator =
     propList.iterator();
     Property prop = null;
     // プロパティーの名前と値を出力します。
     while (propIterator.hasNext())
     {
       prop = propIterator.next();
       out.println("¥t Property Name: " +
       prop.getName());
       out.println("¥t Property Value: " +
       prop.getValue());
     }
   }
 } catch (BusinessRuleManagementException e)
 {
 e.printStackTrace();
 out.println(e.getMessage());
 }
 return out.toString();
 }
}
```

## 例

例 3 の Web ブラウザー出力

**例 3 の実行**

```
Business Rule Group
Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ApprovalValues
Property Name: IBMSystemDisplayName
Property Value: ApprovalValues
```

## 例 4: OR で結合した複数のプロパティーを基準にビジネス・ルール・グループを取得する

この例は例 3 と同様ですが、Department という名前のプロパティーの値が「accounting」に設定されているか、または RuleType という名前のプロパティーの値が「monetary」に設定されているビジネス・ルール・グループのみを取得します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.OrNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example4
{
static Formatter out = new Formatter();
static public String executeExample4()
{
  try
  {
    out.clear();
```

さまざまなプロパティーを使用して照会を構成し、異なるビジネス・ルール・グループを返します。

```
    // 2 つの条件を基準にビジネス・ルール・グループを取得します。
    // それぞれの条件に対して PropertyQueryNode を作成します。
    PropertyQueryNode propertyNode1 = QueryNodeFactory
      .createPropertyQueryNode("Department",
      QueryOperator.EQUAL,"Accounting");
    PropertyQueryNode propertyNode2 = QueryNodeFactory
      .createPropertyQueryNode("RuleType",
      QueryOperator.EQUAL,"monetary");
    // 2 つの PropertyQueryNode を結合して OR ノードを作成します。
    OrNode orNode =
    QueryNodeFactory.createOrNode(propertyNode1,
      propertyNode2);
    // orNode を使用してビジネス・ルール・グループを検索します。
    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsByProperties(orNode, 0, 0);

    Iterator<BusinessRuleGroup> iterator = brgList.iterator();

    BusinessRuleGroup brg = null;
    // ビジネス・ルール・グループのリストを繰り返し処理します。
    while (iterator.hasNext())
    {
      brg = iterator.next();
      // 各ビジネス・ルール・グループの属性を出力します。
      out.println1nBold("Business Rule Group");
      out.println("Name: " + brg.getName());
      out.println("Namespace: " +
      brg.getTargetNameSpace());
      out.println("Display Name: " + brg.getDisplayName());
      out.println("Description: " + brg.getDescription());
      out.println("Presentation Time zone: "
```

```
        + brg.getPresentationTimezone());
      out.println("Save Date: " + brg.getSaveDate());

      PropertyList propList = brg.getProperties();

      Iterator<Property> propIterator =
      propList.iterator();
      Property prop = null;
      // プロパティーの名前と値を出力します。
      while (propIterator.hasNext())
      {
        prop = propIterator.next();
        out.println("¥t Property Name: " +
        prop.getName());
        out.println("¥t Property Value: " +
        prop.getValue());
      }
      out.println("");
    }
  } catch (BusinessRuleManagementException e)
 {
   e.printStackTrace();
   out.println(e.getMessage());
 }
 return out.toString();
 }
}
```

## 例

例 4 の Web ブラウザー出力

**例 4 の実行**

```
Business Rule Group
Name: ApprovalValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ApprovalValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
Property Value: Accounting
Property Name: RuleType
Property Value: regulatory
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ApprovalValues
Property Name: IBMSystemDisplayName
Property Value: ApprovalValues

Business Rule Group
Name: DiscountRules
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: DiscountRules
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: Department
Property Value: Accounting
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: RuleType
Property Value: monetary
```

```
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: DiscountRules
Property Name: IBMSystemDisplayName
Property Value: DiscountRules
```

## 例 5: 複雑な照会を使用してビジネス・ルール・グループを取得する

この例では、例 3 と例 4 を組み合わせて、さらに複雑な照会の作成方法を説明します。この例では、2 つの照会条件を組み合わせた照会を使用して検索を行います。最初の照会条件は、Department という名前のプロパティーの値が「General」に設定されているか、MissingProperty という名前のプロパティーの値が「somevalue」に設定されているビジネス・ルール・グループを取得することです。この条件に、RuleType という名前のプロパティーの値が「messages」に設定されているという条件を、AND を使用して結合します。

付録には、ビジネス・ルール・グループを照会するその他の例が記載されています。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Property;
import com.ibm.wbiserver.brules.mgmt.PropertyList;
import com.ibm.wbiserver.brules.mgmt.query.AndNode;
import com.ibm.wbiserver.brules.mgmt.query.OrNode;
import com.ibm.wbiserver.brules.mgmt.query.PropertyQueryNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryNodeFactory;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example5
{
static Formatter out = new Formatter();
static public String executeExample5()
{
  try
  {
    out.clear();

    // 3 つの条件を基準にビジネス・ルール・グループを取得します。条件の
    // うちの 2 つは結合して 1 つの OR ノードを構成します。
    // OR ノードを構成するそれぞれの条件に対して PropertyQueryNode を作成します。
    PropertyQueryNode propertyNode1 = QueryNodeFactory
      .createPropertyQueryNode("Department",
      QueryOperator.EQUAL,"General");
    PropertyQueryNode propertyNode2 = QueryNodeFactory
      .createPropertyQueryNode("MissingProperty",
    QueryOperator.EQUAL, "SomeValue");
    // 2 つの PropertyQueryNode を結合して OR ノードを作成します。
    OrNode orNode =
    QueryNodeFactory.createOrNode(propertyNode1, propertyNode2);
    // 3 つ目の PropertyQueryNode を作成します。
      PropertyQueryNode propertyNode3 = QueryNodeFactory
      .createPropertyQueryNode("RuleType",
    QueryOperator.EQUAL,"messages");
```

左側の条件と右側の条件を結合して AND ノードを作成します。この AndNode が照会ツリーのルートになります。

```
    // OR ノードと 3 つ目の PropertyQueryNode を結合します。
    AndNode AndNode andNode =
    QueryNodeFactory.createAndNode(propertyNode3, orNode);

    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsByProperties(andNode, 0, 0);

    Iterator<BusinessRuleGroup> iterator = brgList.iterator();

    BusinessRuleGroup brg = null;
    // ビジネス・ルール・グループのリストを繰り返し処理します。
    while (iterator.hasNext())
    {
      brg = iterator.next();
      // 各ビジネス・ルール・グループの属性を出力します。
      out.println1nBold("Business Rule Group");
      out.println("Name: " + brg.getName());
      out.println("Namespace: " +
        brg.getTargetNameSpace());
      out.println("Display Name: " + brg.getDisplayName());
      out.println("Description: " + brg.getDescription());
      out.println("Presentation Time zone: "
        + brg.getPresentationTimezone());
      out.println("Save Date: " + brg.getSaveDate());
      PropertyList propList = brg.getProperties();

      Iterator<Property> propIterator =
      propList.iterator();
      Property prop = null;
      // プロパティーの名前と値を出力します。
      while (propIterator.hasNext())
      {
        prop = propIterator.next();
        out.println("¥t Property Name: " +
        prop.getName());
        out.println("¥t Property Value: " +
        prop.getValue());
      }
    }
   }   catch (BusinessRuleManagementException e)
  {
  e.printStackTrace();
  out.println(e.getMessage());
  }
 return out.toString();
 }
}
```

## 例

例 5 の Web ブラウザー出力

**例 5 の実行**

```
Business Rule Group
Name: ConfigurationValues
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Display Name: ConfigurationValues
Description: null
Presentation Time zone: LOCAL
Save Date: Sun Jan 06 17:56:51 CST 2008
Property Name: IBMSystemVersion
Property Value: 6.2.0
Property Name: Department
```

```
Property Value: General
Property Name: RuleType
Property Value: messages
Property Name: IBMSystemTargetNameSpace
Property Value: http://BRSamples/com/ibm/websphere/sample/brules
Property Name: IBMSystemName
Property Value: ConfigurationValues
Property Name: IBMSystemDisplayName
Property Value: ConfigurationValues
```

## 例 6: ビジネス・ルール・グループのプロパティーを更新して公開する

この例では、ビジネス・ルール・グループのプロパティーを更新してから、ビジネス・ルール・グループを公開します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.UserDefinedProperty;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example6
{
static Formatter out = new Formatter();

static public String executeExample6()
{
 try
 {
   out.clear();
   out.println1nBold("Business Rule Group before publish:");
   // 単一のプロパティー値を基準にビジネス・ルール・グループを取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
     .getBRGsBySingleProperty("Department",
     QueryOperator.EQUAL,"General", 0, 0);

   if (brgList.size() > 0)
   {
     // リストから最初のビジネス・ルール・グループを取得します。
     BusinessRuleGroup brg = brgList.get(0);
     // ビジネス・ルール・グループからプロパティーを取得します。
     UserDefinedProperty userDefinedProperty =
     (UserDefinedProperty) brg
       .getProperty("Department");

     out.println("Business Rule Group: " + brg.getName());
     out.println("Department Property value: "
     + brg.getProperty("Department").getValue());
```

getProperty メソッドは参照によってプロパティーを返すため、そのプロパティーを変更するとビジネス・ルール・グループが直接変更されます。

```
     // brg のプロパティー値を変更します。
     // これにより、brg オブジェクトのプロパティー値が直接
     更新されます。
     userDefinedProperty.setValue("GeneralConfig");
     // 元のリストを使用するか、ビジネス・ルール・グループの
     // 新しいリストを作成します。
```

```
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();
    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);
```

ビジネス・ルール・グループの変更内容を公開するには、BusinessRuleManager クラスを使用します。変更を公開するには、1 つの項目のみを公開する場合でも、BusinessRuleManager の publish メソッドにリストを渡します。

```
    // 更新されたビジネス・ルール・グループを含むリストを公開します。
    BusinessRuleManager.publish(publishList, true);

    out.println("");

    // ビジネス・ルール・グループを再取得し、変更内容が
    // 公開されていることを確認します。
    out.println1nBold("Business Rule Group after publish:");
    brgList = BusinessRuleManager
    .getBRGsBySingleProperty("Department",
    QueryOperator.EQUAL, "GeneralConfig", 0, 0);

    brg = brgList.get(0);

    out.println("Business Rule Group: " + brg.getName());
    // プロパティー値を表示して変更を示します。
    out.println("Department Property value: "
    + brg.getProperty("Department").getValue());
  }
 } catch (BusinessRuleManagementException e)
 {
 e.printStackTrace();
 out.println(e.getMessage());
 }
return out.toString();
}
}
```

### 例

例 6 の Web ブラウザー出力

**例 6 の実行**

```
Business Rule Group before publish:
Business Rule Group: ConfigurationValues
Department Property value: General

Business Rule Group after publish:
Business Rule Group: ConfigurationValues
Department Property value: GeneralConfig
```

## 例 7: 複数のビジネス・ルール・グループのプロパティーを更新して公開する

この例では、複数のビジネス・ルール・グループのプロパティーを更新して公開します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
```

```
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.UserDefinedProperty;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example7
{
static Formatter out = new Formatter();

static public String executeExample7()
{
  try
  {
    out.clear();
    out.printlnBold("Business Rule Group before publish:");
    // 単一のプロパティー値を基準にビジネス・ルール・グループを取得します。
    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsBySingleProperty("Department",
      QueryOperator.EQUAL, "Accounting", 0, 0);

    Iterator<BusinessRuleGroup> iterator = brgList.iterator();

    BusinessRuleGroup brg = null;

    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

    // すべてのビジネス・ルール・グループを繰り返し処理して、
    // プロパティーを変更します。
    while (iterator.hasNext())
    {
      // ビジネス・ルール・グループからプロパティーを取得します。
      brg = iterator.next();

      out.println("Business Rule Group: " + brg.getName());

      // ビジネス・ルール・グループからプロパティーを取得します。
      UserDefinedProperty prop = (UserDefinedProperty) brg
        .getProperty("Department");
      out.println("Department Property value: "
      +
      brg.getProperty("Department").getValue())
      ;

      // brg のプロパティー値を変更します。
      // これにより、brg オブジェクトのプロパティー値が直接
      更新されます。
      prop.setValue("Finance");
```

変更したビジネス・ルール・グループのそれぞれを追加します。

```
      // 変更したビジネス・ルール・グループをリストに追加します。
      publishList.add(brg);
      }

      // 更新されたビジネス・ルール・グループを含むリストを
      公開します。
      BusinessRuleManager.publish(publishList, true);

      out.println("");

      // ビジネス・ルール・グループを再取得し、変更内容が
      // 公開されていることを確認します。
      out.printlnBold("Business Rule Group after
      publish:");
```

```
      brgList = BusinessRuleManager
        .getBRGsBySingleProperty("Department",
        QueryOperator.EQUAL,
        "Finance", 0, 0);
      iterator = brgList.iterator();

      while (iterator.hasNext())
      {
        brg = iterator.next();
        out.println("Business Rule Group: " +
        brg.getName());
        out.println("Department Property value: "
         +
         brg.getProperty("Department").getVa
         lue());
      }
} catch (BusinessRuleManagementException e)
{
  e.printStackTrace();
  out.println(e.getMessage());
}
return out.toString();
}
}
```

### 例

例 7 の Web ブラウザー出力

**例 7 の実行**

```
Business Rule Group before publish:
Business Rule Group: ApprovalValues
Department Property value: Accounting
Business Rule Group: DiscountRules
Department Property value: Accounting

Business Rule Group after publish:
Business Rule Group: ApprovalValues
Department Property value: Finance
Business Rule Group: DiscountRules
Department Property value: Finance
```

## 例 8: ビジネス・ルール・グループのデフォルト・ビジネス・ルールを変更する

この例では、デフォルト・ビジネス・ルールを、特定の操作に使用可能なターゲット・リストに含まれる別のビジネス・ルールに変更します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example8
{
```

```
static Formatter out = new Formatter();

static public String executeExample8()
{
  try
  {
    out.clear();

  // ターゲット名前空間と名前によってビジネス・ルール・グループを取得します。
  List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
      "http://BRSamples/com/ibm/websphere
      /sample/brules",
      QueryOperator.EQUAL,
      "DiscountRules",
      QueryOperator.EQUAL, 0, 0);

  if (brgList.size() > 0)
  {
    out.printlnBold("Business Rule Group before publish:");
    // リストから最初のビジネス・ルール・グループを取得します。
    // ターゲット名前空間と名前の組み合わせは固有であることから、
    // これがリスト内の唯一のビジネス・ルール・グループとなるはずです。
    BusinessRuleGroup brg = brgList.get(0);

    out.print("Business Rule Group: ");
    out.println(brg.getName());

    // デフォルト・ビジネス・ルールを更新する
    // ビジネス・ルール・グループの操作を取得します。
    Operation op =
    brg.getOperation("calculateShippingDiscount");
```

デフォルト・ビジネス・ルールを取得した後、操作に使用可能なターゲット・リストに含まれる別のルールで更新します。ルール・セットとデシジョン・テーブルはそれぞれの操作に固有であるため、デフォルトに設定したり、操作の別のスケジュールに使用できるのは、その操作のビジネス・ルール成果物のみです。

```
    // 操作のデフォルト・ビジネス・ルールを取得します。
    BusinessRule defaultRule =
    op.getDefaultBusinessRule();
    out.print("Default Rule: ");
    out.println(defaultRule.getName());

    // この操作に使用可能なビジネス・ルールのリストを
    取得します。
    List<BusinessRule> ruleList =
    op.getAvailableTargets();

    Iterator<BusinessRule> iterator =
    ruleList.iterator();
    BusinessRule rule = null;

    // 現行のデフォルト・ビジネス・ルール
    とは
    // 異なるビジネス・ルールを
    // 検索します。
    while (iterator.hasNext())
    {
      rule = iterator.next();
      if
      (!defaultRule.getName().equals(rule.getName()))
      {
```

操作オブジェクトのデフォルト・ビジネス・ルールを設定します。デフォルト・ビジネス・ルールをヌルに設定すると、あらゆるデフォルト・ビジネス・ルールが操作から除去されますが、すべての操作にはデフォルト・ビジネス・ルールを指定することをお勧めします。

```
        // デフォルト・ビジネス・ルールを異なる
        // ビジネス・ルールに設定します。
        // この変更は、操作オブジェクトに対して直接
        // 行われます。
        op.setDefaultBusinessRule(rule);
        break;
      }
    }
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();
    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);
    // 更新されたビジネス・ルール・グループを含むリストを
     公開します。
    BusinessRuleManager.publish(publishList, true);

    out.println("");

    // ビジネス・ルール・グループを再取得し、変更内容が
    // 公開されていることを確認します。

    out.printlnBold("Business Rule Group after publish:");
    brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
      "http://BRSamples/com/ibm/websphere/sample/brules",
      QueryOperator.EQUAL, "DiscountRules",
      QueryOperator.EQUAL, 0, 0);

    brg = brgList.get(0);
    out.println("Business Rule Group: " + brg.getName());
    op = brg.getOperation("calculateShippingDiscount");

    // 操作のデフォルト・ビジネス・ルールを取得します。
    defaultRule = op.getDefaultBusinessRule();
    out.print("Default Rule: ");
    out.println(defaultRule.getName());
   }
 } catch (BusinessRuleManagementException e)
 {
   e.printStackTrace();
   out.println(e.getMessage());
 }
 return out.toString();
 }
}
```

## 例

例 8 の Web ブラウザー出力

**例 8 の実行**

```
Business Rule Group before publish:
Business Rule Group: DiscountRules
Default Rule: calculateShippingDiscount

Business Rule Group after publish:
Business Rule Group: DiscountRules
Default Rule: calculateShippingDiscountHoliday
```

## 例 9: ビジネス・ルール・グループの操作に別のルールをスケジュールする

この例では、特定の操作を公開してから 1 時間、ビジネス・ルールがアクティブになるようにスケジュールします。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example9 {
static Formatter out = new Formatter();

static public String executeExample9()
{
 try
 {
   out.clear();

   // ターゲット名前空間と名前によってビジネス・ルール・グループを取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
     .getBRGsByTNSAndName(
     "http://BRSamples/com/ibm/websphere
     /sample/brules",
     QueryOperator.EQUAL,
     "DiscountRules",
     QueryOperator.EQUAL, 0, 0);

   if (brgList.size() > 0)
   {
     out.println("");
     out.printlnBold("Business Rule Group before publish:");
     // リストから最初のビジネス・ルール・グループを取得します。
     // ターゲット名前空間と名前の組み合わせは固有で
     あることから、
     // これがリスト内の唯一のビジネス・ルール・グループとなるはずです。
     BusinessRuleGroup brg = brgList.get(0);

     // 新しいビジネス・ルールをスケジュールする
     // ビジネス・ルール・グループの操作を取得します。
     Operation op =
     brg.getOperation("calculateShippingDiscount");

     printOperationSelectionRecord(op);
     // この操作に使用可能なビジネス・ルールのリストを取得します。
     List<BusinessRule> ruleList =
     op.getAvailableTargets();

     // リストの先頭にあるルールを取得します。
     // このルールを使用して操作をスケジュールします。
     BusinessRule rule = ruleList.get(0);

     // スケジュールされたビジネス・ルールのリストを取得します。
```

```
    OperationSelectionRecordList opList = op
      .getOperationSelectionRecordList();
    // ビジネス・ルールの終了日を現在より後の日付で作成します。
    Date future = new Date();
    long futureTime = future.getTime() + 3600000;
```

ルールを新規にスケジュールするときには、ルールと併せて開始日と終了日を指定できます。開始日をヌルに設定すると、ルールは公開と同時にアクティブになります。終了日をヌルに設定すると、ルールの終了日は指定されないことになります。スケジュールのオーバーラップは許可されません。これを確認するには、操作の validate メソッドを呼び出します。

```
    // スケジュールされた新しいビジネス・ルールを
    // 作成します。公開したときに即時にアクティブ
    // になるように現在の日付を指定し、さらに現在
    // より後の日付を指定します。
    newOperationSelectionRecord(new Date(),
      new Date(futureTime), rule);
    // 新規にスケジュールしたビジネス・ルールを、
    // スケジュールされたルールのリストに追加します。
    opList.addOperationSelectionRecord(newRecord);
```

操作を検証して、オーバーラップしていないことを確認します。

```
    // リストを検証してオーバーラップがないことを確認します。
    List<Problem> problems = op.validate();
    if (problems.size() == 0)
    {
      // 元のリストを使用するか、ビジネス・ルール・グループの
      // 新しいリストを作成します。
      List<BusinessRuleGroup> publishList = new
      ArrayList<BusinessRuleGroup>();
      // 変更したビジネス・ルール・グループをリストに追加します。
      publishList.add(brg);
      // 更新されたビジネス・ルール・グループを含むリストを
      公開します。
      BusinessRuleManager.publish(publishList, true);
      out.println("");

      // ビジネス・ルール・グループを再取得し、
      変更内容が
      // 公開されていることを確認します。
      out.printlnBold("Business Rule Group after
      publish:");

      brgList =
      BusinessRuleManager.getBRGsByTNSAndName(
        "http://BRSamples/com/ibm/websphere
        /sample/brules",
        QueryOperator.EQUAL,
        "DiscountRules",
        QueryOperator.EQUAL, 0, 0);
        brg = brgList.get(0);

        op =
        brg.getOperation("calculateShippingDiscount");

        printOperationSelectionRecord(op);
      }
       // 検証エラーを処理します。
    }
} catch (BusinessRuleManagementException e)
{
e.printStackTrace();
out.println(e.getMessage());
}
```

```
 return out.toString();
}
/*
操作の操作選択レコードをプリントするためのメソッド。開始日、
終了日、およびスケジュールされた期間のルール成果物の名前が
プリントされます。
*/
private static void printOperationSelectionRecord(Operation op)
{
 OperationSelectionRecordList opSelectionRecordList = op
   .getOperationSelectionRecordList();
 Iterator<OperationSelectionRecord> opSelRecordIterator =
 opSelectionRecordList
   .iterator();
 OperationSelectionRecord record = null;
 while (opSelRecordIterator.hasNext())
 {
   out.println1nBold("Scheduled Destination:");
   record = opSelRecordIterator.next();
   out.println("Start Date: " + record.getStartDate()
   + " - End Date: " + record.getEndDate());
   BusinessRule ruleArtifact = record.getBusinessRuleTarget();
   out.println("Rule: " + ruleArtifact.getName());
   }
 }
}
```

### 例

例 9 の Web ブラウザー出力

**例 9 の実行**

```
Business Rule Group before publish:
Scheduled Destination:
Start Date: Thu Dec 01 00:00:00 CST 2005 - End Date: Sun Dec 25 00:00:00 CST 2005
Rule: calculateShippingDiscountHoliday

Business Rule Group after publish:
Scheduled Destination:
Start Date: Thu Dec 01 00:00:00 CST 2005 - End Date: Sun Dec 25 00:00:00 CST 2005
Rule: calculateShippingDiscountHoliday
Scheduled Destination:
Start Date: Mon Jan 07 21:08:31 CST 2008 - End Date: Mon Jan 07 22:08:31 CST 2008
Rule: calculateShippingDiscount
```

## 例 10: ルール・セットのテンプレートのパラメーター値を変更する

この例では、テンプレートで定義されたルール・インスタンスを変更するためにパラメーター値を変更し、変更後のルール・インスタンスを公開します。

```
package com.ibm.websphere.sample.brules.mgmt;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import
```

```
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;

public class Example10
{
static Formatter out = new Formatter();

static public String executeExample10()
{
  try
  {
    out.clear();

    // ターゲット名前空間と名前によってビジネス・ルール・グループ
    を取得します。
    List<BusinessRuleGroup> brgList = BusinessRuleManager
      .getBRGsByTNSAndName(
        "http://BRSamples/com/ibm/websphere
        /sample/brules",
        QueryOperator.EQUAL,
        "ApprovalValues",
        QueryOperator.EQUAL, 0, 0);
    if (brgList.size() > 0)
    {
      // リストから最初のビジネス・ルール・グループを取得します。
      // ターゲット名前空間と名前の組み合わせは固有で
      あることから、
      // これがリスト内の唯一のビジネス・ルール・グループと
      なるはずです。
      BusinessRuleGroup brg = brgList.get(0);
      // 変更するビジネス・ルールが関連付けられている
      // 操作をビジネス・ルール・グループから取得します。
      // 特定の操作にはビジネス・ルールが関連付けられて
      // います。
      Operation op = brg.getOperation("getApprover");

      // 変更するビジネス・ルールを操作から
      取得します。
      List<BusinessRule> ruleList =
      op.getBusinessRulesByName(
        "getApprover", QueryOperator.EQUAL, 0,
        0);

      if (ruleList.size() > 0)
      {
        out.println("");
        out.printlnBold("Rule set before publish:");
        // 変更するルールを取得します。ルールは、
        ターゲット名前空間と
        // 名前によって固有のものになりますが、
        この例の場合には、
        // 「getApprover」という名前のビジネス・ルール
        しかありません。
        RuleSet ruleSet = (RuleSet) ruleList.get(0);
        out.print(RuleArtifactUtility.printRuleSet(rule
        Set));
```

ルール・セットに含まれるすべてのルールは、ルール・ブロック内にあります。サポートされるルール・ブロックは 1 つのみで、ルール・ブロックを取得するには getFirstRuleBlock メソッドを使用する必要があります。

```
      // ルール・セットには、ルール・ブロック内に定義された
      すべてのルールが含まれます。
      RuleBlock ruleBlock =
      ruleSet.getFirstRuleBlock();
```

```
Iterator<RuleSetRule> ruleIterator =
ruleBlock.iterator();

// ルール・ブロック内のルールを繰り返し処理し、「LargeOrderApprover」
という名前の
// ルール・インスタンスを見つけます。
while (ruleIterator.hasNext())
{
  RuleSetRule rule = ruleIterator.next();
```

ルール・テンプレートによってルールが定義されていない場合、そのルールで取得できるのは Web 表示のみです。テンプレートで定義されていないルールは、更新できません。ルールの名前が不明の場合は、そのルールがテンプレートによって定義されているかどうかを調べることが最善です。

```
// ルールを変更するためには、そのルールがテンプレートによって
定義されている
// 必要があります。現行のルールが
テンプレートに
// 基づいているかどうかをチェックします。
if (rule instanceof
RuleSetTemplateInstanceRule)
{
```

TemplateInstance オブジェクトを使用して、ルールを作成します。

```
  // ルール・テンプレート・インスタンスを取得します。
  RuleSetTemplateInstanceRule
  templateInstance =
  (RuleSetTemplateInstanceRule) rule;

  // 変更するルールと一致するルール・
  インスタンスを
  // 見つけます。
  if
  (templateInstance.getName().equals(
    "LargeOrderApprover"))
  {
```

テンプレート・インスタンスで変更できるのはパラメーター値のみです。パラメーターを変更するには、ParameterValue を取得し、これを適切な値に設定します。ParameterValue は参照によって渡されるため、更新はルール、ルール・セット、およびビジネス・ルール・グループで直接行われます。

```
    // ルール・インスタンスからパラメーターを
    取得します。
    ParameterValue parameter =
    templateInstance
      .getParameterValue("par
      am2");

    // パラメーターの値を
    変更します。
    parameter.setValue("superviso
    r");
    break;
  }
 }
}
// 元のリストを使用するか、ビジネス・ルール・グループの
// 新しいリストを作成します。
List<BusinessRuleGroup> publishList = new
ArrayList<BusinessRuleGroup>();
```

```
    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);

    // 更新されたビジネス・ルール・グループを含むリストを
     公開します。
     BusinessRuleManager.publish(publishList, true);

    out.println("");
    // ビジネス・ルール・グループを再取得し、
    変更内容が
    // 公開されていることを確認します。
    out.printlnBold("Rule set after publish:");

    brgList = BusinessRuleManager
      .getBRGsByTNSAndName(
        "http://BRSamples/com/ibm/websphere/sample/brules",
        QueryOperator.EQUAL, "ApprovalValues",
        QueryOperator.EQUAL, 0, 0);

    brg = brgList.get(0);
    op = brg.getOperation("getApprover");
    ruleList = op.getBusinessRulesByName(
      "getApprover", QueryOperator.EQUAL, 0,0);

    ruleSet = (RuleSet) ruleList.get(0);
    out.print(RuleArtifactUtility.printRuleSet(ruleSet));
    }
  }
  } catch (BusinessRuleManagementException e)
  {
    e.printStackTrace();
    out.println(e.getMessage());
  }
  return out.toString();
  }
}
```

## 例

例 10 の Web ブラウザー出力

**例 10 の実行**

```
Rule set before publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of manager
User Presentation: If the number of items order is above {0} and
the order is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: manager
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer
```

```
Rule set after publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer
```

## 例 11: 新規ルールをテンプレートからルール・セットに追加する

この例では、新規ルールをテンプレートからルール・セットに追加します。新規ルール・インスタンスを作成する前に、その新規ルール・インスタンスのパラメーターを作成します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRuleTemplate;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example11
{
static Formatter out = new Formatter();

static public String executeExample11()
{
 try
 {
  out.clear();
  // ターゲット名前空間と名前によってビジネス・ルール・グループ
  を取得します。
  List<BusinessRuleGroup> brgList = BusinessRuleManager
   .getBRGsByTNSAndName(
   "http://BRSamples/com/ibm/websphere
```

```
 /sample/brules",
 QueryOperator.EQUAL,
 "ApprovalValues",
 QueryOperator.EQUAL, 0, 0);

if (brgList.size() > 0)
{
 // リストから最初のビジネス・ルール・グループを取得します。
 // ターゲット名前空間と名前の組み合わせは固有で
 あることから、
 // これがリスト内の唯一のビジネス・ルール・グループと
 なるはずです。
 BusinessRuleGroup brg = brgList.get(0);
 // 変更するビジネス・ルールが関連付けられている
 // 操作をビジネス・ルール・グループから取得します。
 // 特定の操作にはビジネス・ルールが関連付けられて
 // います。
 Operation op = brg.getOperation("getApprover");

 // 変更するビジネス・ルールを操作から
 取得します。
 List<BusinessRule> ruleList =
 op.getBusinessRulesByName(
  "getApprover", QueryOperator.EQUAL, 0,0);

 if (ruleList.size() > 0)
 {
  out.println("");
  out.printlnBold("Rule set before publish:");
  // 変更するルールを取得します。ルールは、
  // ターゲット名前空間と名前によって固有のものになりますが、
  // この例の場合には、「getApprover」という名前のビジネス・ルール
  しかありません。
  RuleSet ruleSet = (RuleSet) ruleList.get(0);
  out.print(RuleArtifactUtility.printRuleSet(rule
  Set));
```

新しいルールをルール・セットに追加するには、ルール・セットから適切なテンプレートを見つけ、そのテンプレートからインスタンスを作成する必要があります。テンプレートは、名前を基準に見つけられます。

```
    // ルール・テンプレートのリストを取得します。
    ListRuleSetRuleTemplate> ruleTemplates =
    ruleSet
     .getRuleTemplates();

    Iterator<RuleSetRuleTemplate> templateIterator
    = ruleTemplates
     .iterator();

    while (templateIterator.hasNext())
    {
     RuleSetRuleTemplate template =
     templateIterator.next();

     // 新規ルールを作成するために使用するテンプレートを
     見つけます。
     if
     (template.getName().equals("Template_Larg
     eOrder"))
     {
```

テンプレート・インスタンスを作成するには、パラメーターのリストを作成する必要があります。

```
// このテンプレート・ルール・インスタンスの
パラメーターの
// リストを作成します。
List<ParameterValue> paramList =
new ArrayList<ParameterValue>();

// テンプレート定義から、
特定のパラメーターを取得して
// 値を設定します。
Parameter param =
template.getParameter("param0");
ParameterValue paramValue = param
 .createParameterValue("
 20");

// パラメーターをリストに追加します。
paramList.add(paramValue);

// 次のパラメーターを取得して値を
設定します。
param = template.getParameter("param1");
paramValue =
param.createParameterValue("7500");

// パラメーターをリストに追加します。
paramList.add(paramValue);

// 次のパラメーターを取得して値を
設定します。
param =
template.getParameter("param2");
 paramValue = param
 .createParameterValue("
 2nd-line manager");

// パラメーターをリストに追加します。
paramList.add(paramValue);
```

パラメーターの作成が完了したら、テンプレート・インスタンスを作成します。

```
// パラメーター・リストを使用して
テンプレート・ルール・インスタンスを
// 作成します。
RuleSetTemplateInstanceRule
 templateInstance = template
 .createRuleFromTemplate
 ("ExtraLargeOrder",
 paramList);
// ルール・セットのルール・ブロックを
 取得します。
RuleBlock ruleBlock =
ruleSet.getFirstRuleBlock();
```

テンプレート・インスタンスの作成が完了したら、それをルール・ブロックに追加します。ルール・ブロックに追加したテンプレート・インスタンスは、他のテンプレート・ルール・インスタンスとの間で順序を設定できます。

```
// テンプレート・ルールを
ルール・ブロックに追加します。
ruleBlock.addRule(templateInstance)
;

break;
}
}
```

```
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

    // 変更したビジネス・ルール・グループをリストに
    追加します。
    publishList.add(brg);

    // 更新されたビジネス・ルール・グループを含むリストを
     公開します。
    BusinessRuleManager.publish(publishList, true);

    out.println("");

    // ビジネス・ルール・グループを再取得し、
    変更内容が
    // 公開されていることを確認します。
    out.printlnBold("Rule set after publish:");

    brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
      "http://BRSamples/com/ibm/websphere
      /sample/brules",
      QueryOperator.EQUAL,
      "ApprovalValues",
      QueryOperator.EQUAL, 0, 0);

    brg = brgList.get(0);
    op = brg.getOperation("getApprover");
    ruleList = op.getBusinessRulesByName(
         "getApprover", QueryOperator.EQUAL,
         0, 0);

    ruleSet = (RuleSet) ruleList.get(0);
    out.print(RuleArtifactUtility.printRuleSet(rule
    Set));
    }
  }
 } catch (BusinessRuleManagementException e)
 {
  e.printStackTrace();
  out.println(e.getMessage());
 }
 return out.toString();
 }
}
```

## 例

例 11 の Web ブラウザー出力

**例 11 の実行**

```
Rule set before publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
```

```
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer

Rule set after publish:
Rule Set
Name: getApprover
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
Rule: LargeOrderApprover
Display Name: LargeOrderApprover
Description: null
Expanded User Presentation: If the number of items order is above 10 and
the order is above $5000, then it requires the approval of supervisor
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 10
Parameter Name: param1
Parameter Value: 5000
Parameter Name: param2
Parameter Value: supervisor
Rule: DefaultApprover
Display Name: DefaultApprover
Description: null
Expanded User Presentation: approver = peer
User Presentation: approver = {0}
Parameter Name: param0
Parameter Value: peer
Rule: ExtraLargeOrder
Display Name:
Description: null
Expanded User Presentation: If the number of items order is above 20 and
the order is above $7500, then it requires the approval of 2nd-line manager
User Presentation: If the number of items order is above {0} and the order
is above ${1}, then it requires the approval of {2}
Parameter Name: param0
Parameter Value: 20
Parameter Name: param1
Parameter Value: 7500
Parameter Name: param2
Parameter Value: 2nd-line manager
```

## 例 12: パラメーター値の変更によってデシジョン・テーブルのテンプレートを変更して公開する

この例では、条件とアクションの両方がテンプレートで定義されているデシジョン・テーブルについて、パラメーター値を変更することでその条件とアクションを変更した後、これを公開します。

デシジョン・テーブルの条件とアクションを変更する最も簡単な方法は、各条件レベルのテンプレートおよび各アクションに割り当てられている固有の名前を使用することです。固有の名前を検索してから、そのテンプレートで定義されたテンプレート・インスタンスを変更できます。特定のテンプレートのテンプレート・インスタンスを変更すると、そのテンプレートで定義された当該レベルの条件値がすべて

更新されます。アクション式の各インスタンスは固有なので、特定のインスタンスを変更しても、他のインスタンスが変更されることはありません。

この例では、更新対象の特定のケース・エッジ、特定のパラメーター値、および特定のテンプレートで定義されたアクション式を見つけやすくするために、多数の追加メソッドが作成されています。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Vector;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example12 {
static Formatter out = new Formatter();

static public String executeExample12()
{
 try
 {
   out.clear();
   // ターゲット名前空間と名前によってビジネス・ルール・グループ
   を取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
    "http://BRSamples/com/ibm/websphere
    /sample/brules",
    QueryOperator.EQUAL,
    "ConfigurationValues",
    QueryOperator.EQUAL, 0, 0);

   if (brgList.size() > 0)
   {
    // リストから最初のビジネス・ルール・グループを取得します。
    // ターゲット名前空間と名前の組み合わせは固有で
    あることから、
    // これがリスト内の唯一のビジネス・ルール・グループと
    なるはずです。
    BusinessRuleGroup brg = brgList.get(0);

    // 変更するビジネス・ルールが関連付けられている
    // 操作をビジネス・ルール・グループから取得します。
    // 特定の操作にはビジネス・ルールが関連付けられて
    // います。
    Operation op = brg.getOperation("getMessages");
```

```
// この操作に使用可能なすべてのビジネス・ルールを
取得します。
List<BusinessRule> ruleList =
op.getAvailableTargets();

// この操作には、1 つのビジネス・ルールしかありません。
これが、
// 更新対象のビジネス・ルールです。
DecisionTable decisionTable = (DecisionTable)
ruleList.get(0);
out.println("");
out.printlnBold("Decision table before publish:");
out
 .print(RuleArtifactUtility
 .printDecisionTable(decisionT
  able));
```

初期ルールと条件、およびアクションは、ツリー・ブロックに含まれます。このツリー・ブロックから、ルート・ノードを取得できます。

```
// デシジョン・テーブルのすべての条件とアクション
が含まれる
// ツリー・ブロックを取得します。
TreeBlock treeBlock = decisionTable.getTreeBlock();
// ツリー・ブロックから、デシジョン・テーブルの
ナビゲーションの
// 開始点となるツリー・ノードを取得
します。
TreeNode treeNode = treeBlock.getRootNode();
```

更新する条件は、「Condition Value Template 2.1」という名前のテンプレートで定義されています。メソッド getCaseEdge は、TreeNode から該当するケース・エッジまで再帰的に検索し、そのテンプレートが定義されたケース・エッジを見つけます。このメソッドでは、テンプレートが定義されているレベルと現行のレベルが既知であることを想定しています。複数のケース・エッジに同じ名前が使用されている場合、このメソッドを使用して、テンプレートを持つケース・エッジを特定の名前を基準に検索することができます。

```
// ルートよりも 1 つ下のレベルにある、パラメーターの
// 値が特定の名前に設定された特定のテンプレートを持つ
// ケース・エッジを検索します。先頭から開始するため、
// 現行の深さは 0 となります。
CaseEdge caseEdge = getCaseEdge(treeNode, "param0",
"Condition Value Template 2.1", 1, 0);
```

ケース・エッジが検出されたら、条件の ConditionValueTemplateInstance を取得できます。

```
if (caseEdge != null)
{
 // ケース・エッジが検出されました。ケース・エッジの
 値の
 // 定義を取得します。
 TreeConditionValueDefinition condition =
  caseEdge
  .getValueDefinition();
 // テンプレートで定義された条件式を取得
  します。
  TemplateInstanceExpression conditionExpression
  = condition
   .getConditionValueTemplateInstance(
 );
```

ConditionValueTemplateInstance において、getParameterValue メソッドを使用して該当するパラメーター値を取得し、その値を更新できます。

```
// 式のテンプレートを取得します。
Template conditionTemplate =
conditionExpression
 .getTemplate();

 // テンプレートが正しいことを確認します。これは、1 つの条件値
 に対して複数の
 // テンプレートが存在する場合があるためですが、実際に適用され
 ているテンプレートは
 // 1 つのみです。
 if (conditionTemplate.getName().equals(
  "Condition Value Template 2.1"))
 {
  // パラメーター値を取得します。
  ParameterValue parameterValue =
  getParameterValue("param0",
   conditionExpression);

  // 新しいパラメーター値を設定します。
  parameterValue.setValue("info");
 }
```

テンプレートで定義されていて、更新する必要がある異なるアクション式を取得できます。 getActionExpressions メソッドは、Action Value Template 1 という名前のテンプレートで定義されたすべてのアクションを返します。

```
ConditionNode conditionNode = (ConditionNode)
treeNode;

// ケース・エッジのツリー・ノードを取得します。
ListCaseEdge> caseEdges =
conditionNode.getCaseEdges();

// 同じく更新する必要があるアクション式のすべてを
保持する
// リストを作成します。テンプレートは共用されていても、
すべての
// アクションは他のアクションとは独立しているため、すべての
アクションを
// 更新する必要があります。
List<TemplateInstanceExpression> expressions =
new Vector<TemplateInstanceExpression>();

// すべての式を取得します。
for (CaseEdge edge : caseEdges)
{
  getActionExpressions("Action Value
  Template 1", edge,
  expressions);
}
```

アクション式のリストを使用して、それぞれの項目を更新できます。テンプレートで定義されたアクション式の場合、正しいパラメーター値を更新できます。

```
// それぞれの式で正しいパラメーター値を更新
します。
for (TemplateInstanceExpression expression
expressions)
{
 for (ParameterValue parameterValue :
 expression
  .getParameterValues())
 {
```

```
        // 正しいパラメーターを確認します。
        ただし、
        // このテンプレートにあるパラメーターは
         1 つだけです。
        if
         (parameterValue.getParameter().getN
         ame().equals("param0")) {
          String value =
          parameterValue.getValue();
          parameterValue.setValue("Info
          "
          +
          value.substring(value.
          indexOf(":"),
          value.length()));
       }
     }
   }
   // 条件値とアクションを更新した後、ビジネス・
   // ルール・グループを公開できます。
   // 元のリストを使用するか、ビジネス・ルール・グループの
   // 新しいリストを作成します。
   List<BusinessRuleGroup> publishList = new
   ArrayList<BusinessRuleGroup>();

   // 変更したビジネス・ルール・グループをリストに
   // 追加します。
   publishList.add(brg);

   // 更新されたビジネス・ルール・グループを含むリストを
   // 公開します。
   BusinessRuleManager.publish(publishList, true);

   out.println("");

   // ビジネス・ルール・グループを再取得し、
   // 変更内容が
   // 公開されていることを確認します。
   out.printlnBold("Decision table after
   publish:");

   brgList =
   BusinessRuleManager.getBRGsByTNSAndName(
    "http://BRSamples/com/ibm/websphere
    /sample/brules",
    QueryOperator.EQUAL,
    "ConfigurationValues",
    QueryOperator.EQUAL, 0, 0);

   brg = brgList.get(0);
   op = brg.getOperation("getMessages");
   ruleList = op.getAvailableTargets();

   decisionTable = (DecisionTable)
   ruleList.get(0);
   out.print(RuleArtifactUtility
    .printDecisionTable(decisionTable))
    ;
    }
   }
 } catch (BusinessRuleManagementException e)
 {
   e.printStackTrace();
   out.println(e.getMessage());
 }
 return out.toString();
}
```

```
/*
デシジョン・テーブルを再帰的にナビゲートして、特定の名前のテンプレート
を持ち、
変更対象となる特定のパラメーターを含むエッジ・ケースを見つけるための
メソッド。このメソッドでは、変更する値が存在しているデシジョン・テーブル
内のレベル (depth) が既知であり、現行のレベル (currentDepth) が追跡され
ていることを想定しています。*
*/
static private CaseEdge getCaseEdge(TreeNode node, String pName,
  String templateName, int depth, int currentDepth)
{
 // 現行のノードがアクションであるかどうかを確認します。これは、
 // ケース・エッジの検索中にデシジョン・テーブルのこの分岐が
 // 使い果たされたことを示します。
 if (node instanceof ActionNode)
 {
return null;
 }

 // このノードのケース・エッジを取得します。
 List<CaseEdge> caseEdges = ((ConditionNode) node).getCaseEdges();
 for (CaseEdge caseEdge : caseEdges)
 {

 // 正しいレベルに到達したかどうかを確認します。
 if (currentDepth < depth)
  {
   // 1 つ下のレベルに移動し、もう一度 getCaseEdge を呼び出して、
   // そのレベルを処理します。
   currentDepth++;
   return getCaseEdge(caseEdge.getChildNode(), pName,
    templateName, depth, currentDepth);
  } else
  {
   // 正しいレベルに到達しています。条件を取得し、その条件の
   // テンプレートが、探していたテンプレートと一致するか
   // どうかを確認します。
   TreeConditionValueDefinition condition = caseEdge
    .getValueDefinition();

   // テンプレートで定義された条件の式を
   // 取得します。
   TemplateInstanceExpression expression = condition
    .getConditionValueTemplateInstance();
   // 式からテンプレートを取得します。
      Template template = expression.getTemplate();

   // これが探していたテンプレートかどうかを確認します。
    if (template.getName().equals(templateName))
    {
     // テンプレートが一致することが分かりました。
      return caseEdge;
    } else
     caseEdge = null;
   }
 }
return null;
}

/*
このメソッドは、式の異なるパラメーター値をチェックし、
正しいパラメーターが見つかると、そのパラメーターの値を返します。
*/
private static ParameterValue getParameterValue(String pName,
  TemplateInstanceExpression expression)
{
```

```
 // 式がヌルでないことを確認します。ヌルは、渡された
 // 式がテンプレートでは定義されておらず、チェック
 // 対象のパラメーターがない可能性を意味します。
 if (expression != null) {
  // 式のパラメーター値を取得します。
  List<ParameterValue> parameterValues = expression
    .getParameterValues();

  for (ParameterValue parameterValue : parameterValues)
  {
   // それぞれのパラメーターについて、探していたパラメーター
   // と一致するかどうかを確認します。

   if
    (parameterValue.getParameter().getName().equals(pName
   ))
   {
    // 一致するパラメーター値を返します。
   }
  }
 }
return null;
}
/*
このメソッドは、特定のテンプレートで定義された
すべてのアクション式を検索します。ケース・エッジを
再帰的に検索し、式パラメーターに一致するアクション式
を追加します。
*/

private static void getActionExpressions(String templateName,
CaseEdge next, List<TemplateInstanceExpression>
expressions)
{
 ActionNode actionNode = null;
 TreeNode treeNode = next.getChildNode();

 // 現行のノードがアクション・ノード・レベルにあるかどうかを確認します。
 if (treeNode instanceof ConditionNode)
 {
  List<CaseEdge> caseEdges = ((ConditionNode) treeNode)
   .getCaseEdges();

  Iterator<CaseEdge> caseEdgesIterator =
  caseEdges.iterator();

  // すべてのケース・エッジを順に処理して、アクション式を
  // 検索します。
  while (caseEdgesIterator.hasNext())
   {
    getActionExpressions(templateName,
    caseEdgesIterator.next(),
     expressions);
   }
 } else {
   // ActionNode を検出
   actionNode = (ActionNode) treeNode;

   List<TreeAction> treeActions = actionNode.getTreeActions();
   // 式に少なくとも 1 つの treeAction が指定されている
   // ことを確認し、式を順に処理して、式が
   // 特定のテンプレートで作成されているかどうか
   // を確認します。
   if (!treeActions.isEmpty())
   {

    Iterator<TreeAction> iterator =
```

```
    treeActions.iterator();

   while (iterator.hasNext())
    {
     TreeAction treeAction = iterator.next();
     TemplateInstanceExpression expression =
     treeAction
      .getValueTemplateInstance();

     Template template = expression.getTemplate();

     if (template.getName().equals(templateName))
     {
      // テンプレートが一致する式を検出
      expressions.add(expression);
     }
    }
   }
  }
 }
}
```

### 例

例 12 の Web ブラウザー出力

**例 12 の実行**

```
Rule set before publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Decision table after publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
```

## 例 13: 条件値とアクションをデシジョン・テーブルに追加する

この例では、条件値とアクションをデシジョン・テーブルに追加します。条件値は、テンプレートを使用してデシジョン・テーブルに追加できます。

条件値を条件ノードに追加すると、実際にはケース・エッジを追加することになります。新しいケース・エッジは、ケース・エッジのリストの最後に追加されます。条件値を追加するためには、適切なパラメーター値を設定したテンプレート・インスタンス式を指定する必要があります。テンプレート・インスタンス式を指定するためには、特定のテンプレートを使用する必要があります。テンプレートの名前は、条件ノードのレベルごとに固有の名前にするのがベスト・プラクティスです。これによって、該当するタイプの条件に対応する正しいテンプレートが取得できるようになります。単一のテンプレート定義を使用すると、条件を追加するレベルの判別が困難になるおそれがあります。

条件値を条件ノードに設定すると、実際には同じテンプレート・インスタンスを持つ条件値を、同じレベルにあるすべての条件ノードに追加することになります。これは、デシジョン・テーブルが平衡型であるためです。また、新しい条件値を追加すると、この操作の一環として新しいアクション・ノードも追加されます。これらのアクション・ノードは、ユーザー表示およびテンプレート・インスタンス式にヌルの値が指定されたツリー・アクションを持ちます。条件値は、アクション・ノー

ドを子ノードとして持たない条件ノードに追加できるため、条件ノードの追加によって、アクション・ノードが大幅に増える場合があります。アクション・ノードの数は、条件ノードを追加するレベル、そのレベルにある条件ノードの数、および子レベルそれぞれにある条件ノードの数に応じます。

作成されたアクション・ノードを見つけるには、ユーザー表示およびテンプレート・インスタンス式がヌルのツリー・アクションを持つアクション・ノードの検索を実行します。 TreeAction に設定可能な式を作成するには、TreeActionValueTemplate を使用します。このパターンを、すべての新しいアクション・ノードに対して繰り返す必要があります。

この例には、新規ツリー・アクションのセットアップを支援する 2 つのメソッドが提供されています。 getEmptyActionNode は、現行の条件ノードから空のアクション・ノードを再帰的に検索します。getParameterValue は、名前で指定されたパラメーターの値を返します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionTermDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionValueTemplate;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueTemplate;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;

public class Example13
{
 static Formatter out = new Formatter();

 static public String executeExample13()
 {
  try
  {
   out.clear();

   // ターゲット名前空間と名前によってビジネス・ルール・グループ
   // を取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
    .getBRGsByTNSAndName(
   "http://BRSamples/com/ibm/websphere/sample/brules",
     QueryOperator.EQUAL,"ConfigurationValues",
     QueryOperator.EQUAL, 0, 0);
```

```
if (brgList.size() > 0)
 {
  // リストから最初のビジネス・ルール・グループを
  // 取得します。ターゲット名前空間と名前の
  // 組み合わせは固有であることから、これがリスト内
  // の唯一のビジネス・ルール・グループとなるはずです。
  BusinessRuleGroup brg = brgList.get(0);

  // 変更するビジネス・ルールが関連付けられている
  // 操作をビジネス・ルール・グループから取得します。
  // 特定の操作にはビジネス・ルールが関連付けられて
  // います。
  Operation op = brg.getOperation("getMessages");

  // この操作に使用可能なすべての
  // ビジネス・ルールを取得します。
  List<BusinessRule> ruleList =
   op.getAvailableTargets();

  // この操作には、1 つのビジネス・ルール
  // しかありません。これが、更新対象の
  // ビジネス・ルールです。

  DecisionTable decisionTable = (DecisionTable)
   ruleList.get(0);
   out.printlnBold("Decision table before
   publish:");
   out.print(RuleArtifactUtility
   .printDecisionTable(decisionTable));
```

条件値を追加するレベルを見つける必要があります。クラスを使用するユーザー・インターフェースまたはアプリケーションは、どこに条件を追加するかを認識しているため、このレベルは通常、パラメーターとして渡されます。

```
// デシジョン・テーブルのすべての条件と
// アクションが含まれるツリー・ブロックを
// 取得します。
TreeBlock treeBlock =
 decisionTable.getTreeBlock();

// ツリー・ブロックから、デシジョン・テーブルの
// ナビゲーションの開始点となるツリー・ノードを
// 取得します。
ConditionNode conditionNode = (ConditionNode)
 treeBlock.getRootNode();

// 条件の第 1 レベルであるこのノードの
// ケース・エッジを取得します。
List<CaseEdge> caseEdges =
 conditionNode.getCaseEdges();

// 新しい条件の追加先となるケース・エッジを
// 取得します。
CaseEdge caseEdge = caseEdges.get(0);

// 条件のテンプレートを取得するために、
// ケース・エッジに対応する条件ノードを
// 取得します。
conditionNode = (ConditionNode)
    caseEdge.getChildNode();

// 条件のテンプレートを取得します。
List<TreeConditionValueTemplate>
 treeValueConditionTemplates = conditionNode
 .getAvailableValueTemplates();
```

```
Iterator<TreeConditionValueTemplate>
 treeValueConditionTemplateIterator =
 treeValueConditionTemplates.iterator();

TreeConditionValueTemplate conditionTemplate =
 null;
```

デシジョン・テーブルの各条件ノード・レベルで固有のテンプレート名を使用することで、より簡単に正しい条件ノード値に条件値を追加できます。

```
// 使用すべきテンプレートを検索します。
 while
(treeValueConditionTemplateIterator.hasNext())
{
 conditionTemplate =
  treeValueConditionTemplateIterator
   .next();
 if (conditionTemplate.getName().equals(
  "Condition Value Template
  2.1"))
  {
   // テンプレートを検出
   break;
  }
  conditionTemplate = null;
}
if (conditionTemplate != null)
{
```

正しいテンプレートが検出された後、インスタンスを作成して適切なパラメーター値を設定してから、条件ノードに追加します。

```
// テンプレートからパラメーター定義を
// 取得します。
Parameter conditionParameter =
conditionTemplate.getParameter("param0");

// 新しい条件テンプレート・インスタンスで
// 使用するパラメーター値インスタンスを
// 作成します。
ParameterValue conditionParameterValue =
  conditionParameter
  .createParameterValue("fatal");

List<ParameterValue>
  conditionParameterValues = new
  ArrayList<ParameterValue>();

// パラメーター値をリストに追加します。

conditionParameterValues
 .add(conditionParameterValue);

// パラメーター値を使用して、新しい条件
// テンプレート・インスタンスを作成します。
TemplateInstanceExpression
 newConditionValue =
 conditionTemplate
  .createTemplateInstanceExpression(c
 onditionParameterValues);
// 条件テンプレート・インスタンスをこの条件
// ノードに追加します。
conditionNode

.addConditionValueToThisLevel(newConditionValue);
// 条件ノードが追加されると、新しい
```

```
// 空のアクション・ノードが作成され
// ます。ここにはアクション・テンプレート・
// インスタンスを設定する必要があります。親
// レベルから、それぞれの空のアクション・ノード
// を検索すると、新しい空のアクション・
// ノードをすべて見つけることができます。
conditionNode = (ConditionNode)
conditionNode.getParentNode();
```

条件値を条件ノードに追加した後は、TreeActionValueTemplate を使用して、新規アクション・ノードのツリー・アクションを設定する必要があります。まず、ケース・エッジの空のアクション・ノードを見つけます。親条件ノードを使用することで、条件ノードの繰り返し処理によってすべてのアクション・ノードが確実に検索されるようにします。

```
// 親ノードのケース・エッジを取得します。
caseEdges = conditionNode.getCaseEdges();

Iterator<CaseEdge> caseEdgesIterator =
caseEdges.iterator();

while (caseEdgesIterator.hasNext())
{
 // 各ケース・エッジについて、空のアクション・
 // ノードがある場合にはそれを取得します。
 ActionNode actionNode =
 getEmptyActionNode(caseEdgesIterator
 .next());

 // すべてのアクションが設定されていることを確認します。
 if (actionNode != null)
 {
```

ツリー・アクションが空のアクション・ノードが検出された場合には、TreeActionValueTemplate を使用してツリー・アクションを設定する必要があります。まずテンプレートを見つけて、パラメーターを指定してからテンプレート・インスタンスを作成します。テンプレート・インスタンスが作成されたら、ツリー・アクションを更新できます。この例では、同じ条件ノードの下にある別のアクション・ノードの別のツリー・アクションの値を使用して、パラメーターを設定しています。新規パラメーター値の作成に使用できる値が別のツリー・アクションに存在しない別のデシジョン・テーブルの場合は、アプリケーションによって値をパラメーターとして渡す必要があります。

```
  // ツリー・アクションのリストを
  // 取得します。これらの
  // アクションは、実際のアクション
  // ではなく、アクション
  // のプレースホルダー
  // です。
  List<TreeAction>
  treeActionList = actionNode
   .getTreeActions();

  List<TreeActionTermDefinition>
  treeActionTermDefinitions =
   treeBlock
  .getTreeActionTermDefinitions();

  List<TreeActionValueTemplate>
   treeActionValueTemplates =
   treeActionTermDefinitions
   .get(0).getValueTemplates();
```

```
TreeActionValueTemplate
 actionTemplate = null;

for (TreeActionValueTemplate
  tempActionTemplate :
  treeActionValueTemplates)
  {

  if
  (tempActionTemplate.get
   Name().equals(
   "Action Value
   Template 1"))
   {
    actionTemplate =
    tempActionTemplate;
    break;
   }
  }

  if (actionTemplate != null)
  {
   // 親条件ノードの下に
   // ある別のアクションを
   // 取得します。
   // この値は、新しい
   // アクション・ノード
   // のエラー・メッセージ
   // を作成するときに、
   // ベースとして使用
   // します。
   // まず初めに、
   // 親条件ノードまで
   // 移動します。
   ConditionNode
   parentNode =
   (ConditionNode)
   actionNode
    .getParentNode();

   // 親ノードの最初の
   // ケース・エッジを
   // 取得します。新規
   // アクションはケース・
   // エッジ・リストの
   // 最後に追加される
   // ため、最初のケース・
   // エッジのアクション
   // は常に設定済みです。
   CaseEdge caseE =
    parentNode.getCas
    eEdges().get(
    0);

   // この子ノードは
   // アクション・ノードで、
   // 新規アクション・
   // ノードと同じレベルに
   // あります。
   ActionNode aNode =
   (ActionNode) caseE
    .getChildNode();

   // ツリー・アクションのリストを
   // 取得します。
   TreeAction
```

```
        existingTreeAction =
        aNode
        .getTreeActions()
        .get(0);

       // パラメーターを
       // 取得することが
       // 可能なツリー・
       // アクションの
       // テンプレート・
       // インスタンス式を
       // 取得します。

       TemplateInstanceExpression
       existingExpression =
        existingTreeAction
        .getValueTemplateInstance();

       ParameterValue
       existingParameterValue =
       getParameterValue(
         "param0",
       existingExpression);

       String actionValue =
       existingParameterValue
        .getValue();

       // 既存のツリー・
       // アクションの
       // メッセージから
       // 新規メッセージを
       // 作成します。
       actionValue = "Fatal"
         +
       actionValue.substring(actionValue
        .indexOf(":"), actionValue
        .length());
        Parameter
        actionParameter =
        actionTemplate
        .getParameter("param0");

        // テンプレートから
        // パラメーターを取得します。
        ParameterValue
        actionParameterValue =
         actionParameter
        .createParameterValue(actionValue);

        // パラメーターをテンプレートの
        // リストに追加します。
        List<ParameterValue>
        actionParameterValues = new
        ArrayList<ParameterValue>();

      actionParameterValues.add(actionParameterValue);

        // 新規ツリー・アクション・
        // インスタンスを作成します。

       TemplateInstanceExpression
        treeAction = actionTemplate
.createTemplateInstanceExpression(actionParameterValues);
```

```
            // ツリー・アクションをツリー・
            // アクション・リストに設定すること
            // によって、アクション・ノードに
            // 設定します。
```

ここで、アクション・ノード内のツリー・アクションが更新されます。

```
            treeActionList.get(0)
            .setValueTemplateInstance(
            treeAction);
            }
           }
         }
      }
      // 条件値とアクションを
      // 更新した後、ビジネス・ルール・
      // グループを公開できます。
      // 元のリストを使用するか、ビジネス・ルール・グループの
      // 新しいリストを作成します。
       List<BusinessRuleGroup> publishList = new
        ArrayList<BusinessRuleGroup>();

      // 変更したビジネス・ルール・グループを
      // リストに追加します。
       publishList.add(brg);

      // 更新されたビジネス・ルール・グループを含む
      // リストを公開します。

      BusinessRuleManager.publish(publishList, true);

      brgList =
       BusinessRuleManager.getBRGsByTNSAndName(
       "http://BRSamples/com/ibm/websphere/sample/brules",
       QueryOperator.EQUAL, "ConfigurationValues",
       QueryOperator.EQUAL, 0, 0);
       brg = brgList.get(0);
       op = brg.getOperation("getMessages");
       ruleList = op.getAvailableTargets();
       decisionTable = (DecisionTable)
        ruleList.get(0);
       out.println1nBold("Decision table after
        publish:");
       out
        .print(RuleArtifactUtility
         .printDecisionTable(decisionTable));
      }
    } catch (ValidationException e)
    {
     List<Problem> problems = e.getProblems();

     out.println("Problem = " +
     problems.get(0).getErrorType().name());

     e.printStackTrace();
     out.println(e.getMessage());
    } catch (BusinessRuleManagementException e)
    {
     e.printStackTrace();
     out.println(e.getMessage());
    }
    return out.toString();
   }

   /*
   * このメソッドは、現行のケース・エッジから、空のツリー・
   * アクションを持つすべてのアクション・ノードを検索します。
```

```
* 空のアクション・ノードを検出するには、ケース・エッジ・
* リストの最後を調べ、アクション・ノードにユーザー表示と
* TemplateInstanceExpression の両方がヌルのツリー・アクション
* があるかどうかを確認します。
*/
private static ActionNode getEmptyActionNode(CaseEdge next)
{
 ActionNode actionNode = null;
 TreeNode treeNode = next.getChildNode();

if (treeNode instanceof ConditionNode)
{
 List<CaseEdge> caseEdges = ((ConditionNode) treeNode)
  .getCaseEdges();

 if (caseEdges.size() > 1)
 {
  // 右端のケース・エッジを新規条件として取得
  // します。その結果、空のアクションはケース・
  // エッジの右端に位置することになります。
  actionNode = getEmptyActionNode(caseEdges
   .get(caseEdges.size() - 1));

  if (actionNode != null)
  {
   return actionNode;
  }
}
} else
{
 actionNode = (ActionNode) treeNode;

 List<TreeAction> treeActions =
 actionNode.getTreeActions();

 if (!treeActions.isEmpty())
 {
 if
 ((treeActions.get(0).getValueUserPresentation() == null)
  &&
 (treeActions.get(0).getValueTemplateInstance() == null))
  {
   return actionNode;
  }
 }
  `actionNode = null;
 }
 return actionNode;
}
/*
* このメソッドは、式の異なるパラメーター値をチェックし、
* 正しいパラメーターが見つかると、そのパラメーターの値を
* 返します。
*/
private static ParameterValue getParameterValue(String pName,
  TemplateInstanceExpression expression)
{
 ParameterValue parameterValue = null;

 // 式がヌルでないことを確認します。
 // ヌルは、渡された式がテンプレート
 // では定義されておらず、チェック対象の
 // パラメーターがない可能性を意味します。
 if (expression != null)
 {
  // 式のパラメーター値を取得します。
  List<ParameterValue> parameterValues = expression
```

```
    .getParameterValues();
   Iterator<ParameterValue> parameterIterator =
    parameterValues
    .iterator();

   // それぞれのパラメーターについて、探していた
   // パラメーターと一致するかどうかを確認します。
   while (parameterIterator.hasNext())
   {
    parameterValue = parameterIterator.next();

    if
    (parameterValue.getParameter().getName().equals(pName))
    {
    // 一致するパラメーター値を
    // 返します。
    return parameterValue;
    }
  }
 }
 return parameterValue;
}
}
```

### 例

例 13 の Web ブラウザー出力

**例 13 の実行**

```
Decision table before publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules

Decision table after publish:
Decision Table
Name: getMessages
Namespace: http://BRSamples/com/ibm/websphere/sample/brules
```

## 例 14: ルール・セットでのエラーを処理する

この例では、適切なメッセージを表示したり、問題を修正するためのアクションを実行できるように、ルール・セットの問題をキャッチし、発生した問題を突き止める方法に重点を置きます。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import
com.ibm.wbiserver.brules.mgmt.problem.ProblemStartDateAfterEndDate;
import com.ibm.wbiserver.brules.mgmt.problem.ValidationError;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
```

```
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example14 {
static Formatter out = new Formatter();

static public String executeExample14() {
 try {
   out.clear();

   // ターゲット名前空間と名前によってビジネス・ルール・グループ
   を取得します。
   List<BusinessRuleGroup> brgList = BusinessRuleManager
     .getBRGsByTNSAndName(
     "http://BRSamples/com/ibm/websphere
     /sample/brules",
     QueryOperator.EQUAL,
     "ApprovalValues",
     QueryOperator.EQUAL, 0, 0);

   if (brgList.size() > 0) {
     // リストから最初のビジネス・ルール・グループを取得します。
     // ターゲット名前空間と名前の組み合わせは固有で
     あることから、
     // これがリスト内の唯一のビジネス・ルール・グループと
     なるはずです。
     BusinessRuleGroup brg = brgList.get(0);
     out.println("Business Rule Group retrieved");

     // 変更するビジネス・ルールが関連付けられている
     // 操作をビジネス・ルール・グループから取得します。
     // 特定の操作にはビジネス・ルールが関連付けられて
     // います。
     Operation op = brg.getOperation("getApprover");

     // 特定のルールを名前を基準に取得します。
     List<BusinessRule> ruleList =
     op.getBusinessRulesByName(
       "getApprover", QueryOperator.EQUAL, 0,
       0);

     // 特定のルールを取得します。
     RuleSet ruleSet = (RuleSet) ruleList.get(0);
     out.println("Rule Set retrieved");

     RuleBlock ruleBlock = ruleSet.getFirstRuleBlock();

     Iterator<RuleSetRule> ruleIterator =
     ruleBlock.iterator();

     // ルールを検索して、変更するルールを
     見つけます。
     while (ruleIterator.hasNext()) {
       RuleSetRule rule = ruleIterator.next();

       // ルールがテンプレートで定義されていて、変更可能
       で
       // あることを確認します。
       if (rule instanceof
       RuleSetTemplateInstanceRule) {
        // テンプレート・ルール・インスタンスを取得します。
        RuleSetTemplateInstanceRule
        templateInstance =
        (RuleSetTemplateInstanceRule) rule;
```

```
        // 正しいテンプレート・ルール・インスタンスであることを
        確認します。
        if (templateInstance.getName().equals(
          "LargeOrderApprover")) {
```

問題を発生させるため、この例では、パラメーターを式とは互換しない値に設定します。パラメーターは整数が想定されていますが、ストリングを渡します。

```
          // テンプレート・インスタンスからパラメーターを
          取得します。
          ParameterValue parameter =
          templateInstance
            .getParameterValue("par
            am1");

          // このパラメーターに誤った値を設定
          します。
          // これにより、検証エラーが発生することに
          なります。
          parameter.setValue("$3500");
          out.println("Incorrect parameter
          value set");
          break;
        }
      }
    }
    // 上記で発生したエラーにより、
    この
    // コードには到達できなく
    // なります。

    // 条件値とアクションを更新した後、
    // ビジネス・ルール・グループを
    // 公開できます。
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);

    // 更新されたビジネス・ルール・グループを含むリストを
     公開します。
    BusinessRuleManager.publish(publishList, true);
  }
```

ValidationException をキャッチし、この例外から、問題を取得することができます。それぞれの問題について、エラーを確認することで、発生したエラーを判別できます。メッセージをプリントするか、または適切なアクションを実行することができます。

```
  } catch (ValidationException e) {
  out.println("Validation Error");

  List<Problem> problems = e.getProblems();

  Iterator<Problem> problemIterator = problems.iterator();

  // 該当するエラーの問題リストを確認し、適切な
  // アクション (エラーのレポート、エラーの修正など)
  // を実行します。
  while (problemIterator.hasNext()) {
    Problem problem = problemIterator.next();
    ValidationError error = problem.getErrorType();
```

```
    // 特定のエラーの値を確認します。
    if (error == ValidationError.TYPE_CONVERSION_ERROR) {
      // 問題をレポートすることによって、このエラーを処理します。
      out
        .println("Problem: Incorrect value
        entered for a parameter");
      return out.toString();
    }
    // または....
    // 他のエラーを確認して、適切なエラー・
    // メッセージを表示するか、アクションを実行して
    // 問題を修正することもできます。
  }
} catch (BusinessRuleManagementException e) {
  out.println("Error occurred.");
  e.printStackTrace();
}
return out.toString();
}
}
```

### 例

例 14 の Web ブラウザー出力

**例 14 の実行**

```
Business Rule Group retrieved
Rule Set retrieved
Validation Error
Problem: Incorrect value entered for a parameter
```

## 例 15: ビジネス・ルール・グループでのエラーを処理する

この例は、例 14 と同様で、ビジネス・ルール・グループを公開するときに発生した問題を処理する方法を説明します。この例では、問題を判別し、正しいメッセージをプリントするか、またはアクションを実行する方法を説明します。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleGroup;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManagementException;
import com.ibm.wbiserver.brules.mgmt.BusinessRuleManager;
import com.ibm.wbiserver.brules.mgmt.Operation;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecord;
import com.ibm.wbiserver.brules.mgmt.OperationSelectionRecordList;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.ValidationException;
import com.ibm.wbiserver.brules.mgmt.problem.Problem;
import
com.ibm.wbiserver.brules.mgmt.problem.ProblemStartDateAfterEndDate;
import com.ibm.wbiserver.brules.mgmt.query.QueryOperator;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class Example15
{
```

```
static Formatter out = new Formatter();

static public String executeExample15()
{
 try
 {
  out.clear();

 // ターゲット名前空間と名前によってビジネス・ルール・グループ
 を取得します。
 List<BusinessRuleGroup> brgList = BusinessRuleManager
  .getBRGsByTNSAndName(
  "http://BRSamples/com/ibm/websphere
  /sample/brules",
  QueryOperator.EQUAL,
  "ApprovalValues",
  QueryOperator.EQUAL, 0, 0);
  if (brgList.size() > 0)
  {
   // リストから最初のビジネス・ルール・グループを取得します。
   // ターゲット名前空間と名前の組み合わせは固有で
   あることから、
   // これがリスト内の唯一のビジネス・ルール・グループと
   なるはずです。
   BusinessRuleGroup brg = brgList.get(0);
   out.println("Business Rule Group retrieved");

   // 変更するビジネス・ルールが関連付けられている
   // 操作をビジネス・ルール・グループから取得します。
   // 特定の操作にはビジネス・ルールが関連付けられて
   // います。
   Operation op = brg.getOperation("getApprover");

   // 特定のルールを名前を基準に取得します。
   List<BusinessRule> ruleList =
   op.getBusinessRulesByName(
    "getApprover", QueryOperator.EQUAL, 0,
    0);

   // 特定のルールを取得します。
   RuleSet ruleSet = (RuleSet) ruleList.get(0);
    out.println("Rule Set retrieved");

   RuleBlock ruleBlock = ruleSet.getFirstRuleBlock();

   Iterator<RuleSetRule> ruleIterator =
    ruleBlock.iterator();

   // ルールを検索して、変更するルールを
   見つけます。
   while (ruleIterator.hasNext())
   {
    RuleSetRule rule = ruleIterator.next();

    // ルールがテンプレートで定義されていて、変更可能
    で
    // あることを確認します。
    if (rule instanceof
    RuleSetTemplateInstanceRule)
    {
     // テンプレート・ルール・インスタンスを取得します。
     RuleSetTemplateInstanceRule
     templateInstance =
     (RuleSetTemplateInstanceRule) rule;

     // 正しいテンプレート・ルール・インスタンスであることを
     確認します。
```

```
    if (templateInstance.getName().equals(
      "LargeOrderApprover"))
    {
     // テンプレート・インスタンスからパラメーターを
     取得します。
     ParameterValue parameter =
     templateInstance
      .getParameterValue("par
      am1");

     // このパラメーターの値を設定します。
     // この値は正しいフォーマットである
     ため、
     // 検証エラーは発生しません。
     parameter.setValue("4000");
     out.println("Rule set parameter
     value on set correctly");
     break;
     }
    }
  }
```

ルール・セットが正しいことを確認するために、検証メソッドを呼び出すことができます。検証メソッドはすべてのオブジェクトで使用することが可能で、このメソッドが返す問題のリストを検査して問題を判別できます。オブジェクトで検証メソッドを呼び出すと、そのオブジェクトに含まれるすべてのオブジェクトでも検証メソッドが呼び出されます。

```
     // ルール・セットの変更を検証します。
     List<Problem> problems = ruleSet.validate();
     out.println("Rule set validated");

     // このテスト・ケースではエラーが発生しないはず
     // ですが、
     // ベスト・プラクティスとして、問題があるかどうかを確認し、
     // 修復のためのアクションを実行するか、またはエラーを
     // レポートします。
     if (problems != null)
     {
      Iterator<Problem> problemIterator =
      problems.iterator();

      while (problemIterator.hasNext())
      {
       Problem problem = problemIterator.next();

       if (problem instanceof
       ProblemStartDateAfterEndDate)
       {
        out
        .println("Incorrect
         value entered for a
          parameter");
        return out.toString();
       }
      }
     } else
     {
      out.println("No problems found for the rule
      set");
     }
     // 有効なルール・ターゲットのリストを取得します。
     List<BusinessRule> ruleList2 =
     op.getAvailableTargets();

     // 誤ってスケジュールされることになる最初のルールを
```

```
    // 取得します。
    BusinessRule rule = ruleList2.get(0);

    // スケジュールされたルールの終了時刻が
    // 開始時刻の
    // 1 時間前になるようにエラー条件を設定します。
    // これにより、検証エラーが発生することになります。
    Date future = new Date();
    long futureTime = future.getTime() - 360000;

    // 誤ってスケジュールされた項目を追加するために、
    // 操作選択リストを取得します。
    OperationSelectionRecordList opList = op
    .getOperationSelectionRecordList();

    // スケジュールされたルールの新しいインスタンスを作成します。
    // 検証が行われるか、変更が追加されて公開が行われるまで、
    // エラーはスローされません。
    OperationSelectionRecord newRecord = opList
     .newOperationSelectionRecord(new Date(),
     new Date(
     futureTime), rule);
```

誤った日付のセットを指定したレコードを追加しても、それによってエラーが発生することはありません。変更が行われている間は、オーバーラップが発生したり、選択レコードが設定されていない操作が存在したりする場合があるためです。エラーが検出されるのは、操作選択レコードを持つビジネス・ルール・グループが公開されるときです。オブジェクトが公開される前に検証メソッドが呼び出され、エラーが存在する場合には例外がスローされます。

```
    // スケジュールされたルールのインスタンスを操作に追加します。
    // ここでもエラーは発生しません。
    opList.addOperationSelectionRecord(newRecord);
    out.println("New selection record added with
    incorrect schedule");

    // 条件値とアクションを更新した後、
    // ビジネス・ルール・グループを
    // 公開できます。
    // 元のリストを使用するか、ビジネス・ルール・グループの
    // 新しいリストを作成します。
    List<BusinessRuleGroup> publishList = new
    ArrayList<BusinessRuleGroup>();

    // 変更したビジネス・ルール・グループをリストに追加します。
    publishList.add(brg);

    // 更新されたビジネス・ルール・グループを含むリストを
    // 公開します。
    BusinessRuleManager.publish(publishList, true);
   }
  } catch (ValidationException e) {
  out.println("Validation Error");

  List<Problem> problems = e.getProblems();

  Iterator<Problem> problemIterator = problems.iterator();
  // 複数の問題が存在する可能性があります。
  // 問題を順に検索して 1 つずつ処理するか、問題を
  // レポートします。
  while (problemIterator.hasNext())
  {
   Problem problem = problemIterator.next();

   // それぞれの問題のタイプは異なっており、比較することが
```

```
    // 可能です。
    if (problem instanceof ProblemStartDateAfterEndDate)
    {
     out
     .println("Rule schedule is
      incorrect. Start date is after end
      date.");
     return out.toString();
     }
    // または....
    // 他のエラーを確認して、適切なエラー・
    // メッセージを表示するか、アクションを実行して
    // 問題を修正することもできます。
  }
 }catch (BusinessRuleManagementException e)
 {
  out.println("Error occurred.");
  e.printStackTrace();
 }
 return out.toString();
 }
}
```

### 例

例 15 の Web ブラウザー出力

**例 15 の実行**

```
Business Rule Group retrieved
Rule Set retrieved
Rule set parameter value on set correctly
Rule set validated
Validation Error
Rule schedule is incorrect. Start date is after end date.
```

# 付録

付録には、共通操作を単純化するために例の中で使用した追加クラスと、ワイルドカードを使用してビジネス・ルール・グループを検索する複雑な照会を作成する追加の例を記載します。

## Formatter クラス

このクラスは、さまざまな例を表示するための各種のメソッドを提供し、各種の HTML タグを追加して出力フォーマットを設定します。

```
package com.ibm.websphere.sample.brules.mgmt;

public class Formatter {

private StringBuffer buffer;

public Formatter()
{
  buffer = new StringBuffer();
}

public void println(Object o)
{
  buffer.append(o);
  buffer.append("<br>¥n");
  }
```

```
  public void print(Object o)
  {
    buffer.append(o);
  }

  public void printlnBold(Object o)
  {
    buffer.append("<b>");
    buffer.append(o);
    buffer.append("</b><brbr>¥n");
  }

  public void printBold(Object o)
  {
    buffer.append("<b>");
    buffer.append(o);
    buffer.append("</b>");
  }

  public String toString()
  {
    return buffer.toString();
  }

  public void clear()
  {
    buffer = new StringBuffer();
  }
}
```

## RuleArtifactUtility クラス

このユーティリティー・クラスには 2 つの public メソッドがあります。最初の public メソッドは、デシジョン・テーブルを出力するためのメソッドです。このメソッドは、再帰を使用する private メソッドを使用して、デシジョン・テーブルの条件とアクションを出力します。2 番目の public メソッドは、ルール・セットを出力するためのメソッドです。

```
package com.ibm.websphere.sample.brules.mgmt;

import java.util.Iterator;
import java.util.List;

import com.ibm.wbiserver.brules.mgmt.BusinessRule;
import com.ibm.wbiserver.brules.mgmt.Parameter;
import com.ibm.wbiserver.brules.mgmt.ParameterValue;
import com.ibm.wbiserver.brules.mgmt.RuleTemplate;
import com.ibm.wbiserver.brules.mgmt.Template;
import com.ibm.wbiserver.brules.mgmt.dtable.ActionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.CaseEdge;
import com.ibm.wbiserver.brules.mgmt.dtable.ConditionNode;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTable;
import com.ibm.wbiserver.brules.mgmt.dtable.DecisionTableRule;
import
com.ibm.wbiserver.brules.mgmt.dtable.DecisionTableTemplateInstanceRule;
import com.ibm.wbiserver.brules.mgmt.dtable.TemplateInstanceExpression;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeAction;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeActionTermDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeBlock;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionTermDefinition;
import
com.ibm.wbiserver.brules.mgmt.dtable.TreeConditionValueDefinition;
import com.ibm.wbiserver.brules.mgmt.dtable.TreeNode;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleBlock;
```

```
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSet;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRule;
import com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetRuleTemplate;
import
com.ibm.wbiserver.brules.mgmt.ruleset.RuleSetTemplateInstanceRule;

public class RuleArtifactUtility
{
  static Formatter out = new Formatter();

  /*
  デシジョン・テーブルを印刷するためのメソッド (条件とアクションを
  HTML の表形式で印刷)。デシジョン・テーブルの
  ケース・エッジを再帰的に処理する個別のメソッドにより、
  条件とアクションを印刷する。
  */

  public static String printDecisionTable(BusinessRule
  ruleArtifact)
  {
    out.clear();
    out.printlnBold("Decision Table");
    DecisionTable decisionTable = (DecisionTable)
    ruleArtifact;
    out.println("Name: " +
    decisionTable.getName());
    out.println("Namespace: " +
    decisionTable.getTargetNameSpace());

    // 条件とアクションのテーブルを処理する前に、
    // デシジョン・テーブルの初期ルールを印刷する
    DecisionTableRule initRule =
    decisionTable.getInitRule();
    if (initRule != null)
    {
      out.printBold("Init Rule: ");
      out.println(initRule.getName());
      out.println("Display Name: " +
      initRule.getDisplayName());
      out.println("Description: " +
      initRule.getDescription());
      // ユーザーの拡張プレゼンテーションにより、パラメーターの値が
      // プレゼンテーションに対して自動的に設定され、テンプレートに
      // 初期ルールが定義されていた場合は、ユーザーの拡張プレゼンテーションが
            // 拡張プレゼンテーションは通常のプレゼンテーションと同じになる。
      out.println("Extended User
      Presentation: "
        +
      initRule.getExpandedUse
      rPresentation());
      // 通常のユーザー・プレゼンテーションにはストリング内に
      // プレースホルダーが設定されるため、テンプレートに
      // 置き換えることができる。
      // テンプレートに初期ルールが定義されていない場合、
      // ユーザー・プレゼンテーションはプレースホルダーのない
      // 単純なストリングになる。
      // プレースホルダーは、[n] の形式で表記される。
      // この n は、テンプレート内のパラメーターの索引値 (0 基準)
      // を表す。
      // この値を使用して、編集可能なパラメーターが設定されている
      // フィールドを編集するためのインターフェースを
      // 作成することができる。
      out.println("User Presentation: " +
      initRule.getUserPresentation());
      // テンプレートの有無にかかわらず、初期ルールが定義されて
      // いる可能性がある。パラメーターにアクセスする前に、
      // テンプレートが使用されていたかどうかを確認する。
```

```
      if (initRule instanceof
      DecisionTableTemplateInstanceRule)
      {
        DecisionTableTemplateIn
        stanceRule
        templateInstance =
        (DecisionTableTemplateI
        nstanceRule) initRule;

        RuleTemplate template =
        templateInstance.getRul
        eTemplate();

        List<Parameter>
        parameters =
        template.getParameters(
        );
        Iterator<Parameter>
        paramIterator =
        parameters.iterator();

        Parameter parameter =
        null;

        while
        (paramIterator.hasNext(
        )) {
        parameter =
        paramIterator.next();

         out.println("Parameter
        Name: " +
        parameter.getName());
        out.println("Parameter
        Value: "
        +
        templateInstance.getPar
        ameterValue(parameter
        .getName()));
        }
      }
    }
    // デシジョン・テーブルの残りの部分については root から
    // 処理を開始し、異なるケース・エッジとアクションを
    // 再帰的に処理する
    TreeBlock treeBlock =
    decisionTable.getTreeBlock();
    TreeNode treeNode = treeBlock.getRootNode();

    printDecisionTableConditionsAndActions(treeNode
    , 0);
    out.println("");
    return out.toString();
  }
  /*ケース・エッジを再帰的に処理して
  条件とアクションを印刷するメソッド
*/
  static private void printDecisionTableConditionsAndActions(
    TreeNode treeNode, int indent)
  {
    out.print("<table border=¥"1¥">");
    if (treeNode instanceof ConditionNode)
    {
      // 現在の TreeNode に対するケース・エッジを取得し、
      // 各ケース・エッジについて条件を印刷する
      ConditionNode conditionNode =
      (ConditionNode) treeNode;
```

```
List<CaseEdge> caseEdges =
conditionNode.getCaseEdges();
Iterator<CaseEdge> caseEdgeIterator
= caseEdges.iterator();

CaseEdge caseEdge = null;

while (caseEdgeIterator.hasNext())
{
  out.print("<tr>");
  // これが条件ノードの条件の開始部分である場合は、
  // 条件の内容を印刷する
  if (indent == 0)
  {
  out.print("<td>");

  TreeConditionTermDefinition
  termDefinition =
  conditionNode
  .getTermDefinition();

  out.print(termDefinitio
  n.getUserPresentation()
  );
  out.print("</td>");
  indent++;
  } else {
  // ケース・エッジの条件内容を印刷したら、
  // 残りのケース・エッジについてはスキップする
  out.print("<td></td>");
  }

  caseEdge =
  caseEdgeIterator.next()
  ;

  out.print("<td>");

  // caseEdge がテンプレートで定義されているかを確認
  if
  (caseEdge.getValueDefin
  ition() != null)
  {
  TemplateInstanceExpress
  ion templateInstance =
  caseEdge
  .getValueTemplateInstan
  ce();

  out.println(templateIns
  tance.getExpandedUserPr
  esentation());

  TreeConditionValueDefin
  ition valueDef =
  caseEdge
  .getValueDefinition();

  out.println(valueDef.ge
  tUserPresentation());

  Template template =
  templateInstance.getTem
  plate();

  // テンプレート定義のパラメーターを取得し、
```

```
// パラメーターの名前と値を印刷する
List<Parameter>
parameters =
template.getParameters(
);
Iterator<Parameter>
paramIterator =
parameters.iterator();

List<ParameterValue>
parameterValues =
templateInstance
.getParameterValues();
Iterator<ParameterValue
> paramValues =
parameterValues
.iterator();

Parameter parameter =
null;
ParameterValue
parameterValue = null;

while
(paramIterator.hasNext(
) &&
paramValues.hasNext())
{
parameter =
paramIterator.next();
parameterValue =
paramValues.next();

out.println("Parameter
Name: " +
parameter.getName());
out.println("Parameter
Value: "
  +
parameterValue.getValue
());
}
}

out.print("</td><td>");
// caseEdge の子ノードを印刷
printDecisionTableCondi
tionsAndActions(caseEdg
e.getChildNode(),
0);

out.print("</td></tr>")
;
}


// 存在する場合は Otherwise 条件を追加
TreeNode otherwise =
conditionNode.getOtherwiseCase();

if (otherwise != null)
{
  out.print("<tr><td></td>
  <td>Otherwise</td><td>
  ");
  // Otherwise ConditionNode を印刷
  printDecisionTableCondi
```

```
    tionsAndActions(otherwi
    se, 0);
    out.print("</td></td>")
    ;
  }
  out.print("</table>");
} else {
  // ActionNode が見つかり、TreeActions を印刷するための
  // 追加ロジックが必要な場合
  ActionNode actionNode =
  (ActionNode) treeNode;
  List<TreeAction> treeActions =
  actionNode.getTreeActions();

  Iterator<TreeAction>
  treeActionIterator =
  treeActions.iterator();

  TreeAction treeAction = null;

  // 印刷対象の複数の TreeActions を
  // ActionNode に格納
  while
  (treeActionIterator.hasNext())
  {
    out.print("<tr>");
    treeAction =
    treeActionIterator.next
    ();

    TreeActionTermDefinitio
    n treeActionTerm =
    treeAction
    .getTermDefinition();

    if (indent == 0) {
    out.print("<td>");
    out.print(treeActionTer
    m.getUserPresentation()
    );
    out.print("</td>");
    }
    out.print("<td>");
    TemplateInstanceExpress
    ion templateInstance =
    treeAction
    .getValueTemplateInstan
    ce();

    // パラメーターの名前と値を処理する前に、
    // TreeAction に対してテンプレートが指定
    // されていたかどうかを確認する
    if (templateInstance !=
    null) {
    out.println(templateIns
    tance.getExpandedUserPr
    esentation());

    Template template =
    templateInstance.getTem
    plate();

    List<Parameter>
    parameters =
    template.getParameters(
    );
```

```
        Iterator<Parameter>
        paramIterator =
        parameters.iterator();

        List<ParameterValue>
        parameterValues =
        templateInstance
        .getParameterValues();
        Iterator<ParameterValue
        > paramValues =
        parameterValues
        .iterator();

        Parameter parameter =
        null;
        ParameterValue
        parameterValue = null;

        while
        (paramIterator.hasNext(
        ) &&
        paramValues.hasNext())
        {
        {parameter =
        paramIterator.next();
        parameterValue =
        paramValues.next();

        out.println(" Parameter
        Name: " +
        parameter.getName());
        out.println(" Parameter
        Value: "
        +
        parameterValue.getValue
        ());

        }
        } else
        {
        // テンプレートが使用されていなかった場合は、
        // UserPresentation のみ使用可能 (ルールを
        // 作成した際に指定した場合)
        out.print(treeAction.ge
        tValueUserPresentation(
        ));
        }

        out.print("</td></tr>")
        ;
      }
      out.print("</table>");
    }
  }
  /*
    ルール・セットを印刷するメソッド
  */
public static String printRuleSet(BusinessRule
ruleArtifact)
{
   out.clear();
   out.printlnBold("Rule Set");
   RuleSet ruleSet = (RuleSet) ruleArtifact;
   out.println("Name: " + ruleSet.getName());
   out.println("Namespace: " +
   ruleSet.getTargetNameSpace());
```

```
// ルール・セットのルールをルール・ブロックに格納
RuleBlock ruleBlock =
ruleSet.getFirstRuleBlock();

Iterator<RuleSetRule> ruleIterator =
ruleBlock.iterator();

RuleSetRule rule = null;

// ルール・ブロック内のルールを反復処理
while (ruleIterator.hasNext())
{
  rule = ruleIterator.next();
  out.printBold("Rule: ");
  out.println(rule.getName());
  out.println("Display Name: " +
  rule.getDisplayName());
  out.println("Description: " +
  rule.getDescription());
  // ユーザーの拡張プレゼンテーションにより、パラメーターの値が
  // プレゼンテーションに対して自動的に設定され、テンプレートに
  // 初期ルールが定義されている場合は、ユーザーの拡張プレゼンテーションが
  // 表示可能になる。テンプレートが定義されていない場合、ユーザーの
  // 拡張プレゼンテーションは通常のプレゼンテーションと同じになる。
  out.println("Expanded User
  Presentation: "
    +
  rule.getExpandedUserPre
  sentation());
  // 通常のユーザー・プレゼンテーションにはストリング内にプレース
  // ホルダーが設定されるため、テンプレートに初期ルールが定義されて
  // いた場合はパラメーターを置き換えることができる。テンプレートに
  // 初期ルールが定義されていなかった場合、ユーザー・プレゼンテーションは
  // プレースホルダーのない単純なストリングになる。プレースホルダーは、
  // [n] の形式で表記される。この n は、テンプレート内のパラメーターの
  // 索引値 (0 基準) を表す。この値を使用して、編集可能なパラメーターが
  // 設定されているフィールドを編集するためのインターフェースを
  // 作成することができる。
  out.println("User Presentation: " +
  rule.getUserPresentation());

  // ルールがテンプレートで定義されていたかどうかを確認
  if (rule instanceof
  RuleSetTemplateInstanceRule) {
    RuleSetTemplateInstance
    Rule templateInstance =
    (RuleSetTemplateInstanc
    eRule) rule;

    RuleSetRuleTemplate
    template =
    templateInstance
    .getRuleSetRuleTemplate
    ();

    List<Parameter>
    parameters =
    template.getParameters(
    );
    Iterator<Parameter>
    paramIterator =
    parameters.iterator();

    Parameter parameter =
    null;

    // すべてのパラメーターを取得して、名前と値を出力
```

```
            while
            (paramIterator.hasNext(
            ))
            {
            parameter =
            paramIterator.next();

            out.println("Parameter
            Name: " +
            parameter.getName());
            out.println("Parameter
            Value: "
              +
            templateInstance.getPar
            ameterValue(
            parameter.getName()).ge
            tValue());
            }
          }
        }
        out.println("");
        return out.toString();
      }
}
```

## その他の照会例

以下の例は、例 1 から 15 のアプリケーションには含まれていませんが、ビジネス・ルール・グループを取得する照会の作成方法を示す追加の例として提供されています。

これらの例では、異なるプロパティーとワイルドカード値 (「_」、「%」) が、さまざまな演算子 (AND、OR、LIKE、NOT_LIKE、EQUAL および NOT_EQUAL) とともに使用されています。

### 例

これらの例では、4 つのビジネス・ルール・グループのさまざまな組み合わせを返す照会を実行します。照会ではビジネス・ルール・グループのさまざまな属性とプロパティーを使用するため、これらの属性およびプロパティーを理解していることが重要です。

```
名前: BRG1
ターゲット名前空間 : http://BRG1/com/ibm/br/rulegroup
プロパティー:
organization, 8JAA
department, claims
ID, 00000567
地域: SouthCentralRegion
マネージャー: Joe Bean

名前: BRG2
ターゲット名前空間 : http://BRG2/com/ibm/br/rulegroup
プロパティー:
organization, 7GAA
department, accounting
ID, 0000047
ID_cert45, ABC
地域: NorthRegion

名前: BRG3
ターゲット名前空間: http://BRG3/com/ibm/br/rulegroup
プロパティー:
```

```
organization, 7FAB
department, finance
ID, 0000053
ID_app45, DEF
地域: NorthCentralRegion

名前: BRG4
ターゲット名前空間: http://BRG4/com/ibm/br/rulegroup
プロパティー:
organization, 7HAA
department, shipping
ID, 0000023
ID_app45, GHI
地域: SouthRegion
```

## 単一プロパティーによる照会

これは、単一プロパティーによる照会例です。

```
List<BusinessRuleGroup> brgList = null;

brgList = BusinessRuleManager.getBRGsBySingleProperty(
    "department", QueryOperator.EQUAL,
    "accounting", 0, 0);
// BRG2 を返す
```

## 値の先頭と末尾にプロパティーとワイルドカード (%) を指定したビジネス・ルール・グループの照会

これは、値の先頭と末尾にプロパティーとワイルドカード (%) を指定したビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND Prop (Prop はプロパティーを表す)
QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "region", QueryOperator.LIKE,
  "%Region");

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode(
  "ID", QueryOperator.LIKE,
  "000005%");

QueryNode queryNode =
QueryNodeFactory.createAndNode(leftNode,
  rightNode);

brgList =
BusinessRuleManager.getBRGsByProperties(queryNode, 0, 0);
// BRG1 と BRG3 を返す
```

## プロパティーとワイルドカード (_) によるビジネス・ルール・グループの照会

これは、プロパティーとワイルドカード (_) によるビジネス・ルール・グループの照会例です。

```
brgList = BusinessRuleManager.getBRGsBySingleProperty("ID",
QueryOperator.LIKE, "00000_3", 0, 0);

// BRG3 と BRG4 を返す
```

### 複数のワイルドカード (「_」と「%」) を使用したプロパティーによるビジネス・ルール・グループの照会

これは、複数のワイルドカード (「_」と「%」) を使用したプロパティーによるビジネス・ルール・グループの照会例です。

```
brgList =
BusinessRuleManager.getBRGsBySingleProperty("region",
QueryOperator.LIKE, "__uth%Region",
  0, 0);

// BRG1 と BRG4 を返す
```

### NOT_LIKE 演算子とワイルドカード (「_」) によるビジネス・ルール・グループの照会

これは、NOT_LIKE 演算子とワイルドカード (「_」) によるビジネス・ルール・グループの照会例です。

```
brgList =
BusinessRuleManager.getBRGsBySingleProperty("organization",
QueryOperator.NOT_LIKE,
  "7__A", 0, 0);

// BRG1 と BRG3 を返す

brgList =
BusinessRuleManager.getBRGsBySingleProperty("organization",
QueryOperator.NOT_LIKE,
  "7%", 0, 0);

// BRG1 を返す
```

### NOT_EQUAL 演算子によるビジネス・ルール・グループの照会

これは、NOT_EQUAL 演算子によるビジネス・ルール・グループの照会例です。

```
brgList =
BusinessRuleManager.getBRGsBySingleProperty("department",
QueryOperator.NOT_EQUAL,
  "claims", 0, 0);
// BRG1 を返す
```

### PropertyIsDefined によるビジネス・ルール・グループの照会

これは、PropertyIsDefined によるビジネス・ルール・グループの照会例です。

```
PropertyIsDefinedQueryNode node =
QueryNodeFactory.createPropertyIsDefinedQueryNode("manager"
);

brgList = BusinessRuleManager.getBRGsByProperties(node, 0,
0);

// BRG1 を返す
```

### NOT PropertyIsDefined によるビジネス・ルール・グループの照会

これは、NOT PropertyIsDefined によるビジネス・ルール・グループの照会例です。

```
// 形式: NOT Prop  (Prop はプロパティーを表す)
QueryNode node =
QueryNodeFactory.createPropertyIsDefinedQueryNode("manager"
);

NotNode notNode = QueryNodeFactory.createNotNode(node);
```

```
brgList = BusinessRuleManager.getBRGsByProperties(notNode,
0, 0);

// BRG1 を返す
```

## 単一の NOT ノードを使用した複数のプロパティーによるビジネス・ルール・グループの照会

これは、単一の NOT ノードを使用した複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND NOT Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.EQUAL, "accounting");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID",
  QueryOperator.LIKE, "00000%");

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
  notNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG2 を返す
```

## 複数の NOT ノードが AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、複数の NOT ノードが AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT Prop AND NOT Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
QueryOperator.EQUAL, "accounting");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "cla%");

NotNode notNode2 =
QueryNodeFactory.createNotNode(leftNode);

AndNode andNode = QueryNodeFactory.createAndNode(notNode,
notNode2);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG1 と BRG2 を返す
```

## 複数の NOT ノードが OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、複数の NOT ノードが OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT Prop OR NOT Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "acc%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "department", QueryOperator.EQUAL,
  "claims");

NotNode notNode2 =
QueryNodeFactory.createNotNode(leftNode);

OrNode orNode = QueryNodeFactory.createOrNode(notNode,
notNode2);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

//BRG1、BRG2、BRG3、BRG4 を返す
```

## 複数の AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、複数の AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND Prop) AND (Prop AND Prop)  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "acc%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.EQUAL, "7GAA");

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(leftNode,rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("ID",
QueryOperator.LIKE,"000004_");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
QueryOperator.EQUAL,
  "NorthRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, rightNode2);

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft,andNodeRight);
```

```
brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG2 を返す
```

## AND 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、AND 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND Prop) OR (Prop AND NOT Prop)  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE, "acc%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.EQUAL, "7GAA");

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(leftNode, rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.EQUAL, "8JAA");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE, "%lRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, notNode);

OrNode orNode = QueryNodeFactory.createOrNode(andNodeLeft,
andNodeRight);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG2 と BRG3 を返す
```

## AND 演算子と NOT 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、AND 演算子と NOT 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND NOT (Prop AND Prop)  (Prop はプロパティーを表す)
QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID",
QueryOperator.LIKE, "000005%");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
QueryOperator.EQUAL,
  "8JAA");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
  "%lRegion");
```

```
AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2, rightNode2);

NotNode notNode =
QueryNodeFactory.createNotNode(andNodeRight);

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
notNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG3 を返す
```

## NOT 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、NOT 演算子と OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT (Prop AND Prop) OR Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8_A_");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
  "%lRegion");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

NotNode notNode =
QueryNodeFactory.createNotNode(andNodeRight);

OrNode orNode = QueryNodeFactory.createOrNode(notNode,
rightNode);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG3 を返す
```

## ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND (Prop AND (Prop AND Prop))  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");
```

```
QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

PropertyIsDefinedQueryNode node2 =
QueryNodeFactory.createPropertyIsDefinedQueryNode("ID_cert4
5");

AndNode andNode = QueryNodeFactory.createAndNode(node2,
andNodeLeft);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);
// BRG2 を返す
```

## ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND (Prop AND Prop)) AND Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",QueryOper
ator.LIKE,
"___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
QueryOperator.LIKE,
"7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
QueryOperator.LIKE,
"%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_app45",QueryOp
erator.LIKE, "GH_");

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft, leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG4 を返す
```

## ネストされた AND 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会

これは、ネストされた AND 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop AND (Prop AND (Prop AND NOT Prop))  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%lRegion");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,notNode);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "AB_");

AndNode andNode = QueryNodeFactory.createAndNode(leftNode,
andNodeLeft);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG2 を返す
```

## ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、ネストされた AND 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop AND (Prop AND Prop)) AND Prop - 空の値を返す  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");
```

```
AndNode andNodeRight =
QueryNodeFactory.createAndNode(leftNode2,rightNode2);

AndNode andNodeLeft =
QueryNodeFactory.createAndNode(rightNode,andNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "GH_");

AndNode andNode =
QueryNodeFactory.createAndNode(andNodeLeft, leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

//BRG は返さない
```

## ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop OR (Prop OR Prop)) OR Prop  (Prop はプロパティーを表す)

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(leftNode2,rightNode2);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(rightNode,orNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "GH_");

OrNode orNode = QueryNodeFactory.createOrNode(orNodeLeft,
leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG1 を返す
```

## ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会

これは、ネストされた OR 演算子で結合された複数のプロパティーによるビジネス・ルール・グループの照会例です。

```
// 形式: (Prop OR (Prop OR NOT Prop)) OR Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(leftNode2,notNode);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(rightNode,orNodeRight);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("ID_cert45",
  QueryOperator.LIKE,
  "GH_");

OrNode orNode = QueryNodeFactory.createOrNode(orNodeLeft,
leftNode);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG3 を返す
```

## ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会

これは、ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: Prop OR NOT(Prop OR Prop)  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "___thRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode(
  "organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "department",
  QueryOperator.LIKE,
```

```
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(rightNode2,
  rightNode);

NotNode notNode =
QueryNodeFactory.createNotNode(orNodeRight);

OrNode orNodeLeft = QueryNodeFactory.createOrNode(leftNode,
notNode);

brgList =
BusinessRuleManager.getBRGsByProperties(orNodeLeft, 0, 0);

// BRG3 を返す
```

## ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会

これは、ネストされた OR 演算子で結合された複数のプロパティーと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT(Prop OR Prop) OR Prop  (Prop はプロパティーを表す)
QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%lRegion");

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode(
  "organization",
  QueryOperator.LIKE,
  "7%");

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode(
  "department",
  QueryOperator.LIKE,
  "%ing");

OrNode orNodeRight =
QueryNodeFactory.createOrNode(rightNode2,rightNode);

NotNode notNode =
QueryNodeFactory.createNotNode(orNodeRight);

OrNode orNodeLeft =
QueryNodeFactory.createOrNode(notNode,leftNode);

brgList =
BusinessRuleManager.getBRGsByProperties(orNodeLeft, 0, 0);

// BRG2 と BRG4 を返す
```

## AND 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会

これは、AND 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会例です。

```
// 形式: AND リスト
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
```

```
  QueryOperator.LIKE,
  "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7%");

list.add(rightNode2);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "7H%");

list.add(leftNode2);

AndNode andNode = QueryNodeFactory.createAndNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG4 を返す
```

## AND 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会

これは、AND 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT ノードによる AND リスト
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

NotNode notNode =

QueryNodeFactory.createNotNode(rightNode2);

list.add(notNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

list.add(leftNode);
```

```
QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",

list.add(leftNode2);

AndNode andNode = QueryNodeFactory.createAndNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(andNode,
0, 0);

// BRG4 を返す
```

## OR 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会

これは、OR 演算子で結合されたノードのリストによるビジネス・ルール・グループの照会例です。

```
// 形式: OR リスト
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%thRegion");

list.add(rightNode);

QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

list.add(rightNode2);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

list.add(leftNode);

OrNode orNode = QueryNodeFactory.createOrNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

// BRG3 を返す
```

## OR 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会

これは、OR 演算子で結合されたノードのリストと NOT ノードによるビジネス・ルール・グループの照会例です。

```
// 形式: NOT ノードによる OR リスト
List<QueryNode> list = new ArrayList<QueryNode>();

QueryNode rightNode =
QueryNodeFactory.createPropertyQueryNode("region",
  QueryOperator.LIKE,
  "%thRegion");

list.add(rightNode);
```

```
QueryNode rightNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

NotNode notNode =
QueryNodeFactory.createNotNode(rightNode2);

list.add(notNode);

QueryNode leftNode =
QueryNodeFactory.createPropertyQueryNode("department",
  QueryOperator.LIKE,
  "%ing");

list.add(leftNode);

QueryNode leftNode2 =
QueryNodeFactory.createPropertyQueryNode("organization",
  QueryOperator.LIKE,
  "8%");

list.add(leftNode2);

OrNode orNode = QueryNodeFactory.createOrNode(list);

brgList = BusinessRuleManager.getBRGsByProperties(orNode,
0, 0);

//BRG1、BRG2、BRG3、BRG4 を返す
```

# 第 13 章 ビジネス・プロセスおよびタスク用クライアント・アプリケーションの開発

モデル化ツールを使用して、ビジネス・プロセスやタスクを作成しデプロイすることができます。そのようなプロセスとタスクは実行時に相互作用し、例えば、プロセスが開始すると、タスクが要求されて完了します。プロセスおよびタスクとは、Business Process Choreographer Explorer を使用して対話できますが、Business Process Choreographer API を使用して、このような対話用にカスタマイズしたクライアントを開発することもできます。

### このタスクについて

これらのクライアントは、Business Process Choreographer Explorer JavaServer Faces (JSF) コンポーネントを活用する Enterprise JavaBeans™ (EJB) クライアント、Web サービス・クライアント、または Web クライアントです。これらのクライアントを開発するために、Business Process Choreographer は、Enterprise JavaBeans (EJB) API と Web サービス用インターフェースを提供しています。EJB API は、別の EJB アプリケーションを含むすべての Java アプリケーションによってアクセスできます。 Web サービス用インターフェースには、Java 環境または Microsoft® .Net 環境のいずれかからアクセスできます。

## ビジネス・プロセスおよびヒューマン・タスクと対話するためのプログラミング・インターフェースの比較

ビジネス・プロセスおよびヒューマン・タスクと対話するクライアント・アプリケーションの作成には、Enterprise JavaBeans (EJB)、Web サービス、Java Message Service (JMS) および Representational State Transfer (REST) サービスなどの汎用プログラミング・インターフェースを使用できます。これらのインターフェースには、それぞれ異なる特性があります。

選択するプログラミング・インターフェースは、いくつかの要因によって左右されます。この要因には例えば、クライアント・アプリケーションが提供しなければならない機能、既存のエンド・ユーザー・クライアント・インフラストラクチャーがあるかどうか、ヒューマン・ワークフローを処理するかどうか、などがあります。使用するインターフェースを決定するためのヒントとして、EJB、Web サービス、JMS、および REST プログラミング・インターフェースの特性を比較した表を以下に示します。

© IBM Corporation 2005, 2006 © IBM 2005, 2008

| | EJB インターフェース | Web サービス・インターフェース | JMS メッセージ・インターフェース | REST インターフェース |
|---|---|---|---|---|
| 機能 | このインターフェースはビジネス・プロセスとヒューマン・タスクの両方に使用できます。一般的にプロセスおよびタスクを処理するクライアントを作成する場合に、このインターフェースを使用します。 | このインターフェースはビジネス・プロセスとヒューマン・タスクの両方に使用できます。プロセスとタスクからなる既知のセットに関してクライアントを作成する場合に、このインターフェースを使用します。 | このインターフェースはビジネス・プロセスにのみ使用できます。既知のプロセス・セットに関してメッセージング・クライアントを作成する場合に、このインターフェースを使用します。 | このインターフェースはビジネス・プロセスとヒューマン・タスクの両方に使用できます。このインターフェースは、既知の一式のプロセスおよびタスクに対する Web 2.0 スタイルのクライアントを作成するために使用します。 |
| データ処理 | ビジネス・オブジェクト・メタデータにアクセスするためのスキーマのリモート成果物ロードがサポートされます。<br><br>EJB クライアント・アプリケーションが接続先の WebSphere Process Server と同じセル内で稼働する場合、プロセスおよびタスクのビジネス・オブジェクトに必要なスキーマをクライアント上で使用可能にする必要はなく、リモート成果物ローダー (RAL) を使用してそれらのスキーマをサーバーからロードすることができます。<br><br>クライアント・アプリケーションが完全な WebSphere Process Server サーバー・インストール環境で稼働する場合は、セル間でも RAL を使用できます。ただし、クライアント・アプリケーションが WebSphere Process Server クライアント・インストールで稼働するセル間セットアップでは、RAL を使用できません。 | 入力データ、出力データ、および変数のスキーマ成果物は、クライアント上で適切な形式で使用できなければなりません。 | 入力データ、出力データ、および変数のスキーマ成果物は、クライアント上で適切な形式で使用できなければなりません。 | 入力データ、出力データ、および変数のスキーマ成果物は、クライアント上で適切な形式で使用できなければなりません。 |
| クライアント環境 | WebSphere Process Server インストールまたは WebSphere Process Server クライアント・インストール。 | Web サービス呼び出しをサポートするランタイム環境 (例えば Microsoft.NET 環境など)。 | JMS クライアントをサポートするランタイム環境 (例えば、SCA JMS インポートを使用する SCA モジュールなど)。 | REST クライアントをサポートする任意のランタイム環境。 |

| | EJB インターフェース | Web サービス・インターフェース | JMS メッセージ・インターフェース | REST インターフェース |
|---|---|---|---|---|
| セキュリティー | Java 2 Enterprise Edition (J2EE) セキュリティー。 | Web サービス・セキュリティー。 | WebSphere Process Server インストール用の Java 2 Enterprise Edition (J2EE) セキュリティー。JMS クライアント・アプリケーションが API メッセージを格納するキューを、例えば WebSphere MQ セキュリティー・メカニズムを使用して、保護することもできます。 | REST メソッドを呼び出すクライアント・アプリケーションは、適切な HTTP 認証メカニズムを使用する必要があります。 |

**関連タスク**

218 ページの『ビジネス・プロセスおよびヒューマン・タスク用 EJB クライアント・アプリケーションの開発』
EJB API は、WebSphere Process Server 上にインストールされているビジネス・プロセスやヒューマン・タスクを処理する EJB クライアント・アプリケーションを開発するための汎用的な方法をいくつか提供します。

283 ページの『Web サービス API クライアント・アプリケーションの開発』
Web サービス API を介してビジネス・プロセス・アプリケーションとヒューマン・タスク・アプリケーションにアクセスするクライアント・アプリケーションを開発できます。

315 ページの『Business Process Choreographer JMS API を使用したクライアント・アプリケーションの開発』
Java Messaging Service (JMS) API を介してビジネス・プロセス・アプリケーションに非同期でアクセスするクライアント・アプリケーションを開発できます。

## ビジネス・プロセスおよびタスク・データに対する照会

長時間実行ビジネス・プロセスおよびヒューマン・タスクのインスタンス・データはデータベースに永続的に格納され、照会によってアクセスできます。また、ビジネス・プロセス・テンプレートおよびヒューマン・タスク・テンプレートのテンプレート・データには、QUERY インターフェースを使用してアクセスできます。

Business Process Choreographer では以下の EJB QUERY インターフェースが使用可能です。

| | 説明 |
|---|---|
| Business Process Choreographer EJB 照会 API | インスタンス・データとテンプレート・データにアクセスできます。プロセスとタスクに関連したシステム内のデータはすべて、このインターフェースからアクセス可能です。Business Flow Manager および/または Human Task Manager の関連メソッドには、以下のものがあります。<br>• query<br>• queryAll<br>• queryProcessTemplates<br>• queryTaskTemplates |

| | 説明 |
|---|---|
| Business Process Choreographer EJB 照会テーブル API | インスタンス・データとテンプレート・データにアクセスできます。このインターフェースは、Business Process Choreographer 照会テーブルの照会に使用されます。これは、プロセス・リストとタスク・リストの照会専用のテーブルです。Business Flow Manager の関連メソッドは、以下のとおりです。<br>• queryEntities<br>• queryEntityCount<br>• queryRows<br>• queryRowCount |

プロセスまたはタスクの関連データにアクセスするクライアントによっては、上記にリストしたインターフェースの 1 つ以上に対応している可能性があります。Business Process Choreographer では、REST および Web サービスの API を使用して、タスクおよびプロセス・リストのデータを照会することもできます。ただし、大容量のプロセス・リストおよびタスク・リストの照会には、パフォーマンス上の理由から、Business Process Choreographer EJB 照会テーブル API を使用してください。

## Business Process Choreographer での照会テーブル

照会テーブルとは、タスク・リストおよびプロセス・インスタンス・リストとして参照される情報の抽象定義で、タスクまたはビジネス・プロセスを処理するユーザーに提示されます。照会テーブルはカスタマイズできます。例えば構成オプションで、特定のシナリオに関連するタスクまたはプロセス・インスタンスだけが照会テーブルに含まれるように指定できます。パフォーマンスが重要な場合 (大容量のプロセス・リストやタスク・リストを照会する場合など) は、照会テーブルを使用してください。

照会テーブルは、Business Process Choreographer の事前定義データベース・ビューと既存の QUERY インターフェースを、機能面とパフォーマンス面で拡張します。

- 照会テーブルは、パフォーマンスの観点から最適化されたアクセス・パターンを使用して、プロセスおよびタスク・リスト照会を実行するために最適化されています。
- 照会テーブルは、タスクまたはプロセス・リストに格納する内容の抽象定義です。照会テーブルを定義すると、必要な情報へのアクセスが簡素化され、統合されます。
- 照会テーブルにより、許可およびフィルターのオプションをきめ細かく構成できます。
- 照会テーブルの内容の定義は、タスクおよびプロセス・リストに固有です。例えば、処理するエンティティー (タスク、プロセス、エスカレーションなど) をまず選択します。次に、エンティティーの表示に必要な追加情報 (タスクの説明や照会プロパティーなど) を選択します。

照会テーブルには、定義済み照会テーブル、補足照会テーブル、複合照会テーブルの 3 種類があります。これらのテーブルは、いずれも照会テーブル API を使用して照会します。複合照会テーブルおよび補足照会テーブルは、Query Table Builder というツールを使用して作成します。

**関連タスク**

照会テーブルの管理
Query Table Builder を使用して開発した Business Process Choreographer の照会テーブルを管理するには、manageQueryTable.py スクリプトを使用します。

照会テーブルのデプロイ
manageQueryTable.py スクリプトを使用して、補足照会テーブルと複合照会テーブルを Business Process Choreographer 内にデプロイします。照会テーブルは、稼働中のスタンドアロン・サーバー、または少なくとも 1 つのメンバーが稼働中のクラスターにデプロイされます。補足照会テーブルと複合照会テーブルのアンデプロイメントは稼働中のサーバーに対しても実行されます。補足照会テーブルでは、関連する物理データベース・オブジェクト (通常は、データベース・ビューまたはデータベース表) が存在しない場合は、照会テーブルを使用する前にそれらを作成する必要があります。

**関連資料**

Business Process Choreographer のデータベース・ビュー
この参照情報では、事前定義データベース・ビューの列について説明します。

## 定義済み照会テーブル

Business Process Choreographer の定義済み照会テーブルは、TASK や PROCESS_INSTANCE などの定義済み Business Process Choreographer データベース・ビューの照会テーブル表現です。定義済み照会テーブルは、Business Process Choreographer データベースのデータを単純なビューに表示します。ただし、定義済み照会テーブルは、許可、機能、およびパフォーマンスという点で、定義済みデータベース・ビューとは異なります。

照会テーブルを使用するときには、許可およびフィルター・オプションを詳細に構成できます。

定義済み照会テーブルは、定義済みデータベース・ビューと同じ物理データを基礎とするため、その構造も定義済みデータベース・ビューと同じです。ただし、定義済み照会テーブルは、プロセスおよびタスク・リストの照会を実行するために最適化されるという点で、定義済みデータベース・ビューよりも機能とパフォーマンスが強化されています。

定義済み照会テーブルは、照会テーブル API を使用して直接照会できます。しかし提案されている照会テーブルの使用方法は、複合照会テーブルを開発し、そこに照会の実行時に取得する情報を含めることです。

照会テーブル API を使用して照会テーブルで照会を実行する際には、パラメーターをサブミットできます。定義済み照会テーブルは、パラメーターをサポートしません。

以下の定義済み照会テーブルは、直接照会に使用することも、複合照会テーブルの 1 次照会テーブル、あるいは複合照会テーブルに接続される照会テーブルとして使用することもできます。

以下の定義済み照会テーブルにはインスタンス・データが含まれます。これらのテーブルに対する照会は、すべての認証済みユーザーが実行できます。

- ACTIVITY
- ACTIVITY_ATTRIBUTE
- ACTIVITY_SERVICE
- APPLICATION_COMP
- ESCALATION
- ESCALATION_CPROP
- ESCALATION_DESC
- PROCESS_ATTRIBUTE
- PROCESS_INSTANCE
- QUERY_PROPERTY
- TASK
- TASK_CPROP
- TASK_DESC

許可はすべての作業項目に対して有効になります。つまり、全員、個人、グループの作業項目、および継承された作業項目に対して有効になります。インスタンス・データが含まれる定義済み照会テーブルでは、指定されていない限り、照会テーブル API は全員、個人、およびグループの作業項目にデフォルト設定されます。

以下の定義済み照会テーブルにはテンプレート・データが含まれます。これらのテーブルに対する照会は、管理ユーザーが照会テーブル API を使用して実行できます。

- ESC_TEMPL
- ESC_TEMPL_CPROP
- ESC_TEMPL_DESC
- PROCESS_TEMPLATE
- TASK_TEMPL
- TASK_TEMPL_CPROP
- TASK_TEMPL_DESC

テンプレート・データが含まれる定義済み照会テーブルの許可は、作業項目には適用されません。これらの定義済み照会テーブルは、管理者が AdminAuthorizationOptions オブジェクトを使用することによってのみ照会できます。

**関連概念**

193 ページの『補足照会テーブル』

Business Process Choreographer の補足照会テーブルは、外部データベース表またはビューからのデータ、あるいは全般的なデータベース・オブジェクトからのデータを照会テーブル API に公開します。補足照会テーブルを使用すると、この外部データを、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報と結合することができます。補足照会テーブルは、照会テーブル API を使用して直接照会できます。

194 ページの『複合照会テーブル』
Business Process Choreographer の複合照会テーブルは、定義済み照会テーブルと補足照会テーブルで構成されます。複合照会テーブルは、既存のテーブルまたはビューからのデータを結合します。一般に、複合照会テーブルはプロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) に表示される情報を取得するために使用されます。

197 ページの『照会テーブルの作成』
Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder というツールを使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

200 ページの『照会テーブル API の概要』
照会テーブル API のエンティティー・ベースの照会と行ベースの照会を使用して、Business Process Choreographer の照会テーブルに対する照会を実行できます。

**関連資料**

Business Process Choreographer のデータベース・ビュー
この参照情報では、事前定義データベース・ビューの列について説明します。

## 補足照会テーブル

Business Process Choreographer の補足照会テーブルは、外部データベース表またはビューからのデータ、あるいは全般的なデータベース・オブジェクトからのデータを照会テーブル API に公開します。補足照会テーブルを使用すると、この外部データを、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報と結合することができます。補足照会テーブルは、照会テーブル API を使用して直接照会できます。

補足照会テーブルが説明するオブジェクトは、Business Process Choreographer が維持するデータの付加的データが含まれるデータベース内にあります。オブジェクトは通常、データベース・ビューまたはデータベース表です。補足照会テーブルは、関連データベース・オブジェクトの列を説明します。補足照会テーブルの名前は、接頭部と名前で構成する必要があります (例えば、COMPANY.EXT_DATA)。

**注:** 照会テーブルと照会テーブル API のコンテキストでは、列は通常、属性と呼ばれます。照会テーブルのコンテンツはデータベースに保管されるため、「列」という用語が使用される場合もあります。

照会テーブル API を使用して照会テーブルで照会を実行する際には、パラメーターをサブミットできます。補足照会テーブルは、パラメーターをサポートしません。

作業項目での許可は、補足照会テーブルの場合にはサポートされません。すべての認証済みユーザーが、補足照会テーブルのコンテンツにアクセスできます。

**関連概念**

191 ページの『定義済み照会テーブル』
Business Process Choreographer の定義済み照会テーブルは、TASK や

PROCESS_INSTANCE などの定義済み Business Process Choreographer データベース・ビューの照会テーブル表現です。定義済み照会テーブルは、Business Process Choreographer データベースのデータを単純なビューに表示します。ただし、定義済み照会テーブルは、許可、機能、およびパフォーマンスという点で、定義済みデータベース・ビューとは異なります。

『複合照会テーブル』
Business Process Choreographer の複合照会テーブルは、定義済み照会テーブルと補足照会テーブルで構成されます。複合照会テーブルは、既存のテーブルまたはビューからのデータを結合します。一般に、複合照会テーブルはプロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) に表示される情報を取得するために使用されます。

197 ページの『照会テーブルの作成』
Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder というツールを使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

200 ページの『照会テーブル API の概要』
照会テーブル API のエンティティー・ベースの照会と行ベースの照会を使用して、Business Process Choreographer の照会テーブルに対する照会を実行できます。

## 複合照会テーブル

Business Process Choreographer の複合照会テーブルは、定義済み照会テーブルと補足照会テーブルで構成されます。複合照会テーブルは、既存のテーブルまたはビューからのデータを結合します。一般に、複合照会テーブルはプロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) に表示される情報を取得するために使用されます。

複合照会テーブルは、定義済み照会テーブルで利用できる情報と補足照会テーブルで利用できる情報を組み合わせたものです。各種の構成オプションが用意されており (ほとんどは照会の応答時間に影響を及ぼします)、複合照会テーブルで使用可能な情報を指定することができます。

### 構造

複合照会テーブルは、1 次照会テーブルと 0 個以上の接続される照会テーブルから構成されます。複合照会テーブルの名前は、接頭部と名前から構成されます (例: COMPANY.TODO_TASK_LIST)。

- 1 次照会テーブルは、複合照会テーブルに格納される主な情報を構成します。

  許可が必要な場合は、1 次照会テーブルのオブジェクトが使用可能な作業項目に照らして検査され、許可オプションが考慮に入れられます。例えば、ユーザーが潜在的所有者であるヒューマン・タスクのみを返すように照会を定義します。

  また、複合照会テーブルの各オブジェクトは、1 次照会テーブルの 1 次キーによって一意的に識別できます。例えば TASK の場合、これはタスク ID TKIID で

す。1 次照会テーブルとして選択できるのは定義済み照会テーブルのみです。通常、1 次照会テーブルは、定義済み照会テーブル TASK または定義済み照会テーブル PROCESS_INSTANCE です。

- 複合照会テーブルでは、0 個以上の接続される照会テーブルを定義できます。

  許可のフィルターとオプション、および 1 次フィルターの結果として複合照会テーブルに入れられる各オブジェクトには、接続される照会テーブルに含まれる追加情報を付加できます。例えば、特定のロケールのためのタスクの説明を、1 次照会テーブル TASK を持つ複合照会テーブルに追加できます。

  1 次照会テーブルとその接続される照会テーブルの間では、(必要に応じて選択基準を使用して) 1 対 1 または 1 対 0 の関係を維持しなければなりません。接続される照会テーブルは、既にシステムにデプロイされている定義済み照会テーブルおよび補足照会テーブルにすることができます。

**パフォーマンス**

照会テーブルに対する照会の応答時間は、主に選択した許可オプション、フィルター、および選択基準の影響を受けます。

- 許可オプションはパフォーマンスに大きく影響します。許可は、可能な限り少ないオプション (個人およびグループの作業項目など) を使用して有効化してください。継承される作業項目の使用は避けてください。許可オプションは、照会の実行時にさらに制限できます。また、必要でない場合は、作業項目を使用する許可が不要であると指定してください。
- 作業項目を使用する許可が必要な場合は、許可フィルターを指定してください。例えば、潜在的所有者の作業項目を持つ照会テーブルのオブジェクトのみを許可するには、WI.REASON=REASON_POTENTIAL_OWNER を使用します。
- 1 次照会テーブルでのフィルター処理は効率的です (例えば、TASK が 1 次照会テーブルである照会テーブルで作動可能状態になっているタスクのみを許可するなど)。
- 照会テーブルでのフィルター処理および照会フィルターは、照会の実行時に渡されるフィルターであり、1 次フィルターよりもパフォーマンスが落ちます。
- 可能な場合は、フィルターおよび選択基準のパラメーターを使用するのを避けてください。
- フィルターおよび選択基準で `LIKE` 演算子を使用するのは避けてください。

**実装**

複合照会テーブルはデータベースに物理表現を持っていません。複合照会テーブルは、タスクおよびプロセス・リストの照会に最適化された SQL で実現されています。

**許可**

複合照会テーブルは、許可を必要とするようにも許可を必要としないようにも構成できます。許可が必要な場合は、1 次照会テーブルのオブジェクトが、SQL の結合を使用して、関連した作業項目の WORK_ITEM 照会テーブルに照らして検査されます。インスタンス・データを含む 1 次照会テーブル (TASK や PROCESS_INSTANCE などの照会テーブル) の場合は、これがデフォルトです。

許可が必要な場合は、複合照会テーブルの定義で以下の許可オプションを使用できます。

- **「全員」作業項目**: 指定すると、関連した「全員」作業項目を持つオブジェクトが複合照会テーブルに入れられます。
- **「個人」作業項目**: 指定すると、関連した「個人」作業項目を持つオブジェクトが複合照会テーブルに入れられます。
- **「グループ」作業項目**: 指定すると、関連した「グループ」作業項目を持つオブジェクトが複合照会テーブルに入れられます。
- **継承された作業項目**: 指定すると、構成されている関連した「全員」、「個人」、または「グループ」作業項目を持ち、プロセス・インスタンスを親として持つオブジェクト (参加ヒューマン・タスクなど) が複合照会テーブルに入れられます。通常、継承された作業項目は管理者の場合にのみ有用です。

**パラメーター**

照会テーブルのフィルターおよび選択基準でパラメーターを使用すると、定義したフィルターおよび選択基準の一部を動的にすることができます。

**フィルター**

フィルターは、以下のように照会テーブルの内容を制限するために使用します。

- **1 次フィルター**: このフィルターは、1 次照会テーブルで定義します。これは、1 次照会テーブルで定義されている列に対する条件を使用して、複合照会テーブルの内容を制限します。
- **許可フィルター**: これは、定義済み照会テーブル WORK_ITEM (許可の実現に使用します) で定義されている列を使用して、複合照会テーブルの内容を制限します。作業項目の作成は、Business Process Choreographer のプロセスおよびヒューマン・タスクのスタッフ動詞を使用して定義します。

**注:** 照会テーブルと照会テーブル API のコンテキストでは、列は通常、属性と呼ばれます。照会テーブルの内容はデータベースに格納されるため、「列」という用語も使用します。

**選択基準**

1 次照会テーブルのオブジェクトと接続される照会テーブルのオブジェクトの間では、1 対 1 または 1 対 0 の関係を維持しなければなりません。これを行うには、接続される照会テーブルに対する選択基準を使用します。例えば、TASK が 1 次照会テーブルであり、TASK_DESC が接続される照会テーブルであるとすると、通常、選択基準によって、複合照会テーブルのヒューマン・タスクに追加する説明のロケールが 1 つ選択されます。例えば、`LOCALE='en_US'` となります。

**関連概念**

191 ページの『定義済み照会テーブル』

Business Process Choreographer の定義済み照会テーブルは、TASK や PROCESS_INSTANCE などの定義済み Business Process Choreographer データベース・ビューの照会テーブル表現です。定義済み照会テーブルは、Business Process Choreographer データベースのデータを単純なビューに表示します。ただ

し、定義済み照会テーブルは、許可、機能、およびパフォーマンスという点で、定義済みデータベース・ビューとは異なります。

193 ページの『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、外部データベース表またはビューからのデータ、あるいは全般的なデータベース・オブジェクトからのデータを照会テーブル API に公開します。補足照会テーブルを使用すると、この外部データを、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報と結合することができます。補足照会テーブルは、照会テーブル API を使用して直接照会できます。

『照会テーブルの作成』
Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder というツールを使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

200 ページの『照会テーブル API の概要』
照会テーブル API のエンティティー・ベースの照会と行ベースの照会を使用して、Business Process Choreographer の照会テーブルに対する照会を実行できます。

## 照会テーブルの作成

Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder というツールを使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

Query Table Builder ツールは Eclipse プラグインとして使用可能であり、WebSphere Business Process Management SupportPacs のサイトからダウンロードできます。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

照会テーブル API を使用して照会テーブルを照会するコードの例を以下に示します。簡単にするため、例 1 および 2 では定義済み照会テーブル TASK を照会します。例 3 および 4 では、複合照会テーブルを照会します。ここで、この複合照会テーブルはシステムにデプロイされているとします。アプリケーションの開発時には、定義済み照会テーブルを直接照会するのではなく、複合照会テーブルを使用してください。

**例 1**

```
// get the naming context and lookup the business
// flow manager EJB home; note that the business flow
// manager EJB home should be cached for performance
// reasons; also, it is assumed that there's a EJB
// reference to the local business flow manager EJB
Context ctx = new InitialContext();
LocalBusinessFlowManagerHome home =
```

```
  (LocalBusinessFlowManagerHome)
  ctx.lookup("java:comp/env/ejb/BFM");

// create the business flow manager client-side stub
LocalBusinessFlowManager bfm = home.create();

// ************************************************
// ****************** example 1 *******************
// ************************************************

// execute a query against the predefined query table
// TASK; this relates to a simple My ToDo's task list
EntityResultSet ers = null;
ers = bfm.queryEntities("TASK", null, null, null);

// print the result to STDOUT
EntityInfo entityInfo = ers.getEntityInfo();
List attList = entityInfo.getAttributeInfo();
int attSize = attList.size();

Iterator iter = ers.getEntities().iterator();
while( iter.hasNext() ) {
  System.out.print("Entity: ");
  Entity entity = (Entity) iter.next();
  for (int i = attSize - 1; i >= 0; i--) {
    AttributeInfo ai = (AttributeInfo) attList.get(i);
    System.out.print(
     entity.getAttributeValue(ai.getName()));
 }
  System.out.println();
}
```

**例 2**

```
// ************************************************
// ****************** example 2 *******************
// ************************************************

// same example as example 1, but using the row based
// query approach
RowResultSet rrs = null;
rrs = bfm.queryRows("TASK", null, null, null);

attList = rrs.getAttributeInfo();
attSize = attList.size();

// print the result to STDOUT
while (rrs.next()) {
  System.out.print("Row: ");
  for (int i = attSize - 1; i >= 0; i--) {
    AttributeInfo ai = (AttributeInfo) attList.get(i);
    System.out.print(
     rrs.getAttributeValue(ai.getName()));
 }
  System.out.println();
}
```

**例 3**

```
// ************************************************
// ****************** example 3 *******************
// ************************************************

// execute a query against a composite query table
// that has been deployed on the system before;
// the name is assumed to be COMPANY.TASK_LIST
```

```
ers = bfm.queryEntities(
    "COMPANY.TASK_LIST", null, null, null);
^
//  print the result to STDOUT ...
```

**例 4**

```
// ************************************************
// ****************** example 4 *******************
// ************************************************

// query against the same query table as in example 3,
// but with customized options
FilterOptions fo = new FilterOptions();

// return only objects which are in state ready
fo.setQueryCondition("STATE=STATE_READY");

// sort by the id of the object
fo.setSortAttributes("ID");

// limit the number of entities to 50
fo.setThreshold(50);

// only get a sub-set of the defined attributes
// on the query table
fo.setSelectedAttributes("ID, STATE, DESCRIPTION");

AuthorizationOptions ao = new AuthorizationOptions();

// do not return objects that everybody is allowed
// to see
ao.setEverybodyUsed(Boolean.FALSE);

ers = bfm.queryEntities(
    "COMPANY.TASK_LIST", fo, ao, null);

//  print the result to STDOUT ...
```

**関連概念**

191 ページの『定義済み照会テーブル』
Business Process Choreographer の定義済み照会テーブルは、TASK や PROCESS_INSTANCE などの定義済み Business Process Choreographer データベース・ビューの照会テーブル表現です。定義済み照会テーブルは、Business Process Choreographer データベースのデータを単純なビューに表示します。ただし、定義済み照会テーブルは、許可、機能、およびパフォーマンスという点で、定義済みデータベース・ビューとは異なります。

193 ページの『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、外部データベース表またはビューからのデータ、あるいは全般的なデータベース・オブジェクトからのデータを照会テーブル API に公開します。補足照会テーブルを使用すると、この外部データを、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報と結合することができます。補足照会テーブルは、照会テーブル API を使用して直接照会できます。

194 ページの『複合照会テーブル』
Business Process Choreographer の複合照会テーブルは、定義済み照会テーブルと補足照会テーブルで構成されます。複合照会テーブルは、既存のテーブルまたは

ビューからのデータを結合します。一般に、複合照会テーブルはプロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) に表示される情報を取得するために使用されます。

『照会テーブル API の概要』
照会テーブル API のエンティティー・ベースの照会と行ベースの照会を使用して、Business Process Choreographer の照会テーブルに対する照会を実行できます。

**関連タスク**

照会テーブルの管理
Query Table Builder を使用して開発した Business Process Choreographer の照会テーブルを管理するには、manageQueryTable.py スクリプトを使用します。

照会テーブルのデプロイ
manageQueryTable.py スクリプトを使用して、補足照会テーブルと複合照会テーブルを Business Process Choreographer 内にデプロイします。照会テーブルは、稼働中のスタンドアロン・サーバー、または少なくとも 1 つのメンバーが稼働中のクラスターにデプロイされます。補足照会テーブルと複合照会テーブルのアンデプロイメントは稼働中のサーバーに対しても実行されます。補足照会テーブルでは、関連する物理データベース・オブジェクト (通常は、データベース・ビューまたはデータベース表) が存在しない場合は、照会テーブルを使用する前にそれらを作成する必要があります。

## 照会テーブル API の概要

照会テーブル API のエンティティー・ベースの照会と行ベースの照会を使用して、Business Process Choreographer の照会テーブルに対する照会を実行できます。

照会は、特定の 1 つの照会テーブルでのみ実行されます。複数の照会テーブル間の関係、つまり標準照会 API とデータベース・ビューという点での関係は、複合照会テーブルを使用して定義します。

照会テーブル API では、Business Process Choreographer の照会テーブルに対する照会を実行するために、以下の 2 つの概念を使用できます。

- **エンティティー・ベースの照会**: queryEntities メソッドと queryEntityCount メソッドを使用したエンティティー照会。照会テーブルに、1 次照会テーブルで定義された一意的に識別可能なエンティティーが含まれていることを前提とします。これらのエンティティーは、1 次照会テーブルの 1 次キーによって識別されます。
- **行ベースの照会**: queryRows メソッドと queryRowCount メソッドを使用した行の照会。JDBC のような結果セットを返します。この結果セットには、同じエンティティー (例えば、TKIID などのタスク ID で識別されるヒューマン・タスク) が複数回出現する場合があります。

queryEntities メソッドによって返されるエンティティー結果セットは、複合照会テーブル内の 1 次照会テーブルの意味構造を強調表示します。複合照会テーブルは、1 つの 1 次照会テーブルのみで構成されることも、さらに 1 つ以上の照会テーブルが接続されることもあります。複合照会テーブルのエンティティー・タイプは、

そこに含まれる 1 次照会テーブルによって決まります。例えば、1 次照会テーブル TASK が含まれる複合照会テーブルには、TASK タイプのエンティティーが含まれます。

各エンティティーは照会テーブル内で固有であるため、それぞれのエンティティーはエンティティー結果セット内でも固有です。エンティティーの一意性は、基礎となるデータの 1 次キーによって維持されます。例えば TASK エンティティーの場合、その 1 次キーはタスク ID の TKIID です。

queryRows メソッドによって返される行結果セットは、基礎となるデータベース・ビューおよびテーブルに対して実行された JDBC 照会が返す行で構成されます。これは、標準照会 API によって返される QueryResultSet と比較できます。一般に、行の数は、照会テーブルに含まれるエンティティーの数より大きくなります。行結果セットには、特定タスクの重複したエントリーが含まれる場合があります。例えば、照会で WI.REASON が選択された場合がこれに該当します。

**照会テーブル API の概要**

照会テーブル API メソッドのそれぞれに、以下のパラメーターがあります。

- **ストリング queryTableName**: 照会される照会テーブルの名前。定義済み照会テーブルの場合は、定義済み照会テーブルの名前です。複合照会テーブルおよび補足照会テーブルの場合には、*prefix.name* となります。
- **フィルター・オプション FilterOptions**: 結果セットを制限するオプションで、このオプションを使用してソート基準を指定できます。
- **許可オプション AuthorizationOptions**: 考慮する作業項目を指定するオプション。別のユーザーのための照会、または AdminAuthorizationOptions を使用して実行可能な管理照会を指定できます。
- **リスト・パラメーター**: 複合照会テーブルは、フィルターおよび選択基準のパラメーターを使用して定義できます。これらのパラメーターの値は、以下の引数を使用して指定します。

**FilterOptions**

- **個別**: この設定は、行ベースの照会を実行する場合にのみ有効です。 true に設定すると、個別の行が返されます。
- **ロケール**: ロケールは、フィルターまたは選択基準のシステム・パラメーターとして使用できます (例えば、`'LOCALE=$LOCALE'`)。ロケールが設定されていない場合は、サーバーのロケールが使用されます。
- **時間帯**: これは、日付を変換するために使用します (例えば、定義済み照会テーブル TASK の CREATED)。指定されていない場合は、サーバーの時間帯が使用されます。
- **しきい値**: 返される行またはエンティティーの数を制限します。しきい値の設定は、エンティティー・ベースの照会では正確でない場合があります。
- **スキップ・カウント**: 結果セット内でスキップする行またはエンティティーの数を指定します。
- **選択済み属性**: このコンマで区切った属性のリストは、照会によって取得する属性を指定します。インスタンス・データが含まれる定義済み照会テーブルなどで許可が必要な場合、定義された属性の他に「WI.」の接頭部が付いた作業項目情報

(例えば、WI.REASON) を取得できます。選択済み属性が指定されていない場合、照会テーブルに定義されたすべての属性が返され、作業項目情報は返されません。

**注:** 照会テーブルと照会テーブル API のコンテキストでは、列は通常、属性と呼ばれます。照会テーブルのコンテンツはデータベースに保管されるため、「列」という用語が使用される場合もあります。

- **照会条件**: 結果セットで追加のフィルタリングを行います。許可が必要であると設定されている場合には、照会テーブルに定義された属性を参照できます。接頭部「WI.」を使用して (例えば、WI.REASON=REASON_POTENTIAL_OWNER)、WORK_ITEM 照会テーブルに定義された列を参照することもできます。
- **ソート属性**: このコンマで区切った属性のリストは、ソート基準 (例えば、CREATED DESC) を定義します。

**AuthorizationOptions**

- **全員**: true (デフォルト) に設定すると、全員の作業項目 (スタッフ verb「everybody」) が考慮されます (照会テーブルで有効に設定されている場合)。
- **個人**: true (デフォルト) に設定すると、個人の作業項目 (例えば、スタッフ verb「Users」) が考慮されます (照会テーブルで有効に設定されている場合)。
- **グループ**: true (デフォルト) に設定すると、グループの作業項目 (例えば、スタッフ verb「Group」) が考慮されます (照会テーブルおよびヒューマン・タスク・コンテナーの両方で有効に設定されている場合)。
- **継承**: true に設定すると、継承された作業項目が考慮されます。この場合、例えばプロセス・インスタンスの管理者は、そのプロセス・インスタンスに対して作成された参加ヒューマン・タスク・インスタンスが照会テーブルに対して照会を実行するかどうかを確認できます。

AuthorizationOptions オブジェクトの代わりに、AdminAuthorizationOptions オブジェクトを照会テーブル API に渡すことができます。このオブジェクトは、呼び出し側が J2EE 役割 BPESystemAdministrator を持っている場合にのみ使用可能になります。 AdminAuthorizationOptions クラスは、AuthorizationOptions クラスから派生します。以下のオプションを使用できます。

- **onBehalfUser**: null (デフォルト) に設定すると、照会は許可を必要とする照会テーブルに対して実行されます。照会は、この特定のユーザーに対する作業項目に基づく許可を使用して結果を制限することなく実行されます。つまり、照会は照会テーブルに含まれるすべてのオブジェクトを返します。
- **onBehalfUser**: null (デフォルト) に設定すると、照会は許可を必要としない照会テーブルに対して実行されます。すべての認証済みユーザーに、照会テーブルのすべてのコンテンツが表示されます。照会の結果セットは、AuthorizationOptions または AdminAuthorizationOptions のどちらが使用されているかに関係なく、同じになります。
- **onBehalfUser**: 特定ユーザーの名前に設定すると、その指定されたユーザーに代わって照会が実行されます。
- **onBehalfUser**: 定義済み照会テーブルに対して使用する場合、テンプレート・データが必要であれば、onBehalfUser を null に設定する必要があります。

### パラメーター

複合照会テーブルは、フィルターまたは選択基準に含まれるパラメーターを使用して定義できます。 java.util.List リストに含まれる照会テーブル API には、照会を実行するために必要なすべてのパラメーターを com.ibm.bpe.Parameter クラスのパラメーターとして渡す必要があります。

### 照会テーブル条件言語 (QTCL)

照会テーブル条件言語 (QTCL) は、フィルターおよび選択基準を指定するために使用します。この明確に定義された言語を使用して、照会テーブルの属性に基づく条件を指定します。このセクションでは、照会テーブル API に関する QTCL の詳細を説明します。完全な仕様については、WebSphere Business Process Management SupportPacs のサイトを参照してください。このサイトで、PA71 WebSphere Process Server - Query Table Builder を探します。リンクにアクセスするには、このトピックの関連参照のセクションを参照してください。

**注:** 照会テーブルと照会テーブル API のコンテキストでは、列は通常、属性と呼ばれます。その一方、照会テーブルのコンテンツはデータベースに保管されるため、「列」という用語が使用される場合もあります。

- QTCL 式の副次式は、左側のオペランド、演算、右側のオペランドまたはオペランドのリストで構成されます。また、IS NULL などの単項演算子も使用できます。
- 左側のオペランドは、照会テーブルの属性名です。
- 右側のオペランドは、左側のオペランドの属性に定義された定数、またはリテラルです。
- QTCL 式は、特定のスコープ内で実行されます。このスコープが、式の左側で有効な属性を決定します。照会の条件 (照会フィルター) は、照会を実行する照会テーブルのスコープ内で実行されます。式の左側の有効な属性は、照会テーブルの属性です。照会テーブルで許可が必要な場合、接頭部「WI.」が付いた WORK_ITEM 照会テーブルの属性も有効になります。
- 有効な演算子は、<、>、<>、<=、>=、=、IN、NOT IN、IS NULL、IS NOT NULL、LIKE、IS NOT LIKE です。
- 副次式は、既知の意味構造を持つ AND および OR を使用して結合されます。副次式をグループ化するには、大括弧を使用します。定義済み TASK 照会テーブルで実行する照会の場合には、例えば `'(STATE=STATE_READY AND WI.REASON=REASON_POTENTIAL_OWNER) OR (WI.REASON=REASON_OWNER)'` のようになります。

### カウント照会の照会結果の詳細

照会テーブル API の queryEntityCount メソッドと queryRowCount メソッドは、単純な整数値を返します。適格なオブジェクトの数を取得する場合、実装はパフォーマンスのために最適化されます。

### queryEntities メソッドによって返される EntityResultSet の照会結果の詳細:

queryEntities メソッドによる EntityResultSet では、以下の照会結果の詳細が返されます。

- EntityResultSet では、照会テーブル名を取得できます。これは、照会が実行された照会テーブルの名前です。
- エンティティー・タイプ名を取得できます。照会が複合照会テーブルで実行された場合、これは 1 次照会テーブルの名前となります。それ以外の場合は、照会が実行された照会テーブルの名前です。
- EntityInfo オブジェクトを取得できます。 EntityInfo オブジェクトは、EntityResultSet に含まれるエンティティーの詳細を説明します。詳細には、属性、それぞれの属性の関連タイプおよびエンティティー・タイプが含まれます。
- FilterOptions で定義されている場合、その指定の順序でエンティティーがリストされます。
- EntityResultSet に含まれるエンティティーの数は、エンティティー・リストで size() メソッドを使用して取得します。

**queryRows メソッドによって返される RowResultSet の照会結果の詳細:**

queryRows メソッドによる RowResultSet では、以下の照会結果の詳細が返されます。

- RowResultSet では、照会テーブル名を取得できます。これは、照会が実行された照会テーブルの名前です。
- 1 次照会テーブルの名前を取得できます。照会が複合照会テーブルで実行された場合、これは 1 次照会テーブルの名前となります。それ以外の場合は、照会が実行された照会テーブルの名前です。
- 属性とそれぞれの関連タイプのリストを取得できます。
- next()、previous()、first()、および last() メソッドを使用して、指定された順序 (FilterOptions で定義されている場合) で RowResultSet をナビゲートできます。
- RowResultSet のサイズを取得できます。

**関連概念**

191 ページの『定義済み照会テーブル』
Business Process Choreographer の定義済み照会テーブルは、TASK や PROCESS_INSTANCE などの定義済み Business Process Choreographer データベース・ビューの照会テーブル表現です。定義済み照会テーブルは、Business Process Choreographer データベースのデータを単純なビューに表示します。ただし、定義済み照会テーブルは、許可、機能、およびパフォーマンスという点で、定義済みデータベース・ビューとは異なります。

193 ページの『補足照会テーブル』
Business Process Choreographer の補足照会テーブルは、外部データベース表またはビューからのデータ、あるいは全般的なデータベース・オブジェクトからのデータを照会テーブル API に公開します。補足照会テーブルを使用すると、この外部データを、ビジネス・プロセス・インスタンス情報またはヒューマン・タスク情報と結合することができます。補足照会テーブルは、照会テーブル API を使用して直接照会できます。

194 ページの『複合照会テーブル』
Business Process Choreographer の複合照会テーブルは、定義済み照会テーブルと補足照会テーブルで構成されます。複合照会テーブルは、既存のテーブルまたは

ビューからのデータを結合します。一般に、複合照会テーブルはプロセス・インスタンス・リストまたはタスク・リスト (ユーザーの予定など) に表示される情報を取得するために使用されます。

197 ページの『照会テーブルの作成』

Business Process Choreographer の補足照会テーブルおよび複合照会テーブルは、Query Table Builder というツールを使用してアプリケーションの開発中に作成します。定義済み照会テーブルは、作成することもデプロイすることもできません。定義済み照会テーブルは、Business Process Choreographer のインストール時に利用できる照会テーブルで、Business Process Choreographer データベース・スキーマでの成果物を簡略表示することができます。

## Business Process Choreographer EJB 照会 API

サービス API の query メソッドまたは queryAll メソッドを使用して、ビジネス・プロセスとタスクについて保管されている情報を取得します。

すべてのユーザーが query メソッドを呼び出すことができます。このメソッドは、作業項目が存在しているオブジェクトのプロパティーを戻します。 queryAll メソッドは、J2EE ロール BPESystemAdministrator、TaskSystemAdministrator、BPESystemMonitor、TaskSystemMonitor のいずれかが割り当てられているユーザーのみが呼び出すことができます。このメソッドは、データベースに格納されているすべてのオブジェクトのプロパティーを戻します。

すべての API 照会は SQL 照会にマップされます。生成される SQL 照会の形式は、次の要因によって異なります。

- 照会が、J2EE ロールが割り当てられているユーザーによって呼び出されたかどうか。
- 照会対象オブジェクト。オブジェクトのプロパティーを照会するために、事前定義データベース・ビューが提供されています。
- from 文節、結合条件、およびユーザー固有のアクセス制御条件の挿入。

照会には、カスタム・プロパティーと変数プロパティーの両方を含めることができます。複数のカスタム・プロパティーまたは変数プロパティーを照会に含める場合、対応するデータベース表で自己結合が行われます。データベース・システムによっては、これらの `query()` 呼び出しによりパフォーマンスへの影響が出ることがあります。

また、createStoredQuery メソッドを使用して、Business Process Choreographer データベースに照会を保管することもできます。保管照会文を定義する場合は、照会基準を指定します。この基準は、保管照会文が実行されるとき、すなわち実行時にデータがアセンブルされるときに動的に適用されます。保管照会文にパラメーターが含まれている場合は、照会を実行するときにこれらのパラメーターも解決されます。

Business Process Choreographer API について詳しくは、プロセス関連メソッドの com.ibm.bpe.api パッケージおよびタスク関連メソッドの com.ibm.task.api パッケージ内にある Javadoc を参照してください。

## API query メソッドの構文

Business Process Choreographer API の照会の構文は、SQL 照会の構文に似ています。照会には、select 文節、where 文節、order-by 文節、スキップ・タプル・パラメーター、しきい値パラメーター、および時間帯パラメーターを組み込むことができます。

照会の構文はオブジェクト・タイプによって異なります。以下の表は、異なるオブジェクト・タイプごとの構文を示しています。

*表 6.*

| オブジェクト | 構文 |
|---|---|
| プロセス・テンプレート | `ProcessTemplateData[] queryProcessTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);` |
| タスク・テンプレート | `TaskTemplate[] queryTaskTemplates (java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer threshold, java.util.TimeZone timezone);` |
| ビジネス・プロセスとタスク関連データ | `QueryResultSet query (java.lang.String selectClause, java.lang.String whereClause, java.lang.String orderByClause, java.lang.Integer skipTuples java.lang.Integer threshold, java.util.TimeZone timezone);` |

**select 文節:**

照会関数の select 文節は、照会によって戻されるオブジェクト・プロパティーを示します。

select 文節は、照会結果を記述します。これは、戻すオブジェクト・プロパティー (結果の列) を識別する名前のリストを指定します。構文は SQL SELECT 文節の構文と似ており、コンマを使用して文節のパーツを区切ります。文節の各パーツは、事前定義されているビューのいずれか 1 つの列を指定する必要があります。列は、ビュー名と列名を使用して完全に指定される必要があります。QueryResultSet オブジェクトで戻される列は、select 文節で指定されている列と同じ順序で表示されます。

select 文節は、AVG()、SUM()、MIN()、または MAX() などの SQL 集約関数はサポートしていません。

複数の名前と値の対のプロパティー (カスタム・プロパティーや、照会可能な変数のプロパティーなど) を選択する場合は、ビュー名に 1 桁のカウンターを追加します。このカウンターは 1 から 9 の値を取ることができます。

**select 文節の例**

- ″WORK_ITEM.OBJECT_TYPE, WORK_ITEM.REASON″

  関連オブジェクトのオブジェクト・タイプ、および作業項目の割り当て理由を取得します。

- ″DISTINCT WORK_ITEM.OBJECT_ID″

  呼び出し元が作業項目を所有しているオブジェクトの ID すべてを重複なしで取得します。
- ″ACTIVITY.TEMPLATE_NAME, WORK_ITEM.REASON″

  呼び出し元が作業項目を所有しているアクティビティーの名前、およびその割り当て理由を取得します。
- ″ACTIVITY.STATE, PROCESS_INSTANCE.STARTER″

  アクティビティーの状態、およびその関連プロセス・インスタンスのスターターを取得します。
- ″DISTINCT TASK.TKIID, TASK.NAME″

  呼び出し元が作業項目を所有しているタスクの ID と名前すべてを重複なしで取得します。
- ″TASK_CPROP1.STRING_VALUE, TASK_CPROP2.STRING_VALUE″

  さらに where 文節でも指定されているカスタム・プロパティーの値を取得します。
- ″QUERY_PROPERTY1.STRING_VALUE, QUERY_PROPERTY2.INT_VALUE

  照会できる変数のプロパティーの値を取得します。これらの部分は、さらに where 文節でも指定されています。
- ″COUNT( DISTINCT TASK.TKIID)″

  where 文節の条件を満たす固有のタスクの作業項目の数を数えます。

**where 文節:**

照会関数の中の where 文節は、照会ドメインに適用するフィルター基準を記述します。

where 文節の構文は、SQL WHERE 文節の構文に似ています。文節から明示的に SQL を追加したり、API where 文節に述部を結合したりする必要はありません。これらの構成要素は照会の実行時に自動的に追加されます。フィルター基準を適用しない場合は、where 文節に `null` を指定する必要があります。

where 文節構文は以下のものをサポートします。
- キーワード: AND、OR、NOT
- 比較演算子: =、<=、<、<>、>、>=、LIKE

  LIKE 操作では、照会されるデータベースに定義されているワイルドカード文字がサポートされます。
- 設定操作: IN

以下の規則も適用されます。
- オブジェクト ID 定数を `ID('string-rep-of-oid')` に指定します。
- `BIN('UTF-8 string')` としてバイナリー定数を指定します。

- 整数列挙型の代わりにシンボリック定数を使用します。例えば、アクティビティー状態式 ACTIVITY.STATE=2 を指定する代わりに、ACTIVITY.STATE=ACTIVITY.STATE.STATE_READY を指定します。
- 比較ステートメント内のプロパティーの値に単一引用符 (') が含まれる場合、例えば "TASK_CPROP.STRING_VALUE='d''automatisation'" のように、引用符を二重にしてください。
- ビュー名に 1 桁のサフィックスを追加して、複数の名前と値の対のプロパティー (カスタム・プロパティーなど) を参照します。例: "TASK_CPROP1.NAME='prop1' AND "TASK_CPROP2.NAME='prop2'"
- タイム・スタンプ定数を TS('yyyy-mm-ddThh:mm:ss') に指定します。現在日付を参照するには、タイム・スタンプを CURRENT_DATE に指定します。

  タイム・スタンプには、最低でも日付または時間の値を指定する必要があります。
  - 日付のみを指定すると、時間値はゼロに設定されます。
  - 時間のみを指定すると、日付は現在の日付に設定されます。
  - 日付を指定する場合、年は 4 桁の定数で構成する必要があります。月および日の値はオプションです。欠落している月および日の値は、01 に設定されます。例えば、TS('2003') は TS('2003-01-01T00:00:00') と同じです。
  - 日付を指定する場合、この値は 24 時間制で記述されます。例えば、現在日付が 2003 年 1 月 1 日の場合、TS('T16:04') または TS('16:04') は、TS('2003-01-01T16:04:00') と同じです。

**where 文節の例**

- オブジェクト ID と既存の ID の比較

  ```
  "WORK_ITEM.WIID = ID('_WI:800c00ed.df8d7e7c.feffff80.38')"
  ```

  この型の where 文節は、通常、直前の呼び出しの既存オブジェクト ID を使用して、動的に作成されます。このオブジェクト ID が *wiid1* 変数に保管されている場合、文節の構文は次のようになります。

  ```
  "WORK_ITEM.WIID = ID('" + wiid1.toString() + "')"
  ```

- タイム・スタンプの使用

  ```
  "ACTIVITY.STARTED >= TS('2002-06-1T16.00.00')"
  ```

- シンボリック定数の使用

  ```
  "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER"
  ```

- ブール値 true および false の使用

  ```
  "ACTIVITY.BUSINESS_RELEVANCE = TRUE"
  ```

- カスタム・プロパティーの使用

  ```
  "TASK_CPROP1.NAME = 'prop1' AND " TASK_CPROP1.STRING_VALUE = 'v1' AND
   TASK_CPROP2.NAME = 'prop2' AND " TASK_CPROP2.STRING_VALUE = 'v2'"
  ```

**order-by 文節:**

照会関数内の order-by 文節は、照会結果セットのソート基準を指定します。

結果をソートするビューの列のリストを指定できます。これらの列は、ビューと列の名前で完全に修飾されている必要があります。select 文節に含まれている列を指定することをお勧めします。

order-by 文節の構文は、SQL の order-by 文節の構文と似ており、文節の各パーツをコンマで区切ります。列を昇順にソートする場合は ASC を指定し、列を降順にソートする場合は DESC を指定します。照会結果セットをソートしない場合は、order-by 文節で null を指定する必要があります。

ソート基準はサーバーに適用されます。つまり、ソートにサーバーのロケールが使用されます。複数の列を指定すると、照会結果セットはまず最初の列の値で順序付けされ、次に 2 番目の列の値で順序付けされる、という具合に続きます。SQL 照会のように、order-by 文節の列を、位置によって指定することはできません。

**order-by 文節の例**

- ″PROCESS_TEMPLATE.NAME″

  照会結果を、プロセス・テンプレート名でアルファベット順にソートします。
- ″PROCESS_INSTANCE.CREATED, PROCESS_INSTANCE.NAME DESC″

  照会結果を作成日でソートし、特定の日付の場合はその結果を、プロセス・インスタンス名でアルファベット順の逆順にソートします。
- ″ACTIVITY.OWNER, ACTIVITY.TEMPLATE_NAME, ACTIVITY.STATE″

  照会結果を、アクティビティー所有者、アクティビティー・テンプレート名、アクティビティーの状態の順でソートします。

**スキップ・タプル・パラメーター:**

スキップ・タプル・パラメーターは、無視して照会結果セットで呼び出し元に戻さない照会結果セット・タプルの数を指定します。この数は、照会結果セットの先頭から数えます。

このパラメーターをしきい値パラメーターと一緒に使用してクライアント・アプリケーションでページングをインプリメントします (例えば、最初の 20 項目を取得し、それから次の 20 項目を取得し、以下同様に項目を取得します)。

このパラメーターを null に設定したときに、しきい値パラメーターを設定していないと、すべての適格なタプルが戻されます。

**スキップ・タプル・パラメーターの例**

- new Integer(5)

  最初の 5 つの適格なタプルを戻さないように指定します。

**しきい値パラメーター:**

照会関数のしきい値パラメーターは、照会の結果セットとしてサーバーからクライアントに戻されるオブジェクトの数を制限します。

実動シナリオでの照会結果セットには数千から数百万の項目が含まれる可能性があるので、常にしきい値を指定するのがベスト・プラクティスです。しきい値パラメ

ーターは、例えば少数の項目のみが一度に表示されるグラフィカル・ユーザー・インターフェースなどで有用な場合があります。しきい値パラメーターを適宜設定すると、データベース照会が高速化し、サーバーからクライアントへ転送する必要のあるデータが少なくなります。

このパラメーターを null に設定したときに、スキップ・タプル・パラメーターを設定していないと、適格なオブジェクトがすべて戻されます。

**しきい値パラメーターの例**

- new Integer(50)

  50 個の適格なタプルを戻すように指定します。

**時間帯パラメーター:**

照会関数の時間帯パラメーターは、照会内のタイム・スタンプ定数の時間帯を定義します。

照会を開始するクライアントと照会を処理するサーバーの間で、時間帯が異なることがあります。時間帯パラメーターを使用して、where 文節で使用されるタイム・スタンプ定数の時間帯を、例えば地方時を指定するように指定します。照会の結果セットで戻される日付には、照会で指定したものと同じ時間帯が設定されます。

このパラメーターを null に設定すると、タイム・スタンプ定数は協定世界時 (UTC) と想定されます。

**時間帯パラメーターの例**

- ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 (String)null,
                 (Integer)null,
                 java.util.TimeZone.getDefault() );
  ```

  2005 年 1 月 1 日の 17:40 地方時より後に開始されたアクティビティーのオブジェクト ID を戻します。
- ```
  process.query("ACTIVITY.AIID",
                "ACTIVITY.STARTED > TS('2005-01-01T17:40')",
                 (String)null, (Integer)null, (TimeZone)null);
  ```

  2005 年 1 月 1 日の 17:40 UTC より後に開始されたアクティビティーのオブジェクト ID を戻します。この指定は、例えば東部標準時より 6 時間早い時間です。

**保管照会文のパラメーター:**

保管照会文は、データベースに保管され、名前で識別される照会のことです。適格なタプルは、照会が実行されるときに動的にアセンブルされます。保管照会文を再使用可能にするには、実行時に解決される照会定義のパラメーターを使用できます。

例えば、顧客名を保管するカスタム・プロパティーを定義したとします。特定の顧客 ACME Co. に関連したタスクを戻すように、照会を定義することができます。この情報を照会する場合、照会内の where 文節は以下の例のようになります。

```
String whereClause =
  "TASK.STATE = TASK.STATE.STATE_READY
  AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
  AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = 'ACME Co.'";
```

顧客 BCME Ltd. も検索できるようにこの照会を再使用可能にするには、カスタム・プロパティーの値に対してパラメーターを使用できます。パラメーターをタスク照会に追加する場合、以下の例のようになります。

```
String whereClause =
  "TASK.STATE = TASK.STATE.STATE_READY
  AND WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER
  AND TASK_CPROP.NAME = 'company' AND TASK_CPROP.STRING_VALUE = '@param1'";
```

`@param1` パラメーターは、query メソッドに受け渡されるパラメーターのリストから、実行時に解決されます。以下の規則は、照会内でのパラメーターの使用に適用されます。

- パラメーターは where 文節でのみ使用できる。
- パラメーターはストリングである。
- パラメーターは実行時にストリングの置換を使用して置き換えられる。特殊文字が必要な場合、where 文節に指定するか、実行時にパラメーターの一部として受け渡す必要があります。
- パラメーター名は、ストリング `@param` を整数と連結したもので構成される。最小番号は 1 で、実行時に照会 API に受け渡されるパラメーターのリスト内の最初の項目を指します。
- パラメーターは where 文節内で複数回使用することができ、出現するすべてのパラメーターは同じ値で置き換えられます。

**関連タスク**

228 ページの『保管照会文の管理』
保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

**照会結果:**

照会結果セットには、Business Process Choreographer API 照会の結果が入ります。

結果セットのエレメントは、呼び出し元により指定された where 文節の条件を満たし、かつ呼び出し元に対し表示が許可されているオブジェクトのプロパティーです。エレメントは、API の次のメソッドを使用して相対的に読み取るか、あるいは最初と最後のメソッドを使用して絶対的に読み取ります。照会結果セットの暗黙カーソルは初めは最初のエレメントの前に配置されるため、エレメントを読み取る前に、最初または次のメソッドのいずれかを呼び出す必要があります。 size メソッドを使用して、セット内のエレメント数を判別することができます。

照会結果セットのエレメントは、作業項目とそれに関連する参照オブジェクト (アクティビティー・インスタンスやプロセス・インスタンスなど) の選択済み属性を構成します。 QueryResultSet エレメントの最初の属性 (列) は、照会要求の select 文節で指定されている最初の属性の値を指定します。 QueryResultSet エレメントの 2 番目の属性 (列) は、照会要求の select 文節で指定されている 2 番目の属性の値を指定する、という具合に続きます。

属性の値は、その属性タイプと互換性のあるメソッドを呼び出すことによって、また、適切な列インデックスを指定することによって、取得することができます。列インデックスの番号付けは 1 から始まります。

| 属性タイプ | メソッド |
|---|---|
| String | getString |
| OID | getOID |
| Timestamp | getTimestamp<br>getString<br>getTimestampAsLong |
| Integer | getInteger<br>getShort<br>getLong<br>getString<br>getBoolean |
| Boolean | getBoolean<br>getShort<br>getInteger<br>getLong<br>getString |
| byte[] | getBinary |

**例:**

以下の照会が実行されます。

```
QueryResultSet resultSet = process.query("ACTIVITY.STARTED,
                                         ACTIVITY.TEMPLATE_NAME AS NAME,
                                         WORK_ITEM.WIID, WORK_ITEM.REASON",
                                         (String)null, (String)null,
                                         (Integer)null, (TimeZone)null);
```

戻される照会結果セットには、以下の 4 つの列があります。
- 列 1 はタイム・スタンプ
- 列 2 はストリング
- 列 3 はオブジェクト ID
- 列 4 は整数

以下のメソッドを使用して、属性値を取得することができます。

```
while (resultSet.next())
{
 java.util.Calendar activityStarted = resultSet.getTimestamp(1);
 String templateName = resultSet.getString(2);
 WIID wiid = (WIID) resultSet.getOID(3);
 Integer reason = resultSet.getInteger(4);
}
```

結果セットの表示名を、例えば、印刷されるテーブルの見出しなどに使用することができます。これらの名前は、ビューの列名、または照会の AS 文節で定義された名前です。以下のメソッドを使用して、例中の表示名を取得することができます。

```
resultSet.getColumnDisplayName(1) returns "STARTED"
resultSet.getColumnDisplayName(2) returns "NAME"
resultSet.getColumnDisplayName(3) returns "WIID"
resultSet.getColumnDisplayName(4) returns "REASON"
```

## ユーザー固有のアクセス条件

SQL SELECT ステートメントが API 照会から生成されるときに、ユーザー固有のアクセス条件が追加されます。この条件により、呼び出し元により指定されている条件に一致し、呼び出し元に対し許可されているオブジェクトのみが呼び出し元に戻されます。

追加されるアクセス条件は、ユーザーがシステム管理者であるかどうかによって異なります。

### システム管理者以外のユーザーが呼び出した照会

生成される SQL WHERE 文節では、API where 文節と、ユーザー固有のアクセス制御条件が結合されます。この照会は、ユーザーに対しアクセスが許可されているオブジェクト、つまりユーザーが作業項目を所有しているオブジェクトのみを取得します。作業項目とは、ビジネス・オブジェクト (タスクやプロセスなど) の許可ロールへのユーザーまたはユーザー・グループの割り当てを表します。例えば、ユーザー John Smith が特定のタスクの潜在的所有者ロールのメンバーである場合、この関係を表す作業項目オブジェクトがあります。

例えば、グループ作業項目が無効な場合に、システム管理者以外のユーザーがタスクを照会すると、以下のアクセス条件が WHERE 文節に追加されます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'user'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

したがって John Smith が、自身が潜在的所有者であるタスクのリストを取得する場合、API where 文節は次のようになります。

```
"WORK_ITEM.REASON == WORK_ITEM.REASON.REASON_POTENTIAL_OWNER"
```

この API where 文節により、SQL ステートメントに次のアクセス条件が追加されます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
AND ( WI.OWNER_ID = 'JohnSmith'
      OR WI.OWNER_ID = null AND WI.EVERYBODY = true)
AND WI.REASON = 1
```

つまり、John Smith が、自身がプロセス・リーダーまたはプロセス管理者であるアクティビティーやタスクと、作業項目を所有していないアクティビティーやタスクを表示するには、 PROCESS_INSTANCE ビューのプロパティー (PROCESS_INSTANCE.PIID など) を照会の select、where、または order-by 文節に追加する必要があります。

グループ作業項目が有効な場合は、ユーザーに対し、グループがアクセスできるオブジェクトへのアクセスを許可するアクセス条件が WHERE 文節に追加されます。

### システム管理者が呼び出した照会

システム管理者は、query メソッドを呼び出し、作業項目が関連付けられているオブジェクトを取得できます。この場合、生成される SQL 照会には WORK_ITEM ビューとの結合が追加されますが、WORK_ITEM.OWNER_ID のアクセス制御条件は追加されません。

この場合、タスクの SQL 照会には以下が含まれます。

```
FROM TASK TA, WORK_ITEM WI
WHERE WI.OBJECT_ID = TA.TKIID
```

### queryAll 照会

このタイプの照会を呼び出すことができるのは、システム管理者またはシステム・モニターのみです。アクセス制御条件も WORK_ITEM ビューとの結合も追加されません。このタイプの照会では、すべてのオブジェクトの全データが戻されます。

## query メソッドと queryAll メソッドの例

以下の例は、標準的な各種 API 照会の構文と、照会の実行時に生成される関連 SQL ステートメントを示します。

**例: 作動可能状態のタスクの照会:**

この例では、query メソッドを使用して、ログオン・ユーザーが作業可能なタスクを取得する方法を示します。

John Smith は、自分に割り当てられているタスクを一覧表示します。ユーザーがタスクの作業を行うには、そのタスクが作動可能状態になっている必要があります。ログオン・ユーザーには、そのタスクの潜在的所有者作業項目も必要です。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT TASK.TKIID",
           "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
             AND " +
           "TASK.STATE = TASK.STATE.STATE_READY AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
            (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.KIND IN ( 101, 105 )
   AND    TA.STATE = 2
   AND    WI.REASON = 1
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND  WI.EVERYBODY = true )
```

API 照会を特定のプロセスのタスク (sampleProcess など) に限定する場合、この照会は次のようになります。

```
query( "DISTINCT TASK.TKIID",
           "PROCESS_TEMPLATE.NAME = 'sampleProcess' AND "+
           "TASK.KIND IN ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING )
             AND " +
           "TASK.STATE = TASK.STATE.STATE_READY AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
            (String)null, (String)null, (Integer)null, (TimeZone)null )
```

**例: 要求済み状態のタスクの照会:**

この例では、query メソッドを使用して、ログオン・ユーザーが要求したタスクを取得する方法を示します。

ユーザー John Smith は、自身が要求したタスクのうち、まだ要求済み状態であるタスクを検索します。「John Smith により要求された」ことを指定する条件は `TASK.OWNER = 'JohnSmith'` です。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT TASK.TKIID",
           "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
           "TASK.OWNER = 'JohnSmith'",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.STATE = 8
   TA.OWNER = 'JohnSmith'
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

タスクが要求されると、タスクの所有者に対して作業項目が作成されます。したがって、John Smith が要求したタスクを取得する照会のもう 1 つの作成方法として、`TASK.OWNER = 'JohnSmith'` を使用する代わりに、次の条件を照会に追加する方法があります。

```
WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER
```

照会は、次のコード・スニペットのようになります。

```
query( "DISTINCT TASK.TKIID",
           "TASK.STATE = TASK.STATE.STATE_CLAIMED AND " +
           "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_OWNER",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (`TASK.STATE.STATE_READY` など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  WI.OBJECT_ID = TA.TKIID
   AND    TA.STATE = 8
   AND    WI.REASON = 4
   AND  ( WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID = null AND WI.EVERYBODY = true )
```

John は休暇に入るため、所属するチームのリーダーである Anne Grant が、John の現在の作業割り当てを確認するとします。Anne にはシステム管理者権限が付与されています。呼び出す照会は、John が呼び出した照会と同じです。ただし Anne は管理者であるため、生成される SQL ステートメントが異なります。生成される SQL ステートメントを次のコード・スニペットに示します。

```
SELECT DISTINCT TASK.TKIID
   FROM   TASK TA, WORK_ITEM WI,
   WHERE  TA.TKIID = WI.OBJECT_ID =
   AND    TA.STATE = 8
   AND   TA.OWNER = 'JohnSmith')
```

Anne は管理者であるため、アクセス制御条件が WHERE 文節に追加されません。

**例: エスカレーションの照会:**

この例では、query メソッドを使用して、ログオン・ユーザーのエスカレーションを取得する方法を示します。

タスクがエスカレートされると、エスカレーション受信者作業項目が作成されます。ユーザー Mary Jones が、Mary 自身にエスカレートされたタスクのリストを表示するとします。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query( "DISTINCT ESCALATION.ESIID, ESCALATION.TKIID",
         "WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_ESCALATION_RECEIVER",
           (String)null, (String)null, (Integer)null, (TimeZone)null )
```

SQL SELECT ステートメントの生成時には、次のアクションが実行されます。

- アクセス制御条件が where 文節に追加されます。この例では、グループ作業項目が有効でないことが想定されています。
- 定数 (TASK.STATE.STATE_READY など) が、数値に置き換えられます。
- FROM 文節と結合条件が追加されます。

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
 SELECT DISTINCT ESCALATION.ESIID, ESCALATION.TKIID
   FROM   ESCALATION ESC, WORK_ITEM WI
   WHERE  ESC.ESIID = WI.OBJECT_ID
   AND    WI.REASON  = 10
   AND
  ( WI.OWNER_ID = 'MaryJones' OR WI.OWNER_ID = null AND  WI.EVERYBODY = true )
```

**例: queryAll メソッドの使用:**

この例では、queryAll メソッドを使用して、1 つのプロセス・テンプレートに属するアクティビティーをすべて取得する方法を示します。

queryAll メソッドは、システム管理者権限またはシステム・モニター権限が付与されているユーザーだけが使用できます。プロセス・テンプレート sampleProcess に属するすべてのアクティビティーを取得する照会の queryAll メソッド呼び出しを、次のコード・スニペットに示します。

```
queryAll( "DISTINCT ACTIVITY.AIID",
       "PROCESS_TEMPLATE.NAME = 'sampleProcess'",
       (String)null, (String)null, (Integer)null, (TimeZone)null )
```

この API 照会から生成される SQL 照会を、次のコード・スニペットに示します。

```
SELECT DISTINCT ACTIVITY.AIID
  FROM   ACTIVITY AI, PROCESS_TEMPLATE PT
  WHERE  AI.PTID = PT.PTID
  AND    PT.NAME = 'sampleProcess'
```

この呼び出しは管理者により実行されるため、生成される SQL ステートメントにアクセス制御条件は追加されません。WORK_ITEM ビューとの結合も追加されません。つまり、この照会では、プロセス・テンプレートのすべてのアクティビティー (作業項目のないアクティビティーを含む) が取得されます。

**例: 照会への照会プロパティーの組み込み:**

この例では、query メソッドを使用して、ビジネス・プロセスに属するタスクを取得する方法を示します。このプロセスに対して定義されている照会プロパティーを、検索に組み込むとします。

例えば、1 つのビジネス・プロセスに属し、作動可能状態にあるヒューマン・タスクをすべて検索するとします。プロセスには照会プロパティー **customerID** とその値 CID_12345、および名前空間があります。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
                   PROCESS_INSTANCE.NAME",
          " QUERY_PROPERTY.NAME = 'customerID' AND " +
          " QUERY_PROPERTY.STRING_VALUE = 'CID_12345' AND " +
          " QUERY_PROPERTY.NAMESPACE =
             'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

2 番目の照会プロパティー (**Priority** など) と特定の名前空間を照会に追加する場合、照会の query メソッド呼び出しは次のようになります。

```
query ( " DISTINCT TASK.TKIID, TASK_TEMPL.NAME, TASK.STATE,
                   PROCESS_INSTANCE.NAME",
          " QUERY_PROPERTY1.NAME = 'customerID' AND " +
          " QUERY_PROPERTY1.STRING_VALUE = 'CID_12345' AND " +
          " QUERY_PROPERTY1.NAMESPACE =
                 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " QUERY_PROPERTY2.NAME = 'Priority' AND " +
          " QUERY_PROPERTY2.NAMESPACE =
                 'http://www.ibm.com/xmlns/prod/websphere/mqwf/bpel/' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

複数の照会プロパティーを照会に追加する場合は、コード・スニペットに示されているように、追加する各プロパティーに番号を付ける必要があります。ただし、カスタム・プロパティーの照会を実行するとパフォーマンスに影響します。照会に含まれているカスタム・プロパティーの数に応じてパフォーマンスが低下します。

**例: 照会へのカスタム・プロパティーの組み込み:**

この例では、query メソッドを使用して、カスタム・プロパティーが指定されたタスクを取得する方法を示します。

例えば、カスタム・プロパティー **customerID** とその値 CID_12345 が指定されており、作動可能状態にあるヒューマン・タスクをすべて検索するとします。この照会の query メソッド呼び出しを、次のコード・スニペットに示します。

```
query ( " DISTINCT TASK.TKIID ",
          " TASK_CPROP.NAME = 'customerID' AND " +
          " TASK_CPROP.STRING_VALUE = 'CID_12345' AND " +
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

タスクとそのカスタム・プロパティーを取得する場合、照会の query メソッド呼び出しは次のようになります。

```
query ( " DISTINCT TASK.TKIID, TASK_CPROP.NAME, TASK_CPROP.STRING_VALUE",
          " TASK.KIND IN
             ( TASK.KIND.KIND_HUMAN, TASK.KIND.KIND_PARTICIPATING ) AND " +
          " TASK.STATE = TASK.STATE.STATE_READY ",
          (String)null, (String)null, (Integer)null, (TimeZone)null );
```

この API 照会から生成される SQL ステートメントを、次のコード・スニペットに示します。

```
SELECT DISTINCT TA.TKIID , TACP.NAME , TACP.STRING_VALUE
   FROM  TASK TA LEFT JOIN TASK_CPROP TACP ON (TA.TKIID = TACP.TKIID),
         WORK_ITEM WI
   WHERE WI.OBJECT_ID = TA.TKIID
   AND    TA.KIND IN ( 101, 105 )
   AND    TA.STATE = 2
   AND  (WI.OWNER_ID = 'JohnSmith' OR WI.OWNER_ID IS NULL AND WI.EVERYBODY = 1 )
```

この SQL ステートメントには、TASK ビューと TASK_CPROP ビューの外部結合が含まれています。つまり、WHERE 文節の条件を満たすタスクは、カスタム・プロパティーが含まれていない場合でも取得されます。

# ビジネス・プロセスおよびヒューマン・タスク用 EJB クライアント・アプリケーションの開発

EJB API は、WebSphere Process Server 上にインストールされているビジネス・プロセスやヒューマン・タスクを処理する EJB クライアント・アプリケーションを開発するための汎用的な方法をいくつか提供します。

## このタスクについて

この Enterprise JavaBeans (EJB) API を使用すれば、以下を実行するためのクライアント・アプリケーションを作成できます。

- プロセスやタスクのライフ・サイクルの、開始から完了後の削除までの管理
- アクティビティーやプロセスの修復
- ワークグループのメンバーに対するワークロードの管理および配布

EJB API は、次の 2 種類のステートレス・セッション・エンタープライズ Bean として提供されます。

- BusinessFlowManagerService インターフェースは、ビジネス・プロセス・アプリケーション用のメソッドを備えています。
- HumanTaskManagerService インターフェースは、タスク・ベースのアプリケーション用のメソッドを備えています。

EJB API の詳細は、com.ibm.bpe.api パッケージおよび com.ibm.task.api パッケージの中の Javadoc を参照してください。

以下のステップは、EJB アプリケーションの開発に必要なアクションの概要です。

**手順**

1. アプリケーションが提供する機能を決定します。
2. 使用するセッション Bean を決定します。

   アプリケーションでインプリメントするシナリオに応じて、2 つのセッション Bean のうちの 1 つ、または両方を使用することができます。
3. アプリケーションのユーザーが必要とする許可権限を判別します。

   アプリケーションのユーザーには、アプリケーションに組み込まれたメソッドを呼び出し、これらのメソッドが戻すオブジェクトとそのオブジェクトの属性を表示するための、適切な許可のロールが割り当てられている必要があります。該当するセッション Bean のインスタンスを作成するときに、WebSphere Application Server がコンテキストとそのインスタンスを関連付けます。コンテキストには、呼び出し元のプリンシパル ID、グループ・メンバーシップ・リスト、およびロールについての情報が含まれています。この情報は、それぞれの呼び出しごとに、呼び出し元の権限を確認するために使用されます。

   Javadoc には、各メソッドの許可情報が含まれています。
4. アプリケーションをレンダリングする方法を決めます。

   EJB API は、ローカル側でもリモート側でも呼び出すことができます。
5. アプリケーションを開発します。
   a. EJB API にアクセスします。
   b. EJB API を使用して、プロセスまたはタスクと対話します。
      - データを照会します。
      - データで作業を行います。

**関連概念**

187 ページの『ビジネス・プロセスおよびヒューマン・タスクと対話するためのプログラミング・インターフェースの比較』
ビジネス・プロセスおよびヒューマン・タスクと対話するクライアント・アプリケーションの作成には、Enterprise JavaBeans (EJB)、Web サービス、Java

Message Service (JMS) および Representational State Transfer (REST) サービスなどの汎用プログラミング・インターフェースを使用できます。これらのインターフェースには、それぞれ異なる特性があります。

**関連資料**

Business Process Choreographer のデータベース・ビュー
この参照情報では、事前定義データベース・ビューの列について説明します。

# EJB API へのアクセス

Enterprise JavaBeans (EJB) API は、次の 2 種類のステートレス・セッション・エンタープライズ Bean として提供されます。ビジネス・プロセス・アプリケーションおよびタスク・アプリケーションは、Bean のホーム・インターフェースを介して、適切なセッション・エンタープライズ Bean にアクセスします。

## このタスクについて

BusinessFlowManagerService インターフェースは、ビジネス・プロセス・アプリケーション用のメソッドを備え、HumanTaskManagerService インターフェースは、タスク・ベースのアプリケーション用のメソッドを備えています。このアプリケーションは、別の Enterprise JavaBeans (EJB) アプリケーションを含む任意の Java アプリケーションを指します。

## セッション Bean のリモート・インターフェースにアクセスする

ビジネス・プロセスまたはヒューマン・タスク用の EJB クライアント・アプリケーションでは、Bean のリモート・ホーム・インターフェースを介して、セッション Bean のリモート・インターフェースにアクセスします。

### このタスクについて

セッション Bean は、プロセス・アプリケーションに対しては BusinessFlowManager セッション Bean、タスク・アプリケーションに対しては HumanTaskManager セッション Bean のいずれかである可能性があります。

**手順**

1. セッション Bean のリモート・インターフェースへの参照をアプリケーション・デプロイメント記述子に追加します。 参照を以下のファイルの 1 つに追加します。
   - Java 2 Platform Enterprise Edition (J2EE) クライアント・アプリケーションの場合は、application-client.xml ファイル
   - Web アプリケーションの場合は、web.xml ファイル
   - Enterprise JavaBeans (EJB) アプリケーションの場合は、ejb-jar.xml ファイル

   プロセス・アプリケーションの場合のリモート・ホーム・インターフェースへの参照は、以下の例で示されます。

```
<ejb-ref>
 <ejb-ref-name>ejb/BusinessFlowManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
 <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
</ejb-ref>
```

タスク・アプリケーションの場合のリモート・ホーム・インターフェースへの参照は、以下の例で示されます。

```
<ejb-ref>
 <ejb-ref-name>ejb/HumanTaskManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <home>com.ibm.task.api.HumanTaskManagerHome</home>
 <remote>com.ibm.task.api.HumanTaskManager</remote>
</ejb-ref>
```

WebSphere Integration Developer を使用して EJB 参照をデプロイメント記述子に追加する場合、EJB 参照のバインディングが、アプリケーションのデプロイ時に自動的に作成されます。EJB 参照の追加について詳しくは、WebSphere Integration Developer の文書を参照してください。

2. 生成されたスタブをアプリケーションにパッケージします。
   a. プロセス・アプリケーションの場合、<*install_root*>/ProcessChoreographer/client/bpe137650.jar ファイルを、ご使用のアプリケーションのエンタープライズ・アーカイブ (EAR) ファイルにパッケージします。
   b. タスク・アプリケーションの場合、<*install_root*>/ProcessChoreographer/client/task137650.jar ファイルを、ご使用のアプリケーションの EAR ファイルにパッケージします。
   c. アプリケーション・モジュールのマニフェスト・ファイル内の **Classpath** パラメーターを、JAR ファイルを含めるように設定します。

      アプリケーション・モジュールは、J2EE アプリケーション、Web アプリケーション、または EJB アプリケーションの可能性があります。
3. ビジネス・オブジェクトの定義をどのように提供するかを決定します。

   リモート・クライアント・アプリケーション内のビジネス・オブジェクトを処理する場合は、プロセスまたはタスクとの対話に使用されるビジネス・オブジェクトの対応スキーマ (XSD または WSDL ファイル) にアクセスできる必要があります。これらのファイルにアクセスできるようにするには、次のいずれかの方法を行います。

   - クライアント・アプリケーションが管理対象 J2EE 環境で稼働しない場合は、これらのファイルをクライアント・アプリケーションの EAR ファイルにパッケージします。
   - クライアント・アプリケーションが管理対象 J2EE 環境の Web アプリケーションまたは EJB クライアントである場合は、これらのファイルをクライアント・アプリケーションの EAR ファイルにパッケージするか、リモート成果物ロードを利用します。
     a. Business Process Choreographer EJB API の createMessage および ClientObjectWrapper.getObject メソッドを使用して、サーバー上の対応アプリケーションからリモート・ビジネス・オブジェクト定義を透過的にロードします。
     b. サービス・データ・オブジェクトのプログラミング API を使用して、既にインスタンス化されたビジネス・オブジェクトの一部としてビジネス・オブジェクトの作成または読み取りを実行します。これを行うには、DataObject インターフェースで commonj.sdo.DataObject.createDataObject メソッドまたは getDataObject メソッドを使用します。

c. XML スキーマ any または anyType を使用して型付けされるビジネス・オブジェクトのプロパティー値としてビジネス・オブジェクトを作成する場合は、ビジネス・オブジェクト・サービスを使用してビジネス・オブジェクトの作成または読み取りを実行します。これを行うには、スキーマのロード元となるアプリケーションを指すようにリモート成果物ローダーのコンテキストを設定する必要があります。これにより、適切なビジネス・オブジェクト・サービスを使用できるようになります。

例えば、ビジネス・オブジェクトを作成します。ここで ″ApplicationName″ は、ビジネス・オブジェクト定義が含まれるアプリケーションの名前です。

```
BOFactory bofactory = (BOFactory) new
   ServiceManager().locateService("com/ibm/websphere/bo/BOFactory");

com.ibm.wsspi.al.ALContext.setContext
   ("RALTemplateName", "ApplicationName");
try {
     DataObject dataObject = bofactory.create("uriName", "typeName" );
   } finally {
     com.ibm.wsspi.al.ALContext.unset();
   }
```

例えば、XML 入力を読み取ります。ここで ″ApplicationName″ は、ビジネス・オブジェクト定義が含まれるアプリケーションの名前です。

```
BOXMLSerializer serializerService =
  (BOXMLSerializer) new ServiceManager().locateService
    ("com/ibm/websphere/bo/BOXMLSerializer");
ByteArrayInputStream input = new ByteArrayInputStream(<?xml?>..);

com.ibm.wsspi.al.ALContext.setContext
          ("RALTemplateName", "ApplicationName");
try {
     BOXMLDocument document = serializerService.readXMLDocument(input);
     DataObject dataObject = document.getDataObject();
   } finally {
     com.ibm.wsspi.al.ALContext.unset();
   }
```

4. Java Naming and Directory Interface (JNDI) からセッション Bean のリモート・ホーム・インターフェースを見つけます。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
// Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

// Lookup the remote home interface of the BusinessFlowManager bean
Object result =
       initialContext.lookup("java:comp/env/ejb/BusinessFlowManagerHome");

// Convert the lookup result to the proper type
   BusinessFlowManagerHome processHome =
          (BusinessFlowManagerHome)javax.rmi.PortableRemoteObject.narrow
          (result,BusinessFlowManagerHome.class);
```

セッション Bean のリモート・ホーム・インターフェースには、EJB オブジェクトの create メソッドが含まれます。 このメソッドは、セッション Bean のリモート・インターフェースを戻します。

5. セッション Bean のリモート・インターフェースにアクセスします。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
BusinessFlowManager process = processHome.create();
```

セッション Bean へのアクセス権は、呼び出し元が Bean が提供するすべてのアクションを実行できることを保証するものではありません。呼び出し元には、そのアクションに対する許可も必要になります。セッション Bean のインスタンスが作成されると、コンテキストはセッション Bean のそのインスタンスと関連付けられます。コンテキストは、呼び出し元のプリンシパル ID とグループ・メンバーシップ・リストを含み、呼び出し元が Business Process Choreographer J2EE のロールの 1 つを持っているかどうかを示します。このコンテキストを使用して、管理セキュリティーが設定されていない場合でも、呼び出しごとに呼び出し元の権限を確認します。管理セキュリティーが設定されていない場合、呼び出し元のプリンシパル ID の値は UNAUTHENTICATED になります。

6. サービス・インターフェースによって公開されたビジネス関数を呼び出します。

以下の例では、プロセス・アプリケーションでのこのステップを示します。

```
process.initiate("MyProcessModel",input);
```

アプリケーションからの呼び出しは、トランザクションとして実行されます。トランザクションは、以下のいずれかの方法で確立されて終了します。

- WebSphere Application Server から自動的に (デプロイメント記述子が TX_REQUIRED を指定)。
- アプリケーションから明示的に。アプリケーションの呼び出しを 1 つのトランザクションにバンドルすることができます。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

// Begin a transaction
transaction.begin();

// Applications calls ...

// On successful return, commit the transaction
transaction.commit();
```

**ヒント:** データベース・ロック競合を防ぐには、並列で以下のようなステートメントの実行を避けるようにします。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

getActivityInstance メソッドおよびその他の読み取り操作は、読み取りロックを設定します。この例では、アクティビティー・インスタンス上の読み取りロックは、アクティビティー・インスタンス上の更新ロックにアップグレードされま

す。これにより、トランザクションが並列で実行されるときに、データベース・デッドロックが発生することがあります。

### 例

以下に、ステップ 3 から 5 でタスク・アプリケーションを探す方法の例を示します。

```
//Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

//Lookup the remote home interface of the HumanTaskManager bean
Object result =
         initialContext.lookup("java:comp/env/ejb/HumanTaskManagerHome");

//Convert the lookup result to the proper type
HumanTaskManagerHome taskHome =
           (HumanTaskManagerHome)javax.rmi.PortableRemoteObject.narrow
           (result,HumanTaskManagerHome.class);

...
//Access the remote interface of the session bean.
HumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

## セッション Bean のローカル・インターフェースにアクセスする

ビジネス・プロセスまたはヒューマン・タスク用の EJB クライアント・アプリケーションでは、Bean のローカル・ホーム・インターフェースを介して、セッション Bean のローカル・インターフェースにアクセスします。

### このタスクについて

セッション Bean は、プロセス・アプリケーションに対しては BusinessFlowManager セッション Bean、ヒューマン・タスク・アプリケーションに対しては HumanTaskManager セッション Bean のいずれかである可能性があります。

### 手順

1. セッション Bean のローカル・インターフェースへの参照をアプリケーション・デプロイメント記述子に追加します。 参照を以下のファイルの 1 つに追加します。
   - Java 2 Platform Enterprise Edition (J2EE) クライアント・アプリケーションの場合は、application-client.xml ファイル
   - Web アプリケーションの場合は、web.xml ファイル
   - Enterprise JavaBeans (EJB) アプリケーションの場合は、ejb-jar.xml ファイル

   プロセス・アプリケーションの場合のローカル・ホーム・インターフェースへの参照は、以下の例で示されます。

   ```
   <ejb-local-ref>
    <ejb-ref-name>ejb/LocalBusinessFlowManagerHome</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
    <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
   </ejb-local-ref>
   ```

タスク・アプリケーションの場合のローカル・ホーム・インターフェースへの参照は、以下の例で示されます。

```
<ejb-local-ref>
 <ejb-ref-name>ejb/LocalHumanTaskManagerHome</ejb-ref-name>
 <ejb-ref-type>Session</ejb-ref-type>
 <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
 <local>com.ibm.task.api.LocalHumanTaskManager</local>
</ejb-local-ref>
```

WebSphere Integration Developer を使用して EJB 参照をデプロイメント記述子に追加する場合、EJB 参照のバインディングが、アプリケーションのデプロイ時に自動的に作成されます。EJB 参照の追加について詳しくは、WebSphere Integration Developer の文書を参照してください。

2. Java Naming and Directory Interface (JNDI) からセッション Bean のローカル・ホーム・インターフェースを見つけます。

   以下の例では、プロセス・アプリケーションでのこのステップを示します。

   ```
   // Obtain the default initial JNDI context
   InitialContext initialContext = new InitialContext();

   // Lookup the local home interface of the BusinessFlowManager bean

    LocalBusinessFlowManagerHome processHome =
         (LocalBusinessFlowManagerHome)initialContext.lookup
         ("java:comp/env/ejb/LocalBusinessFlowManagerHome");
   ```

   セッション Bean のローカル・ホーム・インターフェースには、EJB オブジェクトの create メソッドが含まれます。このメソッドは、セッション Bean のローカル・インターフェースを戻します。

3. セッション Bean のローカル・インターフェースにアクセスします。

   以下の例では、プロセス・アプリケーションでのこのステップを示します。

   ```
   LocalBusinessFlowManager process = processHome.create();
   ```

   セッション Bean へのアクセス権は、呼び出し元が Bean が提供するすべてのアクションを実行できることを保証するものではありません。呼び出し元には、そのアクションに対する許可も必要になります。セッション Bean のインスタンスが作成されると、コンテキストはセッション Bean のそのインスタンスと関連付けられます。コンテキストは、呼び出し元のプリンシパル ID とグループ・メンバーシップ・リストを含み、呼び出し元が Business Process Choreographer J2EE のロールの 1 つを持っているかどうかを示します。このコンテキストを使用して、管理セキュリティーが設定されていない場合でも、呼び出しごとに呼び出し元の権限を確認します。管理セキュリティーが設定されていない場合、呼び出し元のプリンシパル ID の値は UNAUTHENTICATED になります。

4. サービス・インターフェースによって公開されたビジネス関数を呼び出します。

   以下の例では、プロセス・アプリケーションでのこのステップを示します。

   ```
   process.initiate("MyProcessModel",input);
   ```

   アプリケーションからの呼び出しは、トランザクションとして実行されます。トランザクションは、以下のいずれかの方法で確立されて終了します。

- WebSphere Application Server から自動的に (デプロイメント記述子が TX_REQUIRED を指定)。
- アプリケーションから明示的に。アプリケーションの呼び出しを 1 つのトランザクションにバンドルすることができます。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

// Begin a transaction
transaction.begin();

// Applications calls ...

// On successful return, commit the transaction
transaction.commit();
```

**ヒント:** データベース・デッドロックを防ぐには、並列で以下のようなステートメントの実行を避けるようにします。

```
// Obtain user transaction interface
UserTransaction transaction=
       (UserTransaction)initialContext.lookup("java:comp/UserTransaction");

transaction.begin();

//read the activity instance
process.getActivityInstance(aiid);
//claim the activity instance
process.claim(aiid);

transaction.commit();
```

getActivityInstance メソッドおよびその他の読み取り操作は、読み取りロックを設定します。この例では、アクティビティー・インスタンス上の読み取りロックは、アクティビティー・インスタンス上の更新ロックにアップグレードされます。これにより、トランザクションが並列で実行されるときに、データベース・デッドロックが発生することがあります。

### 例

以下に、ステップ 2 から 4 でタスク・アプリケーションを探す方法の例を示します。

```
//Obtain the default initial JNDI context
InitialContext initialContext = new InitialContext();

//Lookup the local home interface of the HumanTaskManager bean
LocalHumanTaskManagerHome taskHome =
        (LocalHumanTaskManagerHome)initialContext.lookup
        ("java:comp/env/ejb/LocalHumanTaskManagerHome");

...
//Access the local interface of the session bean
LocalHumanTaskManager task = taskHome.create();

...
//Call the business functions exposed by the service interface
task.callTask(tkiid,input);
```

# ビジネス・プロセスおよびタスク関連のオブジェクトの照会

クライアント・アプリケーションは、ビジネス・プロセスとタスク関連オブジェクトを操作します。データベース内のビジネス・プロセス・オブジェクトおよびタスク関連のオブジェクトを照会して、これらのオブジェクトの特定のプロパティーを取得することができます。

## このタスクについて

Business Process Choreographer を構成すると、リレーショナル・データベースは、ビジネス・プロセス・コンテナーおよびタスク・コンテナーの両方に関連付けられます。このデータベースは、ビジネス・プロセスとタスクの管理用のすべてのテンプレート (モデル) とインスタンス (ランタイム) のデータを保管します。そのデータを照会するには、SQL 形式の構文を使用します。

単発の照会を実行して、オブジェクトの特定のプロパティーを取得することができます。また、頻繁に使用する照会を保管しておいて、この保管照会文をアプリケーションに組み込むこともできます。

**関連資料**

Business Process Choreographer のデータベース・ビュー
この参照情報では、事前定義データベース・ビューの列について説明します。

## 照会で変数を使用することによるデータのフィルタリング

照会結果は、照会基準に一致するオブジェクトを戻します。この結果を、変数の値でフィルタリングすることもできます。

## このタスクについて

実行時にプロセスが使用する変数を、そのプロセス・モデルで定義することができます。これらの変数で、照会可能なパートを宣言します。

例えば、John Smith が保険会社のサービス番号を呼び出して、損傷を受けた車に対する保険請求の進捗状況を問い合わせるとします。請求の管理者はカスタマー ID でその請求を検索します。

**手順**

1. オプション: プロセス内の照会可能な変数のプロパティーをリストします。

   プロセス・テンプレート ID を使用して、プロセスを特定します。照会可能な変数がわかっている場合は、このステップはスキップしてください。

   ```
   List variableProperties = process.getQueryProperties(ptid);
   for (int i = 0; i < variableProperties.size(); i++)
   {
      QueryProperty queryData = (QueryProperty)variableProperties.get(i);
      String variableName = queryData.getVariableName();
      String name         = queryData.getName();
      int mappedType      = queryData.getMappedType();
      ...
   }
   ```
2. フィルター基準に一致する変数を持つプロセス・インスタンスをリストします。

このプロセスでは、カスタマー ID は照会可能な変数 customerClaim の一部としてモデル化されます。そのため、カスタマー ID を使用すれば問題の請求を見つけることができます。

```
QueryResultSet result = process.query
    ("PROCESS_INSTANCE.NAME, QUERY_PROPERTY.STRING_VALUE",
     "QUERY_PROPERTY.VARIABLE_NAME = 'customerClaim' AND " +
     "QUERY_PROPERTY.NAME = 'customerID' AND " +
     "QUERY_PROPERTY.STRING_VALUE like 'Smith%'",
      (String)null, (Integer)null,
      (Integer)null, (TimeZone)null );
```

このアクションによって戻される照会結果セットには、プロセス・インスタンス名と、ID が Smith で始まる顧客のカスタマー ID の値が含まれています。

## 保管照会文の管理

保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

### このタスクについて

保管照会文は、データベースに保管され、名前で識別される照会のことです。専用の保管照会文と共通の保管照会文の名前を同じにすることができます。異なる複数の所有者の専用保管照会文を同じ名前にすることもできます。

保管照会文は、ビジネス・プロセス・オブジェクト、タスク・オブジェクト、またはこの 2 つのオブジェクト・タイプの組み合わせたものを対象とします。

**関連概念**

210 ページの『保管照会文のパラメーター』
保管照会文は、データベースに保管され、名前で識別される照会のことです。適格なタプルは、照会が実行されるときに動的にアセンブルされます。保管照会文を再使用可能にするには、実行時に解決される照会定義のパラメーターを使用できます。

**共通保管照会文の管理:**

共通保管照会文がシステム管理者によって作成されます。この照会は、全ユーザーが使用できます。

**このタスクについて**

システム管理者は、共通保管照会文を作成、表示、および削除できます。API 呼び出しでユーザー ID を指定しないと、その保管照会文は共通保管照会文と見なされます。

**手順**

1. 共通の保管照会文を作成します。

   例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、CustomerOrdersStartingWithA という名前を付けて保管します。

```
process.createStoredQuery("CustomerOrdersStartingWithA",
             "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
             "PROCESS_INSTANCE.NAME LIKE 'A%'",
             "PROCESS_INSTANCE.NAME",
              (Integer)null, (TimeZone)null);
```

この保管照会文の結果として、A で始まるプロセス・インスタンス名すべてのソート済みリストが、関連付けられたプロセス・インスタンス ID (PIID) とともに戻されます。

2. 保管照会文で定義された照会を実行します。

```
QueryResultSet result = process.query("CustomerOrdersStartingWithA",
               new Integer(0), null);
```

このアクションにより、基準を満たすオブジェクトが戻されます。この場合は、A で始まる顧客オーダー。

3. 使用可能な共通保管照会文の名前をリストします。

以下のコードの断片では、戻される照会のリストを共通照会のみに限定する方法を示しています。

```
String[] storedQuery = process.getStoredQueryNames(StoredQueryData.KIND_PUBLIC);
```

4. オプション: 特定の保管照会文で定義された照会を検査します。

専用の保管照会文には、共通の保管照会文と同じ名前を付けることができます。名前が同じである場合は、専用の保管照会文が戻されます。以下のコードの断片では、指定した名前の共通照会のみを戻す方法を示しています。Human Task Manager API を使用して、保管照会文に関する情報を取得する場合は、StoredQueryData ではなく、戻されるオブジェクトの StoredQuery を使用します。

```
StoredQueryData storedQuery = process.getStoredQuery
   (StoredQueryData.KIND_PUBLIC, "CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

5. 共通の保管照会文を削除します。

以下のコードの断片では、ステップ 1 で作成した保管照会文の削除方法を示しています。

```
process.deleteStoredQuery("CustomerOrdersStartingWithA");
```

**他のユーザーの専用保管照会文の管理:**

専用照会はどのユーザーでも作成できます。この照会は、照会の所有者とシステム管理者しか使用できません。

**このタスクについて**

システム管理者は、特定ユーザーに属する専用の保管照会文を管理できます。

**手順**

1. ユーザー ID Smith の専用保管照会文を作成します。

例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、Smith というユーザー ID で CustomerOrdersStartingWithA という名前を付けて保管します。

```
process.createStoredQuery("Smith", "CustomerOrdersStartingWithA",
             "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
             "PROCESS_INSTANCE.NAME LIKE 'A%'",
             "PROCESS_INSTANCE.NAME",
              (Integer)null, (TimeZone)null,
              (List)null, (String)null);
```

この保管照会文の結果として、A で始まるプロセス・インスタンス名すべてのソート済みリストが、関連付けられたプロセス・インスタンス ID (PIID) とともに戻されます。

2. 保管照会文で定義された照会を実行します。

```
QueryResultSet result = process.query
                       ("Smith", "CustomerOrdersStartingWithA",
                         (Integer)null, (Integer)null, (List)null);
new Integer(0));
```

このアクションにより、基準を満たすオブジェクトが戻されます。この場合は、A で始まる顧客オーダー。

3. 特定のユーザーに属する専用照会の名前のリストを取得します。

例えば、以下のコードの断片では、ユーザー Smith に属する専用照会のリストを取得する方法を示しています。

```
String[] storedQuery = process.getStoredQueryNames("Smith");
```

4. 特定の照会の詳細を表示します。

以下のコードの断片では、ユーザー Smith が所有する照会 CustomerOrdersStartingWithA の詳細を表示する方法を示しています。

```
StoredQueryData storedQuery = process.getStoredQuery
   ("Smith", "CustomerOrdersStartingWithA");
String selectClause = storedQuery.getSelectClause();
String whereClause = storedQuery.getWhereClause();
String orderByClause = storedQuery.getOrderByClause();
Integer threshold = storedQuery.getThreshold();
String owner = storedQuery.getOwner();
```

Human Task Manager API を使用して、保管照会文に関する情報を取得する場合は、StoredQueryData ではなく、戻されるオブジェクトの StoredQuery を使用します。

5. 専用の保管照会文を削除します。

以下のコードの断片では、ユーザー Smith が所有する専用照会を削除する方法を示しています。

```
process.deleteStoredQuery("Smith", "CustomerOrdersStartingWithA");
```

**専用保管照会文の操作:**

システム管理者でなくても、自分専用の保管照会文は作成、実行、および削除できます。また、システム管理者が作成した共通の保管照会文を使用することもできます。

**手順**

1. 専用の保管照会文を作成します。

   例えば、以下のコード断片では、プロセス・インスタンスの保管照会文を作成し、固有の名前を付けて保管します。ユーザー ID が指定されない場合、その保管照会文はログオン・ユーザーの専用保管照会文と見なされます。

   ```
   process.createStoredQuery("CustomerOrdersStartingWithA",
                "DISTINCT PROCESS_INSTANCE.PIID, PROCESS_INSTANCE.NAME",
                "PROCESS_INSTANCE.NAME LIKE 'A%'",
                "PROCESS_INSTANCE.NAME",
                 (Integer)null, (TimeZone)null);
   ```

   この照会は、文字 A で始まるプロセス・インスタンス名、および関連したプロセス・インスタンス ID (PIID) をすべてソートしたリストにして戻します。

2. 保管照会文で定義された照会を実行します。

   ```
   QueryResultSet result = process.query("CustomerOrdersStartingWithA",
                  new Integer(0));
   ```

   このアクションにより、基準を満たすオブジェクトが戻されます。この場合は、A で始まる顧客オーダー。

3. ログオン・ユーザーがアクセスできる保管照会文の名前のリストを取得します。

   以下のコードの断片では、ユーザーがアクセスできる共通の保管照会文と専用の保管照会文の両方を取得する方法を示しています。

   ```
   String[] storedQuery = process.getStoredQueryNames();
   ```

4. 特定の照会の詳細を表示します。

   以下のコードの断片では、ユーザー Smith が所有する照会 CustomerOrdersStartingWithA の詳細を表示する方法を示しています。

   ```
   StoredQueryData storedQuery = process.getStoredQuery
      ("CustomerOrdersStartingWithA");
   String selectClause = storedQuery.getSelectClause();
   String whereClause = storedQuery.getWhereClause();
   String orderByClause = storedQuery.getOrderByClause();
   Integer threshold = storedQuery.getThreshold();
   String owner = storedQuery.getOwner();
   ```

   Human Task Manager API を使用して、保管照会文に関する情報を取得する場合は、StoredQueryData ではなく、戻されるオブジェクトの StoredQuery を使用します。

5. 専用の保管照会文を削除します。

   以下のコード断片は、専用保管照会文を削除する方法を示しています。

   ```
   process.deleteStoredQuery("CustomerOrdersStartingWithA");
   ```

## ビジネス・プロセス用のアプリケーションの開発

ビジネス・プロセスは、ビジネス・ゴールを達成するために特定のシーケンスで呼び出される、ビジネス関連の一連のアクティビティーです。プロセスに対する標準のアクションに対応したアプリケーションを開発する方法を示した例が提供されています。

## このタスクについて

ビジネス・プロセスは、microflow または長期にわたって実行するプロセスのいずれかです。

- microflow は、同期して実行される短期実行のビジネス・プロセスです。結果は即時に呼び出し元に戻されます。
- 長期実行の割り込み可能プロセスは、まとめてチェーニングされるアクティビティーのシーケンスとして実行されます。プロセスで特定の構成要素を使用するとプロセス・フローが中断し、例えば、ヒューマン・タスクの呼び出し、同期バインディングを使用したサービスの呼び出し、またはタイマー駆動アクティビティーの使用などが割り込みます。

  プロセスの並列分岐は通常、非同期でナビゲートされるので、並列分岐のアクティビティーは平行して実行されます。アクティビティーのタイプとトランザクションの設定に応じて、アクティビティーを独自のトランザクションで実行することができます。

## プロセス・インスタンスに対するアクションに必要なロール

BusinessFlowManager インターフェースへのアクセス権は、呼び出し元がプロセスに対するすべてのアクションを実行できることは保証しません。呼び出し元は、アクションを実行する許可が与えられているロールを使用して、クライアント・アプリケーションにログオンする必要があります。

次の表に、それぞれのロールで実行できるプロセス・インスタンス上のアクションを示します。

| アクション | 呼び出し元のプリンシパルのロール | | |
|---|---|---|---|
| | リーダー | スターター | 管理者 |
| createMessage | x | x | x |
| createWorkItem | | | x |
| delete | | | x |
| deleteWorkItem | | | x |
| forceTerminate | | | x |
| getActiveEventHandlers | x | | x |
| getActivityInstance | x | | x |
| getAllActivities | x | | x |
| getAllWorkItems | x | | x |
| getClientUISettings | x | x | x |
| getCustomProperties | x | x | x |
| getCustomProperty | x | x | x |
| getCustomPropertyNames | x | x | x |
| getFaultMessage | x | x | x |
| getInputClientUISettings | x | x | x |
| getInputMessage | x | x | x |
| getOutputClientUISettings | x | x | x |
| getOutputMessage | x | x | x |

| アクション | 呼び出し元のプリンシパルのロール | | |
|---|---|---|---|
| | リーダー | スターター | 管理者 |
| getProcessInstance | x | x | x |
| getVariable | x | x | x |
| getWaitingActivities | x | x | x |
| getWorkItems | x | | x |
| restart | | | x |
| resume | | | x |
| setCustomProperty | | x | x |
| setVariable | | | x |
| suspend | | | x |
| transferWorkItem | | | x |

## ビジネス・プロセス・アクティビティーのアクションに必要なロール

BusinessFlowManager インターフェースへのアクセス権は、呼び出し元がアクティビティーに対するすべてのアクションを実行できることを保証するものではありません。呼び出し元は、アクションを実行する許可が与えられているロールを使用して、クライアント・アプリケーションにログオンする必要があります。

次の表に、それぞれのロールで実行できるアクティビティー・インスタンス上のアクションを示します。

| アクション | 呼び出し元のプリンシパルのロール | | | | |
|---|---|---|---|---|---|
| | リーダー | 編集者 | 潜在的な所有者 | 所有者 | 管理者 |
| cancelClaim | | | | x | x |
| claim | | | x | | x |
| complete | | | | x | x |
| createMessage | x | x | x | x | x |
| createWorkItem | | | | | x |
| deleteWorkItem | | | | | x |
| forceComplete | | | | | x |
| forceRetry | | | | | x |
| getActivityInstance | x | x | x | x | x |
| getAllWorkItems | x | x | x | x | x |
| getClientUISettings | x | x | x | x | x |
| getCustomProperties | x | x | x | x | x |
| getCustomProperty | x | x | x | x | x |
| getCustomPropertyNames | x | x | x | x | x |
| getFaultMessage | x | x | x | x | x |
| getFaultNames | x | x | x | x | x |
| getInputMessage | x | x | x | x | x |
| getOutputMessage | x | x | x | x | x |
| getVariable | x | x | x | x | x |

| アクション | 呼び出し元のプリンシパルのロール | | | | |
|---|---|---|---|---|---|
| | リーダー | 編集者 | 潜在的な所有者 | 所有者 | 管理者 |
| getVariableNames | x | x | x | x | x |
| getInputVariableNames | x | x | x | x | x |
| getOutputVariableNames | x | x | x | x | x |
| getWorkItems | x | x | x | x | x |
| setCustomProperty | | x | | x | x |
| setFaultMessage | | x | | x | x |
| setOutputMessage | | x | | x | x |
| setVariable | | | | | x |
| transferWorkItem | | | | x<br><br>潜在的な所有者または管理者に対してのみ | x |

## ビジネス・プロセスのライフ・サイクルの管理

プロセスを開始できる Business Process Choreographer API メソッドが呼び出されると、プロセス・インスタンスが生成されます。プロセス・インスタンスのすべてのアクティビティーが終了状態になるまで、プロセス・インスタンスのナビゲーションは続きます。ライフ・サイクルを管理するために、プロセス・インスタンスでさまざまなアクションを実行できます。

### このタスクについて

プロセスに対する以下の標準のライフ・サイクル・アクションに対応したアプリケーションを開発する方法を示した例が提供されています。

**ビジネス・プロセスの開始:**

ビジネス・プロセスを開始する方法は、プロセスが microflow であるか長期実行プロセスであるかによって異なります。プロセスを開始するサービスも、プロセスの開始方法にとって重要です。プロセスに固有の開始サービスを 1 つ設定するか、複数の開始サービスを設定することができます。

#### このタスクについて

microflow や長期実行プロセスを開始する標準のシナリオに対応したアプリケーションを開発する方法を示した例が提供されています。

***固有の開始サービスを含む microflow の実行:***

microflow は、receive アクティビティーまたは pick アクティビティーから開始できます。開始サービスが固有であるのは、microflow が receive アクティビティーを使って開始された場合、または pick アクティビティー内に 1 つの onMessage 定義のみがある場合です。

### このタスクについて

microflow によって要求/応答操作がインプリメントされている場合、つまり、プロセスに応答が入っている場合、call メソッドを使用してそのプロセスを実行し、その呼び出しでパラメーターとしてプロセス・テンプレート名を渡すことができます。

microflow が片方向操作である場合は、sendMessage メソッドを使用してプロセスを実行します。このメソッドは、次の例には含まれていません。

### 手順

1. オプション: プロセス・テンプレートをリストして、実行するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
   ("PROCESS_TEMPLATE.EXECUTION_MODE =
          PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
    "PROCESS_TEMPLATE.NAME",
     new Integer(50),
     (TimeZone)null);
   ```

   結果は名前でソートされます。call メソッドによって開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。
2. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

   ```
   ProcessTemplateData template = processTemplates[0];
   //create a message for the single starting receive activity
   ClientObjectWrapper input = process.createMessage
                               (template.getID(),
                                template.getInputMessageTypeName());
   DataObject myMessage = null;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the strings in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }

   //run the process
   ClientObjectWrapper output = process.call(template.getName(), input);
   DataObject myOutput = null;
   if ( output.getObject() != null && output.getObject() instanceof DataObject )
   {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
   }
   ```

   このアクションによって、プロセス・テンプレート CustomerTemplate のインスタンスが作成され、一部の顧客データが受け渡されます。この操作は、プロセスが完了してからでないと戻りません。プロセスの結果 OrderNo が、呼び出し元に戻されます。

#### *非固有の開始サービスを含む microflow の実行:*

microflow は、receive アクティビティーまたは pick アクティビティーから開始できます。 microflow が複数の onMessage 定義を含む pick アクティビティーを使用して開始された場合、開始サービスは固有ではありません。

### このタスクについて

microflow によって要求/応答操作がインプリメントされている場合、つまり、プロセスに応答が入っている場合、call メソッドを使用してそのプロセスを実行し、その呼び出しで開始サービスの ID を渡すことができます。

microflow が片方向操作である場合は、sendMessage メソッドを使用してプロセスを実行します。このメソッドは、次の例には含まれていません。

### 手順

1. オプション: プロセス・テンプレートをリストして、実行するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
   ("PROCESS_TEMPLATE.EXECUTION_MODE =
          PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_MICROFLOW",
    "PROCESS_TEMPLATE.NAME",
     new Integer(50),
     (TimeZone)null);
   ```

   結果は名前でソートされます。microflow として開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が、照会から戻されます。

2. 呼び出すべき開始サービスを判別します。

   この例では、最初に検出されたテンプレートを使用します。

   ```
   ProcessTemplateData template = processTemplates[0];
   ActivityServiceTemplateData[] startActivities =
           process.getStartActivities(template.getID());
   ```

3. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

   ```
   ActivityServiceTemplateData activity = startActivities[0];
   //create a message for the service to be called
   ClientObjectWrapper input =
         process.createMessage(activity.getServiceTemplateID(),
                               activity.getActivityTemplateID(),
                               activity.getInputMessageTypeName());
   DataObject myMessage = null;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the strings in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   //run the process
   ClientObjectWrapper output = process.call(activity.getServiceTemplateID(),
                                             activity.getActivityTemplateID(),
                                             input);
   //check the output of the process, for example, an order number
   DataObject myOutput = null;
   if ( output.getObject() != null && output.getObject() instanceof DataObject )
   ```

```
{
  myOutput  = (DataObject)output.getObject();
  int order = myOutput.getInt("OrderNo");
}
```

このアクションによって、プロセス・テンプレート CustomerTemplate のインスタンスが作成され、一部の顧客データが受け渡されます。この操作は、プロセスが完了してからでないと戻りません。プロセスの結果 OrderNo が、呼び出し元に戻されます。

***固有の開始サービスを含む長期実行プロセスの開始:***

開始サービスが固有の場合、initiate メソッドを使用して、プロセス・テンプレート名をパラメーターとして渡すことができます。これは、長期実行プロセスが、単一の receive アクティビティーまたは pick アクティビティーのいずれかを使用して開始する、および単一の pick アクティビティーが 1 つのみの onMessage 定義を持つ場合にあてはまります。

**手順**

1. オプション: プロセス・テンプレートをリストして、開始するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
     ("PROCESS_TEMPLATE.EXECUTION_MODE =
          PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
      "PROCESS_TEMPLATE.NAME",
       new Integer(50),
       (TimeZone)null);
   ```

   結果は名前でソートされます。initiate メソッドによって開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。プロセス・インスタンス名を指定する場合、アンダースコアーで開始しないようにする必要があります。プロセス・インスタンス名が指定されていない場合、ストリング・フォーマットのプロセス・インスタンス ID (PIID) が名前として使用されます。

   ```
   ProcessTemplateData template = processTemplates[0];
   //create a message for the single starting receive activity
   ClientObjectWrapper input = process.createMessage
                               (template.getID(),
                               template.getInputMessageTypeName());
   DataObject myMessage = null;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the strings in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   //start the process
   PIID piid = process.initiate(template.getName(), "CustomerOrder", input);
   ```

このアクションによって、インスタンス CustomerOrder が作成され、一部の顧客データが受け渡されます。プロセスが開始されると、新規プロセス・インスタンスのオブジェクト ID を呼び出し元に戻します。

プロセス・インスタンスのスターターは、要求の呼び出し元に設定されます。このユーザーは、このプロセス・インスタンスの作業項目を受信します。プロセス・インスタンスのプロセス管理者、リーダー、および編集者が決定され、プロセス・インスタンスの作業項目を受信します。追加のアクティビティー・インスタンスが決定されます。これらは自動的に開始されるか、または human task、receive、pick アクティビティーの場合、作業項目が潜在的な所有者に対して作成されます。

***非固有の開始サービスを含む長期実行プロセスの開始:***

長期実行プロセスは、複数の開始 receive アクティビティーまたは pick アクティビティーを介して開始することができます。initiate メソッドを使用して、プロセスを開始することができます。例えば、プロセスが複数の receive または pick アクティビティー、または複数の onMessage 定義を持つ pick アクティビティーから開始される場合など、開始サービスが固有のものではない場合、呼び出されるサービスを識別する必要があります。

**手順**

1. オプション: プロセス・テンプレートをリストして、開始するプロセスの名前を探します。

   プロセスの名前がすでに分かっている場合、このステップはオプションです。

   ```
   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
     ("PROCESS_TEMPLATE.EXECUTION_MODE =
          PROCESS_TEMPLATE.EXECUTION_MODE.EXCECUTION_MODE_LONG_RUNNING",
      "PROCESS_TEMPLATE.NAME",
       new Integer(50),
       (TimeZone)null);
   ```

   結果は名前でソートされます。長期実行プロセスとして開始できるソート済みテンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 呼び出すべき開始サービスを判別します。

   ```
   ProcessTemplateData template = processTemplates[0];
   ActivityServiceTemplateData[] startActivities =
           process.getStartActivities(template.getID());
   ```

3. 該当するタイプの入力メッセージを使ってプロセスを開始します。

   メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。プロセス・インスタンス名を指定する場合、アンダースコアーで開始しないようにする必要があります。プロセス・インスタンス名が指定されていない場合、ストリング・フォーマットのプロセス・インスタンス ID (PIID) が名前として使用されます。

   ```
   ActivityServiceTemplateData activity = startActivities[0];
   //create a message for the service to be called
   ClientObjectWrapper input = process.createMessage
                               (activity.getServiceTemplateID(),
                                activity.getActivityTemplateID(),
                                activity.getInputMessageTypeName());
   DataObject myMessage = null;
   ```

```
if ( input.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)input.getObject();
  //set the strings in the message, for example, a customer name
  myMessage.setString("CustomerName", "Smith");
}
//start the process
PIID piid = process.sendMessage(activity.getServiceTemplateID(),
                                activity.getActivityTemplateID(),
                                input);
```

このアクションによって、インスタンスが作成され、一部の顧客データが受け渡されます。プロセスが開始されると、新規プロセス・インスタンスのオブジェクト ID を呼び出し元に戻します。

プロセス・インスタンスの開始は、要求の呼び出し元に設定され、プロセス・インスタンスの作業項目を受信します。プロセス・インスタンスのプロセス管理者、リーダー、および編集者が決定され、プロセス・インスタンスの作業項目を受信します。追加のアクティビティー・インスタンスが決定されます。これらは自動的に開始されるか、または human task、 receive、pick アクティビティーの場合、作業項目が潜在的な所有者に対して作成されます。

**ビジネス・プロセスの中断と再開:**

長期にわたって実行するトップレベルのプロセス・インスタンスを実行中に中断し、再開して完了することができます。

**始める前に**

呼び出し元は、プロセス・インスタンスの管理者、またはビジネス・プロセス管理者でなければなりません。プロセス・インスタンスを中断するには、プロセス・インスタンスが実行状態または失敗状態でなければなりません。

**このタスクについて**

例えば、プロセスで後で使用されるバックエンド・システムへのアクセスを構成するために、プロセス・インスタンスを中断することがあります。プロセスの前提条件を満たしていれば、そのプロセス・インスタンスを再開することができます。また、プロセスを中断し、プロセス・インスタンスの失敗の原因となっている問題を修正して、問題が修正されたら再開することもできます。

**手順**

1. 中断する実行中のプロセス CustomerOrder を取得します。

   ```
   ProcessInstanceData processInstance =
                       process.getProcessInstance("CustomerOrder");
   ```

2. プロセス・インスタンスを中断します。

   ```
   PIID piid = processInstance.getID();
   process.suspend( piid );
   ```

   このアクションにより、指定したトップレベルのプロセス・インスタンスが中断します。プロセス・インスタンスは、中断状態になります。 autonomy 属性が child に設定されたサブプロセスは、実行中、失敗、終了中、または補正中の状

態であれば、中断されます。このプロセス・インスタンスに関連するインライン・タスクも中断されますが、このプロセス・インスタンスに関連するスタンドアロン・タスクは中断されません。

この状態では、開始されたアクティビティーはまだ完了することはできますが、新規のアクティビティーはアクティブ化されません。例えば、要求済み状態のヒューマン・タスク・アクティビティーは完了することができます。

3. プロセス・インスタンスを再開します。

```
process.resume( piid );
```

このアクションにより、プロセス・インスタンスとそのサブプロセスが中断前の状態に戻ります。

**ビジネス・プロセスの再開:**

完了、終了、失敗、補正のいずれかの状態にあるプロセス・インスタンスを再開させることができます。

**始める前に**

呼び出し元は、プロセス・インスタンスの管理者、またはビジネス・プロセス管理者でなければなりません。

**このタスクについて**

プロセス・インスタンスの再開は、プロセス・インスタンスを初めて開始する手順と同様です。ただし、プロセス・インスタンスの再開時には、プロセス・インスタンス ID が認識されているため、インスタンスの入力メッセージが使用可能です。

プロセスに、プロセス・インスタンスを作成可能な複数の receive アクティビティーまたは pick アクティビティー (receive choice アクティビティーとも呼ばれる) が含まれる場合、これらのアクティビティーに属するすべてのメッセージを使用して、プロセス・インスタンスを再始動します。これらのアクティビティーのいずれかが、要求/応答操作をインプリメントする場合、関連する reply アクティビティーがナビゲートされると、応答が再度送信されます。

**手順**

1. 再開させるプロセスを取得します。

```
ProcessInstanceData processInstance =
                    process.getProcessInstance("CustomerOrder");
```

2. プロセス・インスタンスを再開します。

```
PIID piid = processInstance.getID();
process.restart( piid );
```

このアクションにより、指定されたプロセス・インスタンスが再開されます。

**プロセス・インスタンスの終了:**

プロセス管理者権限を持つユーザーが、リカバリー不能状態として認識されているトップレベルのプロセス・インスタンスを終了する必要がある場合があります。プ

ロセス・インスタンスは、未解決のサブプロセスやアクティビティーがあってもこれらを待たずに即時に終了するため、プロセス・インスタンスの終了は例外的な場合にのみ行ってください。

**手順**

1. 終了するプロセス・インスタンスを検索します。

   ```
   ProcessInstanceData processInstance =
          process.getProcessInstance("CustomerOrder");
   ```

2. プロセス・インスタンスを終了します。

   プロセス・インスタンスを終了する場合、補正を使用してプロセス・インスタンスを終了することも、補正を使用せずに終了することもできます。

   補正を使用してプロセス・インスタンスを終了するには、以下のようにします。

   ```
   PIID piid = processInstance.getID();
   process.forceTerminate(piid, CompensationBehaviour.INVOKE_COMPENSATION);
   ```

   補正を使用しないでプロセス・インスタンスを終了するには、以下のようにします。

   ```
   PIID piid = processInstance.getID();
   process.forceTerminate(piid);
   ```

   補正を使用してプロセス・インスタンスを終了する場合、プロセスの補正は、障害が最上位スコープで発生したかのように実行されます。補正を使用せずにプロセス・インスタンスを終了する場合、プロセス・インスタンスはアクティビティー、予定タスク、またはインライン呼び出しタスクが正常に終了するのを待たずに、即時に終了されます。

   プロセスおよびプロセスに関連するスタンドアロン・タスクによって開始されるアプリケーションは、強制終了要求によって終了されません。そのようなアプリケーションを終了させる場合は、プロセスによって開始されるアプリケーションを明示的に終了するステートメントをプロセス・アプリケーションに追加する必要があります。

**プロセス・インスタンスの削除:**

完了済みのプロセス・インスタンスは、プロセス・モデル内のプロセス・テンプレートに対応するプロパティーが設定されていれば、Business Process Choreographer データベースから自動的に削除されます。例えば、監査ログに書き込まれていないプロセス・インスタンスのデータを照会する場合などは、プロセス・インスタンスをデータベースに保存しておくことができます。ただし、格納されたプロセス・インスタンスのデータは、ディスク・スペースとパフォーマンスに影響を与えるだけでなく、同じ相関セット値を使用するプロセス・インスタンスも作成されなくなります。したがって、プロセス・インスタンス・データは、データベースから定期的に削除してください。

**このタスクについて**

プロセス・インスタンスを削除するには、プロセス管理者権限が必要であり、そのプロセス・インスタンスは、トップレベルのプロセス・インスタンスでなければなりません。

以下の例では、完了したプロセス・インスタンスをすべて削除する方法が示されています。

**手順**

1. 完了したプロセス・インスタンスをリストします。

```
QueryResultSet result =
     process.query("DISTINCT PROCESS_INSTANCE.PIID",
                   "PROCESS_INSTANCE.STATE =
                           PROCESS_INSTANCE.STATE.STATE_FINISHED",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、完了したプロセス・インスタンスをリストした照会結果セットを戻します。

2. 完了したプロセス・インスタンスを削除します。

```
while (result.next() )
{
   PIID piid = (PIID) result.getOID(1);
   process.delete(piid);
}
```

   このアクションにより、選択したプロセス・インスタンスとそのインライン・タスクがデータベースから削除されます。

## ヒューマン・タスク・アクティビティーの処理

ビジネス・プロセス内のヒューマン・タスク・アクティビティーは、作業項目を通じて、組織内のさまざまな人に割り当てられます。プロセスが開始されると、潜在的な所有者に対して作業項目が作成されます。

### このタスクについて

ヒューマン・タスク・アクティビティーが活動状態にされると、アクティビティー・インスタンスと、関連した予定タスクの両方が作成されます。ヒューマン・タスク・アクティビティーおよび作業項目管理の処理は、Human Task Manager に委任されます。アクティビティー・インスタンスの状態変更はすべてタスク・インスタンスに反映され、その反対にタスク・インスタンスの状態変更はアクティビティー・インスタンスに反映されます。

潜在的な所有者がアクティビティーを要求します。このユーザーは、関係のある情報の提供とアクティビティーの完了に対して責任があります。

**手順**

1. 作業の準備ができている、ログオン・ユーザーに属するアクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
                    ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
                    WORK_ITEM.REASON =
                         WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、ログオン・ユーザーが作業することができるアクティビティーが含まれる照会結果セットを戻します。

2. 作業対象のアクティビティーを要求します。

```
if (result.size() > 0)
{
 result.first();
 AIID aiid = (AIID) result.getOID(1);
 ClientObjectWrapper input = process.claim(aiid);
 DataObject activityInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    activityInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

アクティビティーが要求されると、アクティビティーの入力メッセージが戻されます。

3. アクティビティーの作業が終了したら、アクティビティーを完了します。 アクティビティーは、正常に完了することも、障害メッセージが表示されて完了することもあります。アクティビティーが正常に完了した場合、出力メッセージが渡されます。アクティビティーが失敗した場合、アクティビティーは失敗状態または停止状態に置かれ、障害メッセージが渡されます。これらのアクションに対して、適切なメッセージを作成する必要があります。メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

   a. アクティビティーを正常に完了するには、出力メッセージを作成します。

```
ActivityInstanceData activity = process.getActivityInstance(aiid);
ClientObjectWrapper output =
      process.createMessage(aiid, activity.getOutputMessageTypeName());
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the activity
process.complete(aiid, output);
```

   このアクションは、オーダー番号が含まれる出力メッセージを設定します。

   b. 障害が発生した場合にアクティビティーを完了するには、障害メッセージを作成します。

```
//retrieve the faults modeled for the human task activity
List faultNames = process.getFaultNames(aiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
      process.createMessage(aiid, faultNames.get(0) );

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if ( myFault.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)myFault.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setInt("error",1304);
```

```
}

process.complete(aiid, myFault,(String)faultNames.get(0) );
```

このアクションは、アクティビティーを失敗状態または停止状態のいずれかに設定します。プロセス・モデル内のアクティビティーの **continueOnError** パラメーターが真に設定されている場合、アクティビティーは失敗状態に置かれ、ナビゲーションが続行されます。 **continueOnError** パラメーターが FALSE に設定されているときに、周囲の有効範囲で障害がキャッチされない場合、そのアクティビティーは停止状態になります。この状態では、強制完了または強制再試行を使用してアクティビティーを修復できます。

## 単独ユーザー・ワークフローの処理

ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。このタイプのワークフローには、並列パスは存在しません。 completeAndClaimSuccessor API は、このタイプのワークフローの処理をサポートします。

### このタスクについて

オンライン・ブックストアでは、購入者は一連の操作を完了することで本を注文します。この一連の操作は、ヒューマン・タスク・アクティビティー (予定タスク) として実装できます。購入者が複数の書籍を注文する場合は、これが次のヒューマン・タスク・アクティビティーの要求に相当します。このタイプのワークフローは、ページ・フローとも呼ばれます。ユーザー・インターフェース定義が、ユーザー・インターフェースのダイアログのフローを制御するアクティビティーと関連付けられているためです。

completeAndClaimSuccessor API はヒューマン・タスク・アクティビティーを完了し、ログオン・ユーザーの同じプロセス・インスタンスで次のアクティビティーを要求します。そして、次に要求したアクティビティーの情報 (処理される入力メッセージなど) を戻します。次のアクティビティーは、完了したアクティビティーと同じトランザクション内で使用可能になるため、プロセス・モデル内のすべてのヒューマン・タスク・アクティビティーのトランザクション動作が participates に設定される必要があります。

この例を、Business Flow Manager API と Human Task Manager API の両方を使用する例と比較してください。

### 手順

1. アクティビティー・シーケンスで最初のアクティビティーを要求します。

```
//
//Query the list of activities that can be claimed by the logged-on user
//
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
                    ACTIVITY.STATE = ACTIVITY.STATE.STATE_READY AND
                    ACTIVITY.KIND = ACTIVITY.KIND.KIND_STAFF AND
                    WORK_ITEM.REASON =
                         WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                   (String)null, (Integer)null, (TimeZone)null);
...
```

```
//
//Claim the first activity
//
if (result.size() > 0)
{
 result.first();
 AIID aiid = (AIID) result.getOID(1);
 ClientObjectWrapper input = process.claim(aiid);
 DataObject activityInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    activityInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

アクティビティーが要求されると、アクティビティーの入力メッセージが戻されます。

2. アクティビティーの作業が終了したら、そのアクティビティーを完了して次のアクティビティーを要求します。

   アクティビティーを完了するには、出力メッセージを渡します。出力メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

```
ActivityInstanceData activity = process.getActivityInstance(aiid);
ClientObjectWrapper output =
      process.createMessage(aiid, activity.getOutputMessageTypeName());
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the activity and claim the next one
CompleteAndClaimSuccessorResult successor =
   process.completeAndClaimSuccessor(aiid, output);
```

   このアクションは、オーダー番号が含まれる出力メッセージを設定し、シーケンス内の次のアクティビティーを要求します。後続アクティビティーに AutoClaim が設定されており、有効なパスが複数存在する場合は、後続アクティビティーのすべてが要求され、ランダムなアクティビティーが次のアクティビティーとして戻されます。このユーザーに割り当て可能な後続アクティビティーが他にない場合は、NULL が戻されます。

   後に続くことができる並列パスがプロセスに含まれ、これらのパスに、ログオン・ユーザーが潜在的な所有者であるヒューマン・タスク・アクティビティーが複数含まれる場合、ランダムなアクティビティーが自動的に要求され、次のアクティビティーとして戻されます。

3. 次のアクティビティーを処理します。

```
String name = successor.getActivityName();

ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject()!=
               null && nextInput.getObject() instanceof DataObject )
{
```

```
  activityInput = (DataObject)input.getObject();
  // read the values
  ...
}

aiid = successor.getAIID();
```

4. アクティビティーを完了する場合は、ステップ 2 に進みます。

**関連タスク**

278 ページの『ヒューマン・タスクを含む単一の個人ワークフローの処理』
ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。この例では、一連のヒューマン・タスク・アクティビティー (予定タスク) として、書籍を注文するための一連のアクションを実装する方法を示しています。ワークフローの処理には、Business Flow Manager と Human Task Manager API の両方が使用されます。

## 待機中のアクティビティーへのメッセージの送信

インバウンド・メッセージ・アクティビティー (receive アクティビティー、pick アクティビティーの onMessage、イベント・ハンドラーの onEvent) を使用して、実行中のプロセスを「外の世界」からのイベントと同期することができます。例えば、情報に対する要求に応えたお客様からの E メールの受信は、このようなイベントとみなされます。

### このタスクについて

親タスクを使用して、アクティビティーにメッセージを送信できます。

**手順**

1. 特定のプロセス・インスタンス ID を持つプロセス・インスタンスのログオンしたユーザーからのメッセージを待っているアクティビティー・サービス・テンプレートをリストします。

```
ActivityServiceTemplateData[] services = process.getWaitingActivities(piid);
```

2. 最初の待機サービスにメッセージを送信します。

   最初のサービスを、ユーザーがサービスを提供しようとするサービスと想定します。呼び出し元は、メッセージを受信するアクティビティーの潜在的なスターター、またはプロセス・インスタンスの管理者である必要があります。

```
VTID vtid = services[0].getServiceTemplateID();
ATID atid = services[0].getActivityTemplateID();
String inputType = services[0].getInputMessageTypeName();

// create a message for the service to be called
  ClientObjectWrapper message =
        process.createMessage(vtid,atid,inputMessageTypeName);
  DataObject myMessage = null;
  if ( message.getObject()!= null && message.getObject() instanceof DataObject )
  {
    myMessage = (DataObject)message.getObject();
    //set the strings in the message, for example, chocolate is to be ordered
    myMessage.setString("Order", "chocolate");
  }

  // send the message to the waiting activity
  process.sendMessage(vtid, atid, message);
}
```

このアクションによって、指定されたメッセージを待機アクティビティー・サービスに送信し、一部のオーダー・データを渡します。

また、プロセス・インスタンス ID を指定して、メッセージが指定されたプロセス・インスタンスに送信されたことを確認することもできます。プロセス・インスタンス ID が指定されていない場合、メッセージは、アクティビティー・サービス、およびメッセージの相関値によって識別されたプロセス・インスタンスに送信されます。プロセス・インスタンス ID が指定された場合、相関値を使用して検出されたプロセス・インスタンスがチェックされ、指定されたプロセス・インスタンス ID であることが確認されます。

## イベントの処理

ビジネス・プロセス全体とビジネス・プロセスの各スコープを、関連するイベントの発生時に呼び出されるイベント・ハンドラーと関連付けることができます。 プロセスによりイベント・ハンドラーを使用して、Web サービス操作を提供できるという点で、イベント・ハンドラーは、receive アクティビティーや pick アクティビティーと似ています。

## このタスクについて

イベント・ハンドラーは、対応するスコープが実行中である限り、何度でも呼び出すことができます。また、イベント・ハンドラーの複数インスタンスを並行して活動化することができます。

以下のコードの断片では、あるプロセス・インスタンス用のアクティブなイベント・ハンドラーを取得する方法、および入力メッセージを送信する方法を示しています。

### 手順

1. プロセス・インスタンス ID のデータを判別し、そのプロセスのアクティブなイベント・ハンドラーをリストします。

```
ProcessInstanceData processInstance =
       process.getProcessInstance( "CustomerOrder2711");
EventHandlerTemplateData[] events = process.getActiveEventHandlers(
                                     processInstance.getID() );
```

2. 入力メッセージを送信します。

   この例では、最初に検出されたイベント・ハンドラーを使用します。

```
EventHandlerTemplateData event = null;
if ( events.length > 0 )
{
    event = events[0];

    // create a message for the service to be called
    ClientObjectWrapper input = process.createMessage(
    event.getID(), event.getInputMessageTypeName());

    if (input.getObject() != null && input.getObject() instanceof DataObject )
    {
       DataObject inputMessage = (DataObject)input.getObject();
       // set content of the message, for example, a customer name, order number
       inputMessage.setString("CustomerName", "Smith");
       inputMessage.setString("OrderNo", "2711");
```

```
          // send the message
         process.sendMessage( event.getProcessTemplateName(),
                              event.getPortTypeNamespace(),
                              event.getPortTypeName(),
                              event.getOperationName(),
       input );
       }
   }
```

このアクションにより、指定されたメッセージがプロセスのアクティブなイベント・ハンドラーに送信されます。

## プロセスの結果の分析

プロセスは、Web Services Description Language (WSDL) の片方向操作または要求/応答操作としてモデル化される Web サービス操作を公開できます。片方向インターフェースを使用する長期実行プロセスの結果は、そのプロセスに出力がないため、getOutputMessage メソッドを使用して取り出すことはできません。ただし、代わりに変数の内容を照会できます。

### このタスクについて

プロセスの結果は、プロセス・インスタンスが派生したプロセス・テンプレートが、派生したプロセス・インスタンスの自動削除を指定しない場合にのみ、データベースに保管されます。

#### 手順

プロセスの結果を分析し、例えば、オーダー番号などを確認します。

```
QueryResultSet result = process.query
                       ("PROCESS_INSTANCE.PIID",
                        "PROCESS_INSTANCE.NAME = 'CustomerOrder' AND
                         PROCESS_INSTANCE.STATE =
                                PROCESS_INSTANCE.STATE.STATE_FINISHED",
                        (String)null, (Integer)null, (TimeZone)null);
if (result.size() > 0)
{
  result.first();
  PIID piid = (PIID) result.getOID(1);
  ClientObjectWrapper output = process.getOutputMessage(piid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject )
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

## アクティビティーの修復

長期実行プロセスには、やはり長期間実行されるアクティビティーが含まれる場合があります。これらのアクティビティーでは、catch されていないエラーが発生して、停止状態になる可能性があります。実行状態のアクティビティーが、反応していないように見える可能性もあります。どちらの場合でも、プロセス管理者は、プロセスのナビゲーションを継続できるように、いくつかの方法でアクティビティーを処理することができます。

## このタスクについて

Business Process Choreographer API は、アクティビティーの修復のために、forceRetry メソッドおよび forceComplete メソッドを提供しています。アクティビティーの修復アクションをアプリケーションに追加する方法を示した例が提供されています。

### アクティビティーの強制完了:

長期実行プロセスのアクティビティーで、障害が発生することがあります。これらの障害が、囲んでいるスコープ内で障害ハンドラーによって catch されておらず、関連したアクティビティー・テンプレートが、エラー発生時にアクティビティーが停止するように指定している場合、アクティビティーは修復することができるように停止状態になります。この状態で、アクティビティーの完了を強制することができます。

### このタスクについて

例えば、アクティビティーが応答しない場合、実行状態のアクティビティーを強制的に完了することもできます。

特定のタイプのアクティビティーでは、追加要件が存在します。

**ヒューマン・タスク・アクティビティー**
送信されるはずだったメッセージ、または引き起こされるはすだった障害など、強制完了呼び出しでパラメーターを渡すことができます。

**script アクティビティー**
強制完了呼び出しで、パラメーターを渡すことはできません。ただし、修復する必要がある変数を設定する必要があります。

**invoke アクティビティー**
invoke アクティビティーが実行状態の場合、サブプロセスでない非同期サービスを呼び出す invoke アクティビティーを強制的に完了することもできます。例えば、非同期サービスが呼び出されて応答がない場合、こうすることがあります。

### 手順

1. 停止状態の停止アクティビティーをリストします。

```
QueryResultSet result =
     process.query("DISTINCT ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                    PROCESS_INSTANCE.NAME='CustomerOrder'",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、 CustomerOrder プロセス・インスタンスに対して停止アクティビティーを戻します。

2. 例えば、停止したヒューマン・タスク・アクティビティーなどのアクティビティーを完了します。

   この例では、出力メッセージが渡されます。

```
if (result.size() > 0)
{
   result.first();
```

```
   AIID aiid = (AIID) result.getOID(1);
   ActivityInstanceData activity = process.getActivityInstance(aiid);
  ClientObjectWrapper output =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( output.getObject()!= null && output.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)output.getObject();
     //set the parts in your message, for example, an order number
     myMessage.setInt("OrderNo", 4711);
   }

   boolean continueOnError = true;
   process.forceComplete(aiid, output, continueOnError);
}
```

このアクションによって、アクティビティーが完了します。エラーが発生すると、**continueOnError** パラメーターに基づいて、障害が forceComplete 要求によって発生する場合に実行されるアクションが決まります。

例では、**continueOnError** が true です。この値は、障害が発生した場合にアクティビティーが失敗状態になることを意味します。障害は、処理されるかプロセス・スコープに到達するまで、アクティビティーの囲んでいるスコープに伝搬されます。次にプロセスは障害状態になり、最終的に失敗状態になります。

**停止されたアクティビティーの再試行:**

長期実行プロセスのアクティビティーに、囲んでいるスコープ内で catch されていない障害が発生した場合、関連したアクティビティー・テンプレートが、エラー発生時にアクティビティーの停止を指定しているときは、アクティビティーは停止状態になり、修復することができます。アクティビティーの実行を再試行することができます。

**このタスクについて**

アクティビティーが使用する変数を設定することができます。また、script アクティビティーの例外とともに、アクティビティーが予想したメッセージなど、強制再試行呼び出しのパラメーターを渡すこともできます。

**手順**

1. 停止アクティビティーをリストします。

```
QueryResultSet result =
      process.query("DISTINCT ACTIVITY.AIID",
                    "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                     PROCESS_INSTANCE.NAME='CustomerOrder'",
                     (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、 CustomerOrder プロセス・インスタンスに対して停止アクティビティーを戻します。

2. 例えば、停止したヒューマン・タスク・アクティビティーなどのアクティビティーの実行を再試行します。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ActivityInstanceData activity = process.getActivityInstance(aiid);
```

```
  ClientObjectWrapper input =
        process.createMessage(aiid, activity.getOutputMessageTypeName());
  DataObject myMessage = null;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
    {
      myMessage = (DataObject)input.getObject();
      //set the strings in your message, for example, chocolate is to be ordered
      myMessage.setString("OrderNo", "chocolate");
    }

   boolean continueOnError = true;
   process.forceRetry(aiid, input, continueOnError);
}
```

このアクションによって、アクティビティーが再試行されます。エラーが発生した場合、**continueOnError** パラメーターによって、forceRetry 要求の処理中にエラーが発生した場合に実行するアクションが決まります。

例では、**continueOnError** が true です。この場合、forceRetry 要求の処理中にエラーが発生すると、アクティビティーは失敗状態になります。障害は、処理されるかプロセス・スコープに到達するまで、アクティビティーの囲んでいるスコープに伝搬されます。次にプロセスは障害状態になり、プロセス状態が失敗状態で終了する前にプロセス・レベルの障害ハンドラーが実行されます。

## BusinessFlowManagerService インターフェース

BusinessFlowManagerService インターフェースは、クライアント・アプリケーションから呼び出すことができるビジネス・プロセス機能を公開します。

BusinessFlowManagerService インターフェースから呼び出すことができるメソッドは、プロセスまたはアクティビティーの状態、およびそのメソッドが含まれているアプリケーションを使用するユーザーの権限によって異なります。ビジネス・プロセス・オブジェクトを操作するための main メソッドを、以下にリストします。これらのメソッドおよび BusinessFlowManagerService インターフェースで使用可能なその他のメソッドについての詳細は、com.ibm.bpe.api パッケージ内の Javadoc を参照してください。

## プロセス・テンプレート

プロセス・テンプレートは、バージョン付けされ、デプロイされ、インストールされるプロセス・モデルで、ビジネス・プロセスの仕様を含んでいます。これは、例えば sendMessage() などの適切な要求を発行することによって、インスタンス化および開始することができます。プロセス・インスタンスの実行は、サーバーによって自動的に駆動されます。

*表 7. プロセス・テンプレート用の API メソッド*

| メソッド | 説明 |
|---|---|
| getProcessTemplate | 指定されたプロセス・テンプレートを取得します。 |
| queryProcessTemplates | データベースに保管されているプロセス・テンプレートを取得します。 |

## プロセス・インスタンス

以下の API メソッドは、プロセス・インスタンスの開始に関連しています。

*表 8. プロセス・インスタンスの開始に関連する API メソッド*

| メソッド | 説明 |
|---|---|
| call | マイクロフローを作成および実行します。 |
| callWithReplyContext | 指定されたプロセス・テンプレートから、固有の開始サービスでのマイクロフローまたは固有の開始サービスでの長期実行プロセスを作成および実行します。呼び出しは、結果を非同期で待ちます。 |
| callWithUISettings | マイクロフローを作成および実行し、出力メッセージとクライアント・ユーザー・インターフェース (UI) の設定を戻します。 |
| initiate | プロセス・インスタンスを作成し、そのプロセス・インスタンスの処理を開始します。このメソッドは、長時間実行プロセスに使用します。このメソッドは、応答不要送信を適用するマイクロフローに対しても使用できます。 |
| sendMessage | 指定されたメッセージを、指定されたアクティビティー・サービスおよびプロセス・インスタンスに送信します。同じ相関セット値を持つプロセス・インスタンスが存在しない場合には作成されます。このプロセスは、固有または非固有のどちらかの開始サービスを持つことができます。 |
| getStartActivities | 指定されたプロセス・テンプレートからプロセス・インスタンスを開始できるアクティビティーに関する情報を戻します。 |
| getActivityServiceTemplate | 指定されたアクティビティー・サービス・テンプレートを取得します。 |

*表 9. プロセス・インスタンスのライフ・サイクルを制御するための API メソッド*

| メソッド | 説明 |
|---|---|
| suspend | 実行状態または失敗状態にある、長期実行中のトップレベルのプロセス・インスタンスの実行を中断します。 |
| resume | 中断状態にある、長期実行中のトップレベルのプロセス・インスタンスの実行を再開します。 |
| restart | 完了、失敗、または終了状態にある、長期実行中のトップレベルのプロセス・インスタンスを再始動します。 |

*表 9. プロセス・インスタンスのライフ・サイクルを制御するための API メソッド (続き)*

| メソッド | 説明 |
| --- | --- |
| forceTerminate | 指定されたトップレベルのプロセス・インスタンスと、子 autonomy を含むそのサブプロセス、およびその実行中のアクティビティー、要求済みのアクティビティー、または待機中のアクティビティーを終了します。 |
| delete | 指定されたトップレベルのプロセス・インスタンスと、子 autonomy を含むそのサブプロセスを削除します。 |
| query | データベースから、検索基準に一致するプロパティーを取得します。 |

## アクティビティー

invoke アクティビティーの場合、プロセス・モデルで、それらのアクティビティーがエラー状態でも続行されるように指定できます。continueOnError フラグが FALSE に設定されているときに未処理エラーが発生すると、そのアクティビティーは停止状態になります。その場合は、プロセス管理者が、そのアクティビティーを修復することができます。continueOnError フラグおよびそれに関連する修復機能は、例えば、invoke アクティビティーが失敗することがある長期実行プロセスなどで使用することができますが、補正および障害処理のモデル化には、かなりの労力が必要です。

アクティビティーの操作および修復には、以下のメソッドが使用可能です。

*表 10. アクティビティー・インスタンスのライフ・サイクルを制御するための API メソッド*

| メソッド | 説明 |
| --- | --- |
| claim | 準備ができたアクティビティー・インスタンスを要求し、ユーザーがそのアクティビティーを使用できるようにします。 |
| cancelClaim | アクティビティー・インスタンスの要求を取り消します。 |
| complete | アクティビティー・インスタンスを完了します。 |
| completeAndClaimSuccessor | アクティビティー・インスタンスを完了し、ログオン担当者の同じプロセス・インスタンス内の次のアクティビティーを要求します。 |
| forceComplete | 以下を強制的に完了します。<br>• 実行中状態または停止状態にあるアクティビティー・インスタンス。<br>• 作動可能状態または要求済み状態にあるヒューマン・タスク・アクティビティー。<br>• 待機状態にある wait アクティビティー。 |

*表 10. アクティビティー・インスタンスのライフ・サイクルを制御するための API メソッド (続き)*

| メソッド | 説明 |
| --- | --- |
| forceRetry | 以下を強制的に反復します。<br>• 実行中状態または停止状態にあるアクティビティー・インスタンス。<br>• 作動可能状態または要求済み状態にあるヒューマン・タスク・アクティビティー。 |
| query | データベースから、検索基準に一致するプロパティーを取得します。 |

### 変数およびカスタム・プロパティー

このインターフェースは、変数の値を取得および設定するための get および set メソッドを提供します。指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスに関連付けたり、指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンスから取得することもできます。カスタム・プロパティーの名前および値は、java.lang.String 型である必要があります。

*表 11. 変数およびカスタム・プロパティーの API メソッド*

| メソッド | 説明 |
| --- | --- |
| getVariable | 指定された変数を取得します。 |
| setVariable | 指定された変数を設定します。 |
| getCustomProperty | 指定されたアクティビティーまたはプロセス・インスタンスの指定されたカスタム・プロパティーを取得します。 |
| getCustomProperties | 指定されたアクティビティーまたはプロセス・インスタンスのカスタム・プロパティーを取得します。 |
| getCustomPropertyNames | 指定されたアクティビティーまたはプロセス・インスタンスのカスタム・プロパティーの名前を取得します。 |
| setCustomProperty | 指定されたアクティビティーまたはプロセス・インスタンスのカスタム固有値を保管します。 |

## ヒューマン・タスク用のアプリケーションの開発

タスクは、コンポーネントが人をサービスとして呼び出したり、人がサービスを呼び出すための手段となります。ヒューマン・タスクに関する標準的なアプリケーションの例が提供されています。

### このタスクについて

Human Task Manager API について詳しくは、com.ibm.task.api パッケージにある Javadoc を参照してください。

## 同期インターフェースを起動する呼び出しタスクの開始

呼び出しタスクは、Service Component Architecture (SCA) コンポーネントに関連付けられます。タスクは、開始されると SCA コンポーネントを起動します。呼び出しタスクを同期的に開始するのは、関連した SCA コンポーネントを同期的に呼び出せる場合に限ってください。

### このタスクについて

このような SCA コンポーネントは、Microflow や単純な Java クラスなどとして実装できます。

このシナリオでは、タスク・テンプレートのインスタンスが作成され、一部の顧客データが渡されます。両方向操作が戻るまで、タスクは実行状態のままです。タスクの結果である OrderNo が呼び出し元に戻されます。

**手順**

1. オプション: タスク・テンプレートをリストして、実行する呼び出しタスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
   ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
    "TASK_TEMPL.NAME",
     new Integer(50),
     (TimeZone)null);
   ```

   結果は名前でソートされます。ソート済みの派生元テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。
2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```
3. タスクを作成して、タスクを同期実行します。

   タスクを同期実行するには、両方向の操作であることが必要です。例では、createAndCallTask メソッドを使用してタスクを作成および実行します。

   ```
   ClientObjectWrapper output = task.createAndCallTask( template.getName(),
                                                        template.getNamespace(),
                                                        input);
   ```
4. タスクの結果を分析します。

   ```
   DataObject myOutput = null;
   if ( output.getObject() != null && output.getObject() instanceof DataObject )
   {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
   }
   ```

## 非同期インターフェースを起動する呼び出しタスクの開始

呼び出しタスクは、Service Component Architecture (SCA) コンポーネントに関連付けられます。タスクは、開始されると SCA コンポーネントを起動します。呼び出しタスクを非同期的に開始するのは、関連した SCA コンポーネントを非同期的に呼び出せる場合に限ってください。

### このタスクについて

このような SCA コンポーネントは、長時間実行プロセスや片方向操作などとして実装できます。

このシナリオでは、タスク・テンプレートのインスタンスが作成され、一部の顧客データが渡されます。

**手順**

1. オプション: タスク・テンプレートをリストして、実行する呼び出しタスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
     ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_ORIGINATING",
      "TASK_TEMPL.NAME",
       new Integer(50),
       (TimeZone)null);
   ```

   結果は名前でソートされます。ソート済みの派生元テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。
2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```
3. タスクを作成して、非同期に実行します。

   例では、createAndStartTask メソッドを使用してタスクを作成および実行します。

   ```
   task.createAndStartTask( template.getName(),
                            template.getNamespace(),
                            input,
                            (ReplyHandlerWrapper)null);
   ```

## タスク・インスタンスの作成と開始

このシナリオでは、コラボレーション・タスク (API ではヒューマン・タスク ともいう) を定義するタスク・テンプレートのインスタンスを作成し、タスク・インスタンスを開始する方法を示します。

**手順**

1. オプション: タスク・テンプレートをリストして、実行するコラボレーション・タスクの名前を探します。

   タスクの名前が既に分かっている場合は、このステップはオプションです。

   ```
   TaskTemplate[] taskTemplates = task.queryTaskTemplates
   ("TASK_TEMPL.KIND=TASK_TEMPL.KIND.KIND_HUMAN",
    "TASK_TEMPL.NAME",
     new Integer(50),
     (TimeZone)null);
   ```

   結果は名前でソートされます。ソート済みのタスク・テンプレートのうちの最初の 50 個を収容した配列が照会から戻されます。

2. 該当する型の入力メッセージを作成します。

   ```
   TaskTemplate template = taskTemplates[0];

   // create a message for the selected task
   ClientObjectWrapper input = task.createInputMessage( template.getID());
   DataObject myMessage = null ;
   if ( input.getObject()!= null && input.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)input.getObject();
     //set the parts in the message, for example, a customer name
     myMessage.setString("CustomerName", "Smith");
   }
   ```

3. コラボレーション・タスクを作成し、開始します。この例では、応答ハンドラーは指定されません。

   この例では、createAndStartTask メソッドを使用してタスクを作成し、開始します。

   ```
   TKIID tkiid = task.createAndStartTask( template.getName(),
                                          template.getNamespace(),
                                          input,
                                          (ReplyHandlerWrapper)null);
   ```

   タスク・インスタンスに関連する人に対して作業項目が作成されます。例えば、潜在的な所有者は、新規タスク・インスタンスを要求できます。

4. タスク・インスタンスを要求します。

   ```
   ClientObjectWrapper input2 = task.claim(tkiid);
   DataObject taskInput = null ;
   if ( input2.getObject()!= null && input2.getObject() instanceof DataObject )
   {
     taskInput = (DataObject)input2.getObject();
     // read the values
     ...
   }
   ```

   タスク・インスタンスが要求されると、タスクの入力メッセージが戻されます。

## 予定タスクまたはコラボレーション・タスクの処理

予定タスク (API では*参加タスク* ともいう) またはコラボレーション・タスク (API では*ヒューマン・タスク* ともいう) は、作業項目を通じて組織内のさまざまな人に割り当てられます。プロセスがヒューマン・タスク・アクティビティーにナビゲートしたときなどに、予定タスクとそれに関連した作業項目が作成されます。

## このタスクについて

潜在的な所有者の中の 1 人が、作業項目に関連したタスクを要求します。このユーザーは、関係のある情報の提供とタスクの完了に対して責任があります。

### 手順

1. 作業の準備ができている、ログオン・ユーザーに属するタスクをリストします。

```
QueryResultSet result =
     task.query("TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_READY AND
                (TASK.KIND = TASK.KIND.KIND_PARTICIPATING OR
                 TASK.KIND = TASK.KIND.KIND_HUMAN)AND
                 WORK_ITEM.REASON =
                   WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                 (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、ログオン・ユーザーが作業することができるタスクが含まれる照会結果セットを戻します。

2. 作業対象のタスクを要求します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper input = task.claim(tkiid);
  DataObject taskInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    taskInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

   タスクが要求されると、タスクの入力メッセージが戻されます。

3. タスクの作業が完了した場合、タスクを完了します。

   タスクは、正常に完了することも、障害メッセージが表示されて完了することもあります。タスクが正常に完了した場合、出力メッセージが渡されます。タスクが正常に完了しなかった場合、障害メッセージが渡されます。これらのアクションに対して、適切なメッセージを作成する必要があります。

   a. タスクを正常に完了するには、出力メッセージを作成します。

```
ClientObjectWrapper output =
      task.createOutputMessage(tkiid);
DataObject myMessage = null ;
if ( output.getObject()!= null && output.getObject() instanceof DataObject )
{
  myMessage = (DataObject)output.getObject();
  //set the parts in your message, for example, an order number
  myMessage.setInt("OrderNo", 4711);
}

//complete the task
task.complete(tkiid, output);
```

      このアクションは、オーダー番号が含まれる出力メッセージを設定します。タスクは、完了状態になります。

b. 障害が発生した場合にタスクを完了するには、障害メッセージを作成します。

```
//retrieve the faults modeled for the task
List faultNames = task.getFaultNames(tkiid);

//create a message of the appropriate type
ClientObjectWrapper myFault =
      task.createFaultMessage(tkiid, (String)faultNames.get(0));

// set the parts in your fault message, for example, an error number
DataObject myMessage = null ;
if ( myFault.getObject()!= null && input.getObject() instanceof DataObject )
{
  myMessage = (DataObject)myFault.getObject();
  //set the parts in the message, for example, a customer name
  myMessage.setInt("error",1304);
}

task.complete(tkiid, (String)faultNames.get(0), myFault);
```

このアクションにより、エラー・コードを含む障害メッセージが設定されます。タスクは、失敗状態になります。

## タスク・インスタンスの中断と再開

コラボレーション・タスク・インスタンス (API ではヒューマン・タスク ともいう) または予定タスク・インスタンス (API では*参加タスク* ともいう) を中断できます。

### 始める前に

タスク・インスタンスは、作動可能状態または要求済み状態にすることができます。これをエスカレートすることが可能です。呼び出し元は、タスク・インスタンスの所有者、オリジネーター、または管理者である必要があります。

### このタスクについて

タスク・インスタンスは、実行中に中断することができます。そうすることによって、例えば、タスクの完了に必要な情報を収集することができます。情報が使用可能になったら、タスク・インスタンスを再開できます。

#### 手順

1. ログオン・ユーザーによって要求されたタスクのリストを取得します。

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.STATE = TASK.STATE.STATE_CLAIMED",
                                   (String)null,
                                   (Integer)null,
                                   (TimeZone)null);
```

   このアクションにより、ログオン・ユーザーによって要求されたタスクのリストを含む照会結果セットが戻されます。
2. タスク・インスタンスを中断します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  task.suspend(tkiid);
}
```

このアクションにより、指定されたタスク・インスタンスが中断されます。タスク・インスタンスは中断状態になります。

3. プロセス・インスタンスを再開します。

```
task.resume( tkiid );
```

このアクションにより、タスク・インスタンスが中断前の状態になります。

## タスクの結果の分析

予定タスク (API では参加 タスクともいう) またはコラボレーション・タスク (API ではヒューマン・タスク ともいう) は非同期に実行します。タスク開始時に応答ハンドラーが指定された場合、タスク完了時に自動的に出力メッセージが戻されます。応答ハンドラーが指定されていない場合、メッセージを明示的に検索する必要があります。

### このタスクについて

タスクの結果は、そのタスク・インスタンスの派生元となったタスク・テンプレートに、派生したタスク・インスタンスの自動削除が指定されていない場合にのみ、データベースに保管されます。

### 手順

タスクの結果を分析します。
例では、正常に完了したタスクのオーダー番号を確認する方法を示します。

```
QueryResultSet result = task.query("DISTINCT TASK.TKIID",
                                   "TASK.NAME = 'CustomerOrder' AND
                                    TASK.STATE = TASK.STATE.STATE_FINISHED",
                                    (String)null, (Integer)null, (TimeZone)null);
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  ClientObjectWrapper output = task.getOutputMessage(tkiid);
  DataObject myOutput = null;
  if ( output.getObject() != null && output.getObject() instanceof DataObject)
  {
     myOutput  = (DataObject)output.getObject();
     int order = myOutput.getInt("OrderNo");
  }
}
```

## タスク・インスタンスの終了

管理者権限を有する担当者が、リカバリー不能状態になったと分かったタスク・インスタンスを終了する必要が生じる場合があります。タスク・インスタンスは即時に終了されるので、例外的な状況の場合にのみタスク・インスタンスを終了してください。

### 手順

1. 終了するタスク・インスタンスを検索します。

```
Task taskInstance = task.getTask(tkiid);
```

2. タスク・インスタンスを終了します。

```
TKIID tkiid = taskInstance.getID();
task.terminate(tkiid);
```

タスク・インスタンスは、未解決のタスクを待機しないで即時に終了します。

## タスク・インスタンスの削除

タスク・インスタンスは、インスタンスの派生元となった関連タスク・テンプレートに自動削除が指定されている場合にのみ、完了時に自動的に削除されます。この例では、完了して自動的に削除されなかったタスク・インスタンスすべてを削除する方法を示します。

**手順**

1. 完了したタスク・インスタンスをリストします。

```
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_FINISHED",
                 (String)null, (Integer)null, (TimeZone)null);
```

このアクションにより、完了したタスク・インスタンスをリストした照会の結果セットが戻されます。

2. 完了したタスク・インスタンスを削除します。

```
while (result.next() )
{
 TKIID tkiid = (TKIID) result.getOID(1);
 task.delete(tkiid);
}
```

## 要求されたタスクの解放

潜在的な所有者がタスクを要求した場合、この所有者にはタスクの完了に対する責任があります。ただし、要求されたタスクを、別の潜在的所有者が要求できるように解放しなければならない場合があります。

### このタスクについて

管理者権限を持つユーザーが、要求されたタスクを解放する必要がある場合があります。この状態は、例えば、タスクを完了する必要があるが、タスクの所有者が不在の場合に発生する可能性があります。タスクの所有者も、要求されたタスクを解放できます。

**手順**

1. 特定のユーザー (例では、Smith) 所有の、要求されたタスクをリストします。

```
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.STATE = TASK.STATE.STATE_CLAIMED AND
                 TASK.OWNER = 'Smith'",
                 (String)null, (Integer)null, (TimeZone)null);
```

このアクションは、指定されたユーザーの Smith が要求したタスクをリストする照会結果セットを戻します。

2. 要求されたタスクを解放します。

```
if (result.size() > 0)
{
  result.first();
  TKIID tkiid = (TKIID) result.getOID(1);
  task.cancelClaim(tkiid, true);
}
```

このアクションは、タスクを作動可能状態に戻すので、他の潜在的な所有者の 1 人が要求することができます。元の所有者が設定した出力データまたは障害データはすべて保持されます。

## 作業項目の管理

アクティビティー・インスタンスまたはタスク・インスタンスの存続時間中に、オブジェクトに関連したユーザーのセットが変わる場合があります。例えば、社員が休暇に入ったり、新しい社員を雇用したり、またはワークロードを異なった方法で分散させる必要がある場合です。このような変更に対応するために、作業項目を作成、削除、または転送するようにアプリケーションを開発することができます。

### このタスクについて

作業項目は、特定の理由でのユーザーまたはユーザーのグループへのオブジェクトの割り当てを表します。通常、このオブジェクトはヒューマン・タスク・アクティビティー・インスタンス、プロセス・インスタンス、またはタスク・インスタンスのいずれかです。理由は、ユーザーがオブジェクトに対して持っているロールから派生します。ユーザーは、オブジェクトとの関連において異なるロールを持つことができ、作業項目はこのようなロールそれぞれに対して作成されるため、1 つのオブジェクトが複数の作業項目を持つことが可能です。例えば、予定タスク・インスタンスでは、管理者、読者、編集者、所有者の作業項目を同時に持つことができます。

作業項目を管理するために実行可能なアクションは、ユーザーのロールによって異なります。例えば、管理者は作業項目を作成、削除、および転送できますが、タスク所有者は作業項目の転送のみが可能です。

- 作業項目を作成します。

```
// query the task instance for which an additional
// administrator is to be specified
QueryResultSet result = task.query("TASK.TKIID",
                                   "TASK.NAME='CustomerOrder'",
                                    (String)null, (Integer)null,
                                    (TimeZone)null);
if ( result.size() > 0 )
{
  result.first();
  // create the work item
  task.createWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_ADMINISTRATOR,"Smith");
}
```

このアクションによって、管理者のロールがあるユーザー Smith の作業項目が作成されます。

- 作業項目を削除します。

```
// query the task instance for which a work item is to be deleted
QueryResultSet result = task.query("TASK.TKIID",
                                   "TASK.NAME='CustomerOrder'",
                                    (String)null, (Integer)null,
                                    (TimeZone)null);
if ( result.size() > 0 )
{
  result.first();
  // delete the work item
  task.deleteWorkItem((TKIID)(result.getOID(1)),
                          WorkItem.REASON_READER,"Smith");
}
```

このアクションによって、リーダーのロールがあるユーザー Smith の作業項目が削除されます。

- 作業項目を転送します。

```
// query the task that is to be rescheduled
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.NAME='CustomerOrder' AND
                 TASK.STATE=TASK.STATE.STATE_READY AND
                 WORK_ITEM.REASON=WORK_ITEM.REASON.REASON_POTENTIAL_OWNER AND
                 WORK_ITEM.OWNER_ID='Miller'",
                 (String)null, (Integer)null, (TimeZone)null);
if ( result.size() > 0 )
{
  result.first();
  // transfer the work item from user Miller to user Smith
  // so that Smith can work on the task
  task.transferWorkItem((TKIID)(result.getOID(1)),
                           WorkItem.REASON_POTENTIAL_OWNER,"Miller","Smith");
}
```

このアクションによって、作業項目をユーザー Smith に転送するので、Smith は作業項目で作業することができます。

## 実行時のタスク・テンプレートおよびタスク・インスタンスの作成

通常、WebSphere Integration Developer などのモデル化ツールを使用して、タスク・テンプレートを作成します。次に、タスク・テンプレートを WebSphere Process Server にインストールし、例えば Business Process Choreographer Explorer を使用して、これらのテンプレートからインスタンスを作成します。ただし、実行時にヒューマン・タスクまたは参加タスクのインスタンスまたはテンプレートを作成することもできます。

### このタスクについて

例えば、アプリケーションのデプロイ時にタスク定義が使用不可である場合、ワークフローに含まれるタスクがまだ認識されていない場合、またはタスクがユーザーのグループ間の随時のコラボレーションを対象とする必要がある場合などに、このようにすることがあります。

臨時の予定タスクまたはコラボレーション・タスクをモデル化できます。それには、com.ibm.task.api.TaskModel クラスのインスタンスを作成し、それを使用して再使用可能なタスク・テンプレートを作成するか、または 1 回実行のタスク・インスタンスを直接作成します。 TaskModel クラスのインスタンスを作成するために、一連のファクトリー・メソッドが com.ibm.task.api.ClientTaskFactory ファクトリー・ク

ラスで使用可能です。実行時のヒューマン・タスクのモデル化は、Eclipse Modeling Framework (EMF) に基づいています。

**手順**

1. createResourceSet ファクトリー・メソッドを使用して org.eclipse.emf.ecore.resource.ResourceSet を作成します。
2. オプション: 複雑なメッセージ・タイプを使用する予定の場合、org.eclipse.xsd.XSDFactory (ファクトリー・メソッド getXSDFactory() を使用して取得できる) を使ってそのメッセージ・タイプを定義するか、または loadXSDSchema ファクトリー・メソッドを使用して既存の XML スキーマを直接インポートできます。

   複雑なタイプを WebSphere Process Server が使用できるようにするには、そのタイプをエンタープライズ・アプリケーションの一部としてデプロイします。
3. タイプ javax.wsdl.Definition の Web サービス記述言語 (WSDL) 定義を作成またはインポートします。

   createWSDLDefinition メソッドを使用して新規の WSDL 定義を作成できます。次に、それをポート・タイプおよび操作に追加できます。また、loadWSDLDefinition ファクトリー・メソッドを使用して、既存の WSDL 定義を直接インポートできます。
4. createTTask ファクトリー・メソッドを使用してタスク定義を作成します。

   より複雑なタスク・エレメントを追加または操作する場合は、getTaskFactory ファクトリー・メソッドを使って取得できる com.ibm.wbit.tel.TaskFactory クラスを使用できます。
5. createTaskModel ファクトリー・メソッドを使用してタスク・モデルを作成し、それをステップ 1 で作成されたリソース・バンドルに渡します。リソース・バンドルはその間に作成されたその他のすべての成果物を集約します。
6. オプション: TaskModel validate メソッドを使用してモデルを検証します。

## タスクの結果

**TaskModel** パラメーターを持つ Human Task Manager EJB API create メソッドの 1 つを使用して、再使用可能なタスク・テンプレートを作成するか、または 1 回実行のタスク・インスタンスを作成します。

### 単純 Java 型を使用するランタイム・タスクの作成:

この例では、インターフェースで単純 Java 型 (例えば String オブジェクト) のみを使用するランタイム・タスクを作成します。

### このタスクについて

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

### 手順

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. 操作の WSDL 定義を作成し、記述を追加します。

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
         ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );

// create a port type
PortType portType = factory.createPortType( definition, "doItPT" );

// create an operation; the input and output messages are of type String:
// a fault message is not specified
Operation operation = factory.createOperation
        ( definition, portType, "doIt",
          new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
          new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
          (Map)null );
```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

```
TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );
```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。アプリケーションが単純 Java 型のみを使用するため、アプリケーション名を指定する必要はありません。

- 以下の断片はタスク・インスタンスを作成します。

```
atask.createTask( taskModel, (String)null, "HTM" );
```

- 以下の断片はタスク・テンプレートを作成します。

```
task.createTaskTemplate( taskModel, (String)null );
```

**タスクの結果**

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

**複合タイプを使用するランタイム・タスクの作成:**

この例では、インターフェースで複合タイプを使用するランタイム・タスクを作成します。複合タイプは既に定義されています。すなわち、クライアントのローカル・ファイル・システムには、複合タイプの記述を含む XSD ファイルがあります。

**このタスクについて**

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

**手順**

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();
ResourceSet resourceSet = factory.createResourceSet();
```

2. 複合タイプの XSD 定義をリソース・セットに追加して、操作の定義時に使用できるようにします。

   ファイルは、コードが実行されたロケーションの相対位置に配置されます。

```
factory.loadXSDSchema( resourceSet, "InputBO.xsd" );
factory.loadXSDSchema( resourceSet, "OutputBO.xsd" );
```

3. 操作の WSDL 定義を作成し、記述を追加します。

```
// create the WSDL interface
Definition definition = factory.createWSDLDefinition
         ( resourceSet, new QName( "http://www.ibm.com/task/test/", "test" ) );

// create a port type
PortType portType = factory.createPortType( definition, "doItPT" );

// create an operation; the input message is an InputBO and
// the output message an OutputBO;
// a fault message is not specified
Operation operation = factory.createOperation
        ( definition, portType, "doIt",
          new QName( "http://Input", "InputBO" ),
          new QName( "http://Output", "OutputBO" ),
          (Map)null );
```

4. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

   ```
   TTask humanTask = factory.createTTask( resourceSet,
                                          TTaskKinds.HTASK_LITERAL,
                                          "TestTask",
                                          new UTCDate( "2005-01-01T00:00:00" ),
                                          "http://www.ibm.com/task/test/",
                                          portType,
                                          operation );
   ```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

5. ヒューマン・タスク・モデルのプロパティーを変更します。

   ```
   // use the methods from the com.ibm.wbit.tel package, for example,
   humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

   // retrieve the task factory to create or modify composite task elements
   TaskFactory taskFactory = factory.getTaskFactory();

   // specify escalation settings
   TVerb verb = taskFactory.createTVerb();
   verb.setName("John");

   // create escalationReceiver and add verb
   TEscalationReceiver escalationReceiver =
                       taskFactory.createTEscalationReceiver();
   escalationReceiver.setVerb(verb);

   // create escalation and add escalation receiver
   TEscalation escalation = taskFactory.createTEscalation();
   escalation.setEscalationReceiver(escalationReceiver);
   ```

6. すべてのリソース定義を含むタスク・モデルを作成します。

   ```
   TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
   ```

7. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

   ```
   ValidationProblem[] validationProblems = taskModel.validate();
   ```

8. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。アプリケーションが Business Process Choreographer によってロードされるように、アプリケーションにダミー・タスクまたはダミー・プロセスも含まれるようにしてください。

   - 以下の断片はタスク・インスタンスを作成します。

     ```
     task.createTask( taskModel, "BOapplication", "HTM" );
     ```

   - 以下の断片はタスク・テンプレートを作成します。

     ```
     task.createTaskTemplate( taskModel, "BOapplication" );
     ```

### タスクの結果

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

### 既存のインターフェースを使用するランタイム・タスクの作成:

この例は、既に定義されているインターフェースを使用するランタイム・タスクを作成します。すなわち、クライアントのローカル・ファイル・システムには、インターフェースの記述を含むファイルがあります。

### このタスクについて

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

### 手順

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

   ```
   ClientTaskFactory factory = ClientTaskFactory.newInstance();
   ResourceSet resourceSet = factory.createResourceSet();
   ```

2. 操作の WSDL 定義および記述にアクセスします。

   インターフェース記述は、コードが実行されたロケーションの相対位置に配置されます。

   ```
   Definition definition = factory.loadWSDLDefinition(
                           resourceSet, "interface.wsdl"  );
   PortType portType = definition.getPortType(
                       new QName( definition.getTargetNamespace(), "doItPT" ) );
   Operation operation = portType.getOperation
                               ("doIt", (String)null, (String)null);
   ```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

   ```
   TTask humanTask = factory.createTTask( resourceSet,
                                          TTaskKinds.HTASK_LITERAL,
                                          "TestTask",
                                          new UTCDate( "2005-01-01T00:00:00" ),
                                          "http://www.ibm.com/task/test/",
                                          portType,
                                          operation );
   ```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

   ```
   // use the methods from the com.ibm.wbit.tel package, for example,
   humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

   // retrieve the task factory to create or modify composite task elements
   TaskFactory taskFactory = factory.getTaskFactory();

   // specify escalation settings
   TVerb verb = taskFactory.createTVerb();
   ```

```
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

   ```
   TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
   ```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

   ```
   ValidationProblem[] validationProblems = taskModel.validate();
   ```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

   HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。アプリケーションが Business Process Choreographer によってロードされるように、アプリケーションにダミー・タスクまたはダミー・プロセスも含まれるようにしてください。

   - 以下の断片はタスク・インスタンスを作成します。

     ```
     task.createTask( taskModel, "BOapplication", "HTM" );
     ```

   - 以下の断片はタスク・テンプレートを作成します。

     ```
     task.createTaskTemplate( taskModel, "BOapplication" );
     ```

**タスクの結果**

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

**呼び出し側アプリケーションのインターフェースを使用するランタイム・タスクの作成:**

この例では、呼び出し側アプリケーションに属するインターフェースを使用するランタイム・タスクを作成します。例えば、ランタイム・タスクがビジネス・プロセスの Java 断片で作成され、プロセス・アプリケーションからのインターフェースを使用します。

**このタスクについて**

この例は、リソースがロードされる呼び出し側エンタープライズ・アプリケーションのコンテキスト内のみで実行されます。

**手順**

1. ClientTaskFactory にアクセスし、新規タスク・モデルの定義を格納するリソース・セットを作成します。

```
ClientTaskFactory factory = ClientTaskFactory.newInstance();

// specify the context class loader so that following resources are found
ResourceSet resourceSet = factory.createResourceSet
              ( Thread.currentThread().getContextClassLoader() );
```

2. 操作の WSDL 定義および記述にアクセスします。

   収容パッケージ JAR ファイル内でパスを指定します。

```
Definition definition = factory.loadWSDLDefinition( resourceSet,
                          "com/ibm/workflow/metaflow/interface.wsdl" );
                      PortType portType = definition.getPortType(
                      new QName( definition.getTargetNamespace(), "doItPT" ) );
Operation operation = portType.getOperation
                                  ("doIt", (String)null, (String)null);
```

3. 新規ヒューマン・タスクの EMF モデルを作成します。

   タスク・インスタンスを作成する場合は、有効開始日 (UTCDate) は必要ありません。

```
TTask humanTask = factory.createTTask( resourceSet,
                                       TTaskKinds.HTASK_LITERAL,
                                       "TestTask",
                                       new UTCDate( "2005-01-01T00:00:00" ),
                                       "http://www.ibm.com/task/test/",
                                       portType,
                                       operation );
```

   このステップは、タスク・モデルのプロパティーをデフォルト値で初期化します。

4. ヒューマン・タスク・モデルのプロパティーを変更します。

```
// use the methods from the com.ibm.wbit.tel package, for example,
humanTask.setBusinessRelevance( TBoolean, YES_LITERAL );

// retrieve the task factory to create or modify composite task elements
TaskFactory taskFactory = factory.getTaskFactory();

// specify escalation settings
TVerb verb = taskFactory.createTVerb();
verb.setName("John");

// create escalationReceiver and add verb
TEscalationReceiver escalationReceiver =
                    taskFactory.createTEscalationReceiver();
escalationReceiver.setVerb(verb);

// create escalation and add escalation receiver
TEscalation escalation = taskFactory.createTEscalation();
escalation.setEscalationReceiver(escalationReceiver);
```

5. すべてのリソース定義を含むタスク・モデルを作成します。

```
TaskModel taskModel = ClientTaskFactory.createTaskModel( resourceSet );
```

6. タスク・モデルを検証し、検証で問題が検出された場合それを訂正します。

```
ValidationProblem[] validationProblems = taskModel.validate();
```

7. ランタイム・タスク・インスタンスまたはランタイム・タスク・テンプレートを作成します。

HumanTaskManagerService インターフェースを使用して、タスク・インスタンスまたはタスク・テンプレートを作成します。データ型定義を利用できるように、データ型定義を含むアプリケーション名を指定する必要があります。

- 以下の断片はタスク・インスタンスを作成します。

```
task.createTask( taskModel, "WorkflowApplication", "HTM" );
```

- 以下の断片はタスク・テンプレートを作成します。

```
task.createTaskTemplate( taskModel, "WorkflowApplication" );
```

**タスクの結果**

ランタイム・タスク・インスタンスが作成された場合、それを始動することができます。ランタイム・タスク・テンプレートが作成された場合、そのテンプレートからタスク・インスタンスを作成できます。

## HumanTaskManagerService インターフェース

HumanTaskManagerService インターフェースは、ローカルまたはリモート・クライアントから呼び出すことができるタスク関連機能を公開します。

呼び出すことができるメソッドは、タスクの状態、およびそのメソッドが含まれているアプリケーションを使用する人物の権限によって異なります。タスク・オブジェクトを操作するための main メソッドを、以下にリストします。これらのメソッドおよび HumanTaskManagerService インターフェースで使用可能なその他のメソッドについての詳細は、com.ibm.task.api パッケージ内の Javadoc を参照してください。

## タスク・テンプレート

タスク・テンプレートを扱う作業には、以下のメソッドが使用可能です。

*表 12. タスク・テンプレート用の API メソッド*

| メソッド | 説明 |
|---|---|
| getTaskTemplate | 指定されたタスク・テンプレートを取得します。 |
| createAndCallTask | 指定されたタスク・テンプレートからタスク・インスタンスを作成および実行し、同期式に結果を待機します。 |
| createAndStartTask | 指定されたタスク・テンプレートからタスク・インスタンスを作成および開始します。 |
| createTask | 指定されたタスク・テンプレートからタスク・インスタンスを作成します。 |
| createInputMessage | 指定されたタスク・テンプレート用の入力メッセージを作成します。例えば、タスクの開始に使用できるメッセージを作成します。 |
| queryTaskTemplates | データベースに保管されているタスク・テンプレートを取得します。 |

## タスク・インスタンス

タスク・インスタンスを扱う作業には、以下のメソッドが使用可能です。

*表 13. タスク・インスタンス用の API メソッド*

| メソッド | 説明 |
|---|---|
| getTask | タスク・インスタンスを取得します。任意の状態のタスク・インスタンスを取得できます。 |
| callTask | 呼び出しタスクを同期で開始します。 |
| startTask | 既に作成済みのタスクを開始します。 |
| suspend | コラボレーション・タスクまたは予定タスクを中断します。 |
| resume | コラボレーション・タスクまたは予定タスクを再開します。 |
| terminate | 指定されたタスク・インスタンスを終了します。呼び出しタスクを終了する場合、このアクションは、呼び出されたサービスには影響しません。 |
| delete | 指定されたタスク・インスタンスを削除します。 |
| claim | 処理のためタスクを要求します。 |
| update | タスク・インスタンスを更新します。 |
| complete | タスク・インスタンスを完了します。 |
| cancelClaim | 別の潜在的な所有者が処理できるように、要求されたタスク・インスタンスをリリースします。 |
| createWorkItem | タスク・インスタンスの作業項目を作成します。 |
| transferWorkItem | 指定された所有者に作業項目を転送します。 |
| deleteWorkItem | 作業項目を削除します。 |

## エスカレーション

エスカレーションを扱う作業には、以下のメソッドが使用可能です。

*表 14. エスカレーションで使用できる API メソッド*

| メソッド | 説明 |
|---|---|
| getEscalation | 指定されたエスカレーション・インスタンスを取得します。 |

## カスタム・プロパティー

タスク、タスク・テンプレート、およびエスカレーションには、すべてカスタム・プロパティーを指定できます。このインターフェースは、カスタム・プロパティーの値を取得および設定するための get および set メソッドを提供します。指定されたプロパティーをプロセス・インスタンスおよびアクティビティー・インスタンス

に関連付けたり、指定されたプロパティーをタスク・インスタンスから取得することもできます。カスタム・プロパティーの名前および値は、java.lang.String 型である必要があります。次のメソッドは、タスク、タスク・テンプレート、およびエスカレーションの場合に有効です。

*表 15. 変数およびカスタム・プロパティーの API メソッド*

| メソッド | 説明 |
|---|---|
| getCustomProperty | 指定されたタスク・インスタンスの指定された 1 つのカスタム・プロパティーを取得します。 |
| getCustomProperties | 指定されたタスク・インスタンスのカスタム・プロパティー (複数) を取得します。 |
| getCustomPropertyNames | タスク・インスタンスのすべてのカスタム・プロパティーの名前を取得します。 |
| setCustomProperty | 指定されたタスク・インスタンスのカスタム固有値を保管します。 |

**各タスクで許可されるアクション:**

タスクに対して実行可能なアクションは、予定タスク、コラボレーション・タスク、呼び出しタスク、または管理タスクのいずれであるかによって異なります。

HumanTaskManager インターフェースで提供されるすべてのアクションが、すべての種類のタスクに対して使用できるわけではありません。次の表に、タスクの種類ごとに実行可能なアクションを示します。

| | タスクの種類 | | | |
|---|---|---|---|---|
| アクション | 予定タスク | コラボレーション・タスク | 呼び出しタスク | 管理タスク |
| callTask | | | X | |
| cancelClaim | X | X[1] | | |
| claim | X | X[1] | | |
| complete | X | X[1] | | X |
| completeWithFollowOnTask[4] | X | X[1] | | |
| completeWithFollowOnTask[5] | | X[3] | X[3] | |
| createFaultMessage | X | X | X | X |
| createInputMessage | X | X | X | X |
| createOutputMessage | X | X | X | X |
| createWorkItem | X | X[1] | X | X |
| delete | X[1] | X[1] | X | X[1] |
| deleteWorkItem | X | X[1] | X | X |
| getCustomProperty | X | X[1] | X | X |
| getDocumentation | X | X[1] | X | X |
| getFaultNames | X | X[1] | | |
| getFaultMessage | X | X[1] | X | |
| getInputMessage | X | X[1] | X | |

| アクション | タスクの種類 | | | |
|---|---|---|---|---|
| | 予定タスク | コラボレーション・タスク | 呼び出しタスク | 管理タスク |
| getOutputMessage | X | X[1] | X | |
| getUsersInRole | X | X[1] | X | X |
| getTask | X | X[1] | X | X |
| getUISettings | X | X[1] | X | X |
| resume | X | X[1] | | |
| setCustomProperty | X | X[1] | X | X |
| setFaultMessage | X | X[1] | | |
| setOutputMessage | X | X[1] | | |
| startTask | X[1] | X[1] | X | X |
| startTaskAsSubtask[6] | X | X[1] | | |
| startTaskAsSubtask[7] | | X[3] | X[3] | |
| suspend | X | X[1] | | |
| suspendWithCancelClaim | X | X[1] | | |
| terminate | X[1] | X[1] | X[1] | |
| transferWorkItem | X | X[1] | X | X |
| update | X | X[1] | X | X |

**注:**

1. スタンドアロン・タスク、臨時タスク、およびタスク・テンプレートの場合のみ
2. スタンドアロン・タスク、ビジネス・プロセス内のインライン・タスク、および臨時タスクの場合のみ
3. スタンドアロン・タスクおよび臨時タスクの場合のみ
4. 後続のタスクを持つことが可能なタスクの種類
5. 後続のタスクとして使用可能なタスクの種類
6. サブタスクを持つことが可能なタスクの種類
7. サブタスクとして使用可能なタスクの種類

# ビジネス・プロセスおよびヒューマン・タスク用アプリケーションの開発

ほとんどのビジネス・プロセス・シナリオには担当者が関係します。例えば、プロセスが開始または管理されるとき、あるいはヒューマン・タスク・アクティビティーが実行されるときには、ビジネス・プロセスに担当者の対話が必要となります。これらのシナリオをサポートするために、Business Flow Manager API と Human Task Manager API の両方を使用する必要があります。

## このタスクについて

ビジネス・プロセス・シナリオに担当者を組み込むために、ビジネス・プロセスに以下の種類のタスクを含めることができます。

- インライン呼び出しタスク (API では*親タスク* ともいう)。

  すべての receive アクティビティー、pick アクティビティーの各 onMessage エレメント、およびイベント・ハンドラーの各 onEvent エレメントに対して呼び出し

タスクを提供できます。次いで、このタスクはプロセスの開始、または実行中のプロセス・インスタンスとの通信を許可されているユーザーを制御します。

- 管理タスク。

  プロセスの管理またはプロセスの失敗したアクティビティーに対する管理操作の実行を許可されたユーザーを指定するための管理タスクを提供できます。

- 予定タスク (API では*参加タスク* ともいう)。

  予定タスクはヒューマン・タスク・アクティビティーを実装します。このタイプのアクティビティーにより、プロセスに担当者を含めることができます。

ビジネス・プロセス内のヒューマン・タスク・アクティビティーは、担当者がビジネス・プロセス・シナリオで実行する予定タスクを表します。 Business Flow Manager API と Human Task Manager API の両方を使用して、以下のシナリオを実現できます。

- ビジネス・プロセスとは、プロセスに属するすべてのアクティビティー (予定タスクによって表されるヒューマン・タスク・アクティビティーを含む) のコンテナーです。プロセス・インスタンスが作成されると、固有オブジェクト ID (PIID) が割り当てられます。
- ヒューマン・タスク・アクティビティーがプロセス・インスタンスの実行中に活動状態にされると、アクティビティー・インスタンスが作成されます。これはその固有オブジェクト ID (AIID) によって識別されます。同時に、インライン予定タスク・インスタンスも作成されます。これはそのオブジェクト ID (TKIID) によって識別されます。ヒューマン・タスク・アクティビティーとタスク・インスタンスの関係は、次のようにオブジェクト ID を使用して実現されます。
  - アクティビティー・インスタンスの予定タスク ID が、関連した予定タスクの TKIID に設定されます。
  - タスク・インスタンスの包含コンテキスト ID が、関連したアクティビティー・インスタンスを含むプロセス・インスタンスの PIID に設定されます。
  - タスク・インスタンスの親コンテキスト ID が、関連したアクティビティー・インスタンスの AIID に設定されます。
- すべてのインライン予定タスク・インスタンスのライフ・サイクルは、プロセス・インスタンスによって管理されます。プロセス・インスタンスが削除されると、タスク・インスタンスも削除されます。言い換えると、包含コンテキスト ID がプロセス・インスタンスの PIID に設定されているすべてのタスクが自動的に削除されます。

## 開始できるプロセス・テンプレートまたはアクティビティーの判別

ビジネス・プロセスは、Business Flow Manager API の call、initiate、または sendMessage メソッドの呼び出しにより開始できます。プロセスに 1 つの開始アクティビティーしかない場合は、パラメーターとしてプロセス・テンプレート名を必要とするメソッド・シグニチャーを使用できます。プロセスに複数の開始アクティビティーがある場合は、開始アクティビティーを明示的に識別する必要があります。

## このタスクについて

ビジネス・プロセスをモデル化する場合、モデラーは、ユーザーのサブセットだけしかプロセス・テンプレートからプロセス・インスタンスを作成できないように決定することができます。これは、インライン呼び出しタスクをプロセスの開始アクティビティーに関連付け、許可制限をそのタスクに指定することで実行できます。タスクの潜在的スターターまたは管理者だけが、タスクのインスタンスの (およびプロセス・テンプレートのインスタンスの) 作成を許可されます。

インライン呼び出しタスクが開始アクティビティーと関連付けられていない場合、または許可制限がタスクに指定されていない場合、誰でも開始アクティビティーを使用してプロセス・インスタンスを作成できます。

プロセスは、それぞれが異なる潜在的スターターまたは管理者に対する担当者照会を持つ、複数の開始アクティビティーを持つことができます。これはつまり、ユーザーがアクティビティー A を使用したプロセスの開始は許可されるが、アクティビティ B を使用しては許可されないということです。

### 手順

1. Business Flow Manager API を使用して、開始済み状態のプロセス・テンプレートの現行バージョンのリストを作成します。

   **ヒント:** queryProcessTemplates メソッドは、開始されていないアプリケーションの一部であるプロセス・テンプレートだけを除外します。そのため、結果をフィルター処理せずにこのメソッドを使用する場合、メソッドはどのような状態にあるかに関係なく、すべてのバージョンのプロセス・テンプレートを戻します。

   ```
   // current timestamp in UTC format, converted to yyyy-mm-ddThh:mm:ss
   String now = (new UTCDate()).toXsdString();
   String whereClause = "PROCESS_TEMPLATE.STATE =
                         PROCESS_TEMPLATE.STATE.STATE_STARTED AND
                         PROCESS_TEMPLATE.VALID_FROM =
                         (SELECT MAX(VALID_FROM) FROM PROCESS_TEMPLATE
                          WHERE NAME=PROCESS_TEMPLATE.NAME AND
                          VALID_FROM <= TS('" + now + "'))";

   ProcessTemplateData[] processTemplates = process.queryProcessTemplates
                                           ( whereClause,
                                             "PROCESS_TEMPLATE.NAME",
                                             (Integer)null, (TimeZone)null);
   ```

   結果はプロセス・テンプレート名でソートされます。
2. プロセス・テンプレートのリストと、ユーザーが許可されている開始アクティビティーのリストを作成します。

   プロセス・テンプレートのリストには、単一の開始アクティビティーがあるプロセス・テンプレートが含まれます。これらのアクティビティーは、保護されていないか、またはログオン・ユーザーによる開始が許可されていないかのいずれかです。あるいは、少なくとも 1 つの開始アクティビティーにより開始できるプロセス・テンプレートを収集することもできます。

**ヒント:** プロセス管理者は、プロセス・インスタンスを開始することもできます。テンプレートの完全なリストを入手するには、プロセス・テンプレートと関連する管理タスク・テンプレートを読み取り、ログオン・ユーザーが管理者であるかを確認することも必要です。

```
List authorizedProcessTemplates = new ArrayList();
List authorizedActivityServiceTemplates = new ArrayList();
```

3. プロセス・テンプレートごとに、開始アクティビティーを判別します。

```
for( int i=0; i<processTemplates.length; i++ )
{
  ProcessTemplateData template = processTemplates[i];
  ActivityServiceTemplateData[] startActivities =
                process.getStartActivities(template.getID());
```

4. 開始アクティビティーごとに、関連したインライン呼び出しタスク・テンプレートの ID を取得します。

```
for( int j=0; j<startActivities.length; j++ )
{
  ActivityServiceTemplateData activity = startActivities[j];
  TKTID tktid = activity.getTaskTemplateID();
```

   a. 呼び出しタスク・テンプレートが存在しない場合、プロセス・テンプレートはこの開始アクティビティーによっては保護されません。

      この場合、この開始アクティビティーを使用して、誰でもプロセス・インスタンスを作成できます。

```
boolean isAuthorized = false;
       if ( tktid == null )
    {
      isAuthorized = true;
      authorizedActivityServiceTemplates.add(activity);
    }
```

   b. 呼び出しタスク・テンプレートが存在する場合は、Human Task Manager API を使用して、ログオン・ユーザーの許可を検査します。

      例ではログオン・ユーザーは Smith です。ログオン・ユーザーは、呼び出しタスクの潜在的なスターターまたは管理者である必要があります。

```
if ( tktid != null )
    {
      isAuthorized =
        task.isUserInRole
            (tkid, "Smith", WorkItem.REASON_POTENTIAL_STARTER) ||
        task.isUserInRole(tktid, "Smith", WorkItem.REASON_ADMINISTRATOR);

      if ( isAuthorized )
      {
        authorizedActivityServiceTemplates.add(activity);
      }
    }
```

      ユーザーに指定されたロールがある場合、またはそのロールの担当者割り当て基準が指定されていない場合、isUserInRole メソッドは値 true を戻します。

5. プロセス・テンプレート名だけを使用してプロセスが開始できるかを確認します。

```
if ( isAuthorized && startActivities.length == 1 )
    {
      authorizedProcessTemplates.add(template);
    }
```

6. ループを終了します。

```
  } // end of loop for each activity service template
}   // end of loop for each process template
```

## ヒューマン・タスクを含む単一の個人ワークフローの処理

ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。この例では、一連のヒューマン・タスク・アクティビティー (予定タスク) として、書籍を注文するための一連のアクションを実装する方法を示しています。ワークフローの処理には、Business Flow Manager と Human Task Manager API の両方が使用されます。

### このタスクについて

オンライン・ブックストアでは、購入者は一連の操作を完了することで本を注文します。この一連の操作は、ヒューマン・タスク・アクティビティー (予定タスク) として実装できます。購入者が複数の書籍を注文する場合は、これが次のヒューマン・タスク・アクティビティーの要求に相当します。一連のタスクに関する情報は Business Flow Manager によって保守されますが、タスク自体は Human Task Manager によって保守されます。

この例を、Business Flow Manager API のみを使用する例と比較してください。

#### 手順

1. Business Flow Manager API を使用して、作業対象となるプロセス・インスタンスを取得します。

   この例では、CustomerOrder プロセスのインスタンスです。

```
ProcessInstanceData processInstance =
   process.getProcessInstance("CustomerOrder");
String piid = processInstance.getID().toString();
```

2. Human Task Manager API を使用して、指定されたプロセス・インスタンスの一部である、準備のできた予定タスク (種類は参加) を照会します。

   タスクの包含コンテキスト ID を使用して、含まれているプロセス・インスタンスを指定します。単一の個人ワークフローの場合、照会によって、一連のヒューマン・タスク・アクティビティーの最初のヒューマン・タスク・アクティビティーに関連付けられている予定タスクが戻されます。

```
//
 // Query the list of to-do tasks that can be claimed by the logged-on user
 // for the specified process instance
//
QueryResultSet result =
     task.query("DISTINCT TASK.TKIID",
                "TASK.CONTAINMENT_CTX_ID = ID('" + piid + "') AND
                 TASK.STATE = TASK.STATE.STATE_READY AND
                 TASK.KIND = TASK.KIND.KIND_PARTICIPATING AND
                 WORK_ITEM.REASON = WORK_ITEM.REASON.REASON_POTENTIAL_OWNER",
                (String)null, (Integer)null, (TimeZone)null);
```

3. 戻される予定タスクを要求します。

```
if (result.size() > 0)
{
 result.first();
 TKIID tkiid = (TKIID) result.getOID(1);
 ClientObjectWrapper input = task.claim(tkiid);
 DataObject activityInput = null ;
  if ( input.getObject()!= null && input.getObject() instanceof DataObject )
  {
    taskInput = (DataObject)input.getObject();
    // read the values
    ...
  }
}
```

タスクが要求されると、タスクの入力メッセージが戻されます。

4. 予定タスクに関連付けられたヒューマン・タスク・アクティビティーを判別します。

   以下のメソッドのいずれかを使用して、アクティビティーをそのタスクに相関させます。

   - task.getActivityID メソッド:

     ```
     AIID aiid = task.getActivityID(tkiid);
     ```

   - タスク・オブジェクトの一部である親コンテキスト ID:

     ```
     AIID aiid = null;
     Task taskInstance = task.getTask(tkiid);

     OID oid = taskInstance.getParentContextID();
     if ( oid != null and oid instanceof AIID )
     {
       aiid = (AIID)oid;
     }
     ```

5. タスクでの作業が終了したら、Business Flow Manager API を使用してタスクとそれに関連するヒューマン・タスク・アクティビティーを完了し、プロセス・インスタンス内の次のヒューマン・タスク・アクティビティーを要求します。

   ヒューマン・タスク・アクティビティーを完了するには、出力メッセージを渡します。出力メッセージを作成する場合、メッセージ・タイプ名を指定して、メッセージ定義が含まれるようにする必要があります。

   ```
   ActivityInstanceData activity = process.getActivityInstance(aiid);
   ClientObjectWrapper output =
         process.createMessage(aiid, activity.getOutputMessageTypeName());
   DataObject myMessage = null ;
   if ( output.getObject()!= null && output.getObject() instanceof DataObject )
   {
     myMessage = (DataObject)output.getObject();
     //set the parts in your message, for example, an order number
     myMessage.setInt("OrderNo", 4711);
   }

   //complete the human task activity and its associated to-do task,
   // and claim the next human task activity
   CompleteAndClaimSuccessorResult successor =
      process.completeAndClaimSuccessor(aiid, output);
   ```

   このアクションは、オーダー番号が含まれる出力メッセージを設定し、シーケンス内の次のヒューマン・タスク・アクティビティーを要求します。後続アクティビティーに AutoClaim が設定されており、有効なパスが複数存在する場合は、

後続アクティビティーのすべてが要求され、ランダムなアクティビティーが次のアクティビティーとして戻されます。このユーザーに割り当て可能な後続アクティビティーが他にない場合は、NULL が戻されます。

後に続くことができる並列パスがプロセスに含まれ、これらのパスに、ログオン・ユーザーが潜在的な所有者であるヒューマン・タスク・アクティビティーが複数含まれる場合、ランダムなアクティビティーが自動的に要求され、次のアクティビティーとして戻されます。

6. 次のヒューマン・タスク・アクティビティーを処理します。

```
ClientObjectWrapper nextInput = successor.getInputMessage();
if ( nextInput.getObject()!=
               null && nextInput.getObject() instanceof DataObject )
{
  activityInput = (DataObject)input.getObject();
  // read the values
  ...
}

aiid = successor.getAIID();
```

7. ステップ 5 を続行して、ヒューマン・タスク・アクティビティーを完了し、次のヒューマン・タスク・アクティビティーを取得します。

**関連タスク**

244 ページの『単独ユーザー・ワークフローの処理』
ワークフローの中には、1 人のユーザーだけで実行されるものがあります。例えば、オンライン・ブックストアでの本の注文などです。このタイプのワークフローには、並列パスは存在しません。 completeAndClaimSuccessor API は、このタイプのワークフローの処理をサポートします。

## 例外および障害の処理

BPEL プロセスでは、プロセス内のさまざまな箇所で障害が発生する可能性があります。

### このタスクについて

Business Process Execution Language (BPEL) の障害には次の発生原因があります。

- Web サービスの呼び出し (Web Services Description Language (WSDL) の障害)
- throw アクティビティー
- Business Process Choreographer によって認識される BPEL 標準障害

これらの障害を処理する機構が存在します。プロセス・インスタンスによって生成される障害を処理するには、以下の手段のいずれかを使用します。

- 対応する障害ハンドラーへの制御の引き渡し
- プロセス内の前の処理の補正
- プロセスの停止と状態の修復の依頼 (強制再試行、強制完了)

BPEL プロセスは、プロセスによって指定される操作の呼び出し元に障害を戻す可能性もあります。プロセス内の障害は、障害名と障害データを指定して reply アクティビティーとしてモデル化できます。これらの障害は、チェック例外として API 呼び出し元に戻されます。

BPEL プロセスで BPEL 障害を処理しない場合、または API 例外が発生した場合は、ランタイム例外が API 呼び出し元に戻されます。API 例外の例には、インスタンスの作成元となるプロセス・モデルが存在しない場合などがあります。

障害および例外の処理は、以下のタスクで説明します。

## API 例外の処理

BusinessFlowManagerService インターフェースまたは HumanTaskManagerService インターフェースのメソッドが正常に完了しない場合、エラーの原因を示す例外がスローされます。この例外を特別に処理して、呼び出し元にガイダンスを提供することができます。

### このタスクについて

ただし、例外のサブセットのみを特別に処理し、その他の潜在的な例外に対しては汎用のガイダンスを提供するのが一般的です。すべての固有の例外は、汎用の ProcessException または TaskException から継承しています。最終の `catch(ProcessException)` または `catch(TaskException)` ステートメントを使用して汎用の例外を catch することが*最良実例* です。このステートメントでは、発生する可能性があるその他すべての例外を考慮に入れるため、このステートメントによって、ご使用のアプリケーション・プログラムの上位互換性を確保することができます。

## ヒューマン・タスク・アクティビティーに設定された障害の検査

ヒューマン・タスク・アクティビティーが処理されると、そのヒューマン・タスク・アクティビティーは正常に完了できます。この場合は、出力メッセージを渡すことができます。ヒューマン・タスク・アクティビティーが正常に完了しない場合は、障害メッセージを渡すことができます。

### このタスクについて

障害メッセージを読んでエラーの原因を調べることができます。

**手順**

1. 失敗状態または停止状態のタスク・アクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "(ACTIVITY.STATE = ACTIVITY.STATE.STATE_FAILED OR
                     ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED) AND
                     ACTIVITY.KIND=ACTIVITY.KIND.KIND_STAFF",
                     (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、失敗または停止のアクティビティーが含まれる照会結果セットを戻します。

2. 障害の名前を読み取ります。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ClientObjectWrapper faultMessage = process.getFaultMessage(aiid);
  DataObject fault = null ;
  if ( faultMessage.getObject() != null && faultMessage.getObject()
       instanceof DataObject )
```

```
  {
    fault = (DataObject)faultMessage.getObject();
    Type type = fault.getType();
    String name = type.getName();
    String uri = type.getURI();
  }
}
```

これは、障害名を戻します。また、障害名を取得する代わりに、停止アクティビティーの未処理の例外を分析することもできます。

## 停止した invoke アクティビティーで発生した障害の検査

適切に設計されたプロセスでは、例外および障害は、通常は障害ハンドラーによって処理されます。invoke アクティビティーで発生した例外または障害に関する情報は、アクティビティー・インスタンスから取得できます。

### このタスクについて

アクティビティーで障害が発生した場合、障害タイプによってそのアクティビティーの修復のために実行できるアクションが決まります。

**手順**

1. 停止状態のヒューマン・タスク・アクティビティーをリストします。

```
QueryResultSet result =
     process.query("ACTIVITY.AIID",
                   "ACTIVITY.STATE = ACTIVITY.STATE.STATE_STOPPED AND
                    ACTIVITY.KIND=ACTIVITY.KIND.KIND_INVOKE",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、停止された invoke アクティビティーが含まれる照会結果セットを戻します。

2. 障害の名前を読み取ります。

```
if (result.size() > 0)
{
  result.first();
  AIID aiid = (AIID) result.getOID(1);
  ActivityInstanceData activity = process.getActivityInstance(aiid);

  ProcessException excp = activity.getUnhandledException();
  if ( excp instanceof ApplicationFaultException )
  {
   ApplicationFaultException fault = (ApplicationFaultException)excp;
   String faultName = fault.getFaultName();
  }
}
```

## 失敗プロセス・インスタンスで発生した未処理の例外または障害の検査

適切に設計されたプロセスでは、例外および障害は、通常は障害ハンドラーによって処理されます。プロセスが両方向操作を実装する場合、障害または処理済みの例外に関する情報は、プロセス・インスタンス・オブジェクトの障害名プロパティーから取得できます。障害の場合は、getFaultMessage API を使用して、対応する障害メッセージを取得することもできます。

### このタスクについて

例外がどの障害ハンドラーにも処理されないことが原因でプロセス・インスタンスが失敗した場合は、未処理の例外に関する情報をプロセス・インスタンス・オブジェクトから取得できます。これとは対照的に、障害が障害ハンドラーによってキャッチされた場合、その障害に関する情報は入手できません。ただし、FaultReplyException 例外を使用することにより、障害名および障害メッセージを取得して、呼び出し元に戻ることができます。

**手順**

1. 失敗状態にあるプロセス・インスタンスをリストします。

```
QueryResultSet result =
     process.query("PROCESS_INSTANCE.PIID",
                   "PROCESS_INSTANCE.STATE =
                       PROCESS_INSTANCE.STATE.STATE_FAILED",
                    (String)null, (Integer)null, (TimeZone)null);
```

   このアクションは、失敗プロセス・インスタンスが含まれる照会結果セットを戻します。

2. 未処理の例外に関する情報を読み取ります。

```
if (result.size() > 0)
{
  result.first();
  PIID piid = (PIID) result.getOID(1);
  ProcessInstanceData pInstance = process.getProcessInstance(piid);

  ProcessException excp = pInstance.getUnhandledException();
  if ( excp instanceof RuntimeFaultException )
  {
   RuntimeFaultException xcp = (RuntimeFaultException)excp;
   Throwable cause = xcp.getRootCause();
  }
  else if ( excp instanceof StandardFaultException )
  {
   StandardFaultException xcp = (StandardFaultException)excp;
   String faultName = xcp.getFaultName();
  }
  else if ( excp instanceof ApplicationFaultException )
  {
   ApplicationFaultException xcp = (ApplicationFaultException)excp;
   String faultName = xcp.getFaultName();
  }
}
```

### タスクの結果

この情報を使用して、障害名または問題の根本原因を調べます。

## Web サービス API クライアント・アプリケーションの開発

Web サービス API を介してビジネス・プロセス・アプリケーションとヒューマン・タスク・アプリケーションにアクセスするクライアント・アプリケーションを開発できます。

### このタスクについて

クライアント・アプリケーションは、Java Web サービスや Microsoft .NET などのすべての Web サービス・クライアント環境で開発できます。

**関連概念**

187 ページの『ビジネス・プロセスおよびヒューマン・タスクと対話するためのプログラミング・インターフェースの比較』
ビジネス・プロセスおよびヒューマン・タスクと対話するクライアント・アプリケーションの作成には、Enterprise JavaBeans (EJB)、Web サービス、Java Message Service (JMS) および Representational State Transfer (REST) サービスなどの汎用プログラミング・インターフェースを使用できます。これらのインターフェースには、それぞれ異なる特性があります。

## Web サービス・コンポーネントおよび一連の制御

多くのクライアント・サイドおよびサーバー・サイドのコンポーネントは、Web サービスの要求と応答を表す一連の制御に関与します。

標準的な一連の制御は以下のとおりです。

1. クライアント・サイド:
   a. クライアント・アプリケーション (ユーザーによって提供される) は、Web サービスの要求を発行します。
   b. プロキシー・クライアント (ユーザーによっても提供されるが、クライアント・サイド・ユーティリティーを使用して自動的に生成することが可能) は、サービス要求を SOAP 要求エンベロープでラップします。
   c. クライアント・サイド開発インフラストラクチャーは、Web サービスのエンドポイントとして定義された URL に要求を転送します。
2. ネットワークは、HTTP または HTTPS を使用して Web サービス・エンドポイントに要求を送信します。
3. サーバー・サイド:
   a. 汎用 Web サービス API は、要求を受信し、デコードします。
   b. 要求は、汎用の Business Flow Manager または Human Task Manager コンポーネントによって直接処理されるか、指定されたビジネス・プロセスまたはヒューマン・タスクに転送されます。
   c. 戻されたデータは SOAP 応答エンベロープでラップされます。
4. ネットワークは、HTTP または HTTPS を使用してクライアント・サイド環境に応答を送信します。
5. クライアント・サイドに戻る:
   a. クライアント・サイド開発インフラストラクチャーは、SOAP 応答エンベロープをアンラップします。
   b. プロキシー・クライアントはデータを SOAP 応答から抽出して、それをクライアント・アプリケーションに受け渡します。
   c. クライアント・アプリケーションは、必要に応じて、戻されたデータを処理します。

# Web サービス API の概要

Web サービス API により、Business Process Choreographer 環境で実行しているビジネス・プロセスおよびヒューマン・タスクに Web サービスを使用してアクセスするクライアント・アプリケーションを開発できます。

Business Process Choreographer Web サービス API は、次の 2 つの別々の Web サービス・インターフェース (WSDL ポート・タイプ) を提供します。

- Business Flow Manager API。クライアント・アプリケーションが microflow および長期間のプロセスと対話できるようにします。例えば、以下の操作を実行できます。
  - プロセス・テンプレートとプロセス・インスタンスを作成
  - 既存のプロセスの要求
  - ID によるプロセスの照会

  実行可能なアクションの詳細なリストについては、 231 ページの『ビジネス・プロセス用のアプリケーションの開発』を参照してください。
- Human Task Manager API。クライアント・アプリケーションは以下の操作を実行できます。
  - タスクの作成と開始
  - 既存のタスクの要求
  - タスクの完了
  - ID によるタスクの照会
  - タスクの集合の照会

  実行可能なアクションの詳細なリストについては、 254 ページの『ヒューマン・タスク用のアプリケーションの開発』を参照してください。

クライアント・アプリケーションは、これらの Web サービス・インターフェースの一方または両方を使用できます。

## 例

Human Task Manager Web サービス API にアクセスして参加ヒューマン・タスクを処理するクライアント・アプリケーションの場合に考えられる処理の概要を次に示します。

1. クライアント・アプリケーションは、query Web サービス呼び出しを WebSphere Process Server に対して発行します。これにより、ユーザーによって処理される参加タスクのリストを要求します。
2. 参加タスクのリストが、SOAP/HTTP 応答エンベロープで戻されます。
3. クライアント・アプリケーションは、claim Web サービス呼び出しを発行して、いずれかの参加タスクを要求します。
4. WebSphere Process Server は、タスクの入力メッセージを戻します。
5. クライアント・アプリケーションは、complete Web サービス呼び出しを発行して、出力メッセージまたは障害メッセージのあるタスクを完了します。

## ビジネス・プロセスとヒューマン・タスクの要件

WebSphere Integration Developer により Business Process Choreographer で実行するために開発されるビジネス・プロセスとヒューマン・タスクは、Web サービス API を介してアクセスできるようにするために特定の規則に準拠する必要があります。

要件は以下のとおりです。

1. ビジネス・プロセスとヒューマン・タスクのインターフェースは、Java API for XML-based RPC (JAX-RPC 1.1) 仕様で定義される「document/literal wrapped」スタイルを使用して定義する必要があります。これは、WID によって開発されるすべてのビジネス・プロセスとヒューマン・タスクのデフォルト・スタイルです。
2. Web サービス操作のビジネス・プロセスとヒューマン・タスクによって公開される障害メッセージは、XML スキーマ・エレメントにより定義される単一の WSDL メッセージ部を構成する必要があります。以下に例を挙げます。

   ```
   <wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
   ```

   **関連情報**

   Java API for XML based RPC (JAX-RPC) のダウンロード・ページ

   Which style of WSDL should I use?

## クライアント・アプリケーションの開発

クライアント・アプリケーションの開発プロセスは多数のステップで構成されています。

**手順**

1. Business Flow Manager API、Human Task Manager API、またはその両方の中から、クライアント・アプリケーションで使用する必要がある Web サービス API を決定します。
2. WebSphere Process Server 環境から必要なファイルをエクスポートします。 あるいは、WebSphere Process Server クライアント CD からファイルをコピーすることもできます。
3. 選択したクライアント・アプリケーション開発環境で、エクスポートした成果物を使用してプロキシー・クライアントを生成します。
4. オプション: ヘルパー・クラス を生成します。 ヘルパー・クラスは、クライアント・アプリケーションが WebSphere サーバーの具象プロセスまたはタスクと直接に対話する場合に必要とされます。ただし、クライアント・アプリケーションが汎用タスク (照会の発行など) を実行するだけの場合は、ヘルパー・クラスは不要です。
5. クライアント・アプリケーション用のコードを開発します。
6. 必要なセキュリティー・メカニズムをクライアント・アプリケーションに追加します。

## 成果物のコピー

クライアント・アプリケーションを作成しやすくするには、WebSphere 環境から数多くの成果物をコピーする必要があります。

これらの成果物は、次の 2 つの方法で入手できます。

- 成果物を WebSphere Process Server 環境で公開してエクスポートします。
- WebSphere Process Server クライアント CD からファイルをコピーします。

## サーバー環境での成果物の公開とエクスポート

クライアント・アプリケーションを開発して Web サービス API にアクセスするには、WebSphere サーバー環境であらかじめ数多くの成果物を公開し、エクスポートしておく必要があります。

### このタスクについて

エクスポートする成果物は以下のとおりです。

- Web サービス API を構成するポート・タイプと操作について記述した Web Service Definition Language (WSDL) ファイル。
- WSDL ファイル内のサービスとメソッドによって参照されるデータ型の定義を記述した XML Schema Definition (XSD) ファイル。
- ビジネス・オブジェクトを記述した追加の WSDL ファイルと XSD ファイル。ビジネス・オブジェクトは、WebSphere サーバーで実行される具象ビジネス・プロセスまたはヒューマン・タスクについて記述します。これらの追加ファイルは、クライアント・アプリケーションが Web サービス API を介して具象ビジネス・プロセスまたはヒューマン・タスクと直接に対話する必要がある場合にのみ必要とされます。クライアント・アプリケーションが、照会の発行などの汎用タスクを実行するだけの場合は不要です。

これらの成果物を公開した後は、それらの成果物をクライアント・プログラミング環境にコピーし、プロキシー・クライアントおよびヘルパー・クラスの生成で使用できるようにする必要があります。

#### Web サービス・エンドポイント・アドレスの指定:

Web サービス・エンドポイント・アドレスは、クライアント・アプリケーションが Web サービス API にアクセスするために指定する必要のある URL です。エンドポイント・アドレスは、クライアント・アプリケーション用のプロキシー・クライアントを生成するためにエクスポートする WSDL ファイルに書き込まれます。

#### このタスクについて

使用する Web サービス・エンドポイント・アドレスは、WebSphere サーバーの構成によって異なります。

- シナリオ 1。単一の WebSphere サーバー。指定する WebSphere エンドポイント・アドレスは、サーバーのホスト名とポート番号です (例: **host1:9080**)。
- シナリオ 2。複数のサーバーで構成される WebSphere クラスターの場合。指定する WebSphere エンドポイント・アドレスは、Web サービス API をホスティングするサーバーのホスト名とポートです (例: **host2:9081**)。
- シナリオ 3。Web サーバーをフロントエンドとして使用する場合。指定する WebSphere エンドポイント・アドレスは、Web サーバーのホスト名とポートです (例: **host:80**)。

デフォルトでは、Web サービス・エンドポイント・アドレスの形式は *protocol*://*host*:*port*/*context_root*/*fixed_path* です。各部の意味は、次のとおりです。

- *protocol*。クライアント・アプリケーションと WebSphere サーバー間で使用される通信プロトコル。デフォルト・プロトコルは HTTP です。代わりに、より安全な HTTPS (HTTP over SSL) プロトコルを使用する方法を選択できます。HTTPS の使用をお勧めします。
- *host:port*。Web サービス API をホスティングするマシンへのアクセスに使用するホスト名とポート番号。この値は、クライアント・アプリケーションがアプリケーションに直接アクセスするか、Web サーバー・フロントエンド経由でアクセスするかなど、場合によって異なります。
- *context_root*。「コンテキスト・ルート」には、任意の値を選択できます。ただし、選択する値は個々の WebSphere セル内で固有でなければなりません。デフォルト値は、「node_server/cluster」サフィックスを使用して、ネーミング競合のリスクを回避しています。
- *fixed_path* は、`/sca/com/ibm/bpe/api/BFMWS` (Business Flow Manager API の場合) または `/sca/com/ibm/task/api/HTMWS` (Human Task Manager API の場合) で、変更できません。

Web サービス・エンドポイント・アドレスは、最初は、ビジネス・プロセス・コンテナーまたはヒューマン・タスク・コンテナーの構成時に指定されます。

**手順**

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」を選択します。

   **注:** 「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」を選択して、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから、「**BPEContainer**」(ビジネス・プロセス・コンテナーの場合) または「**TaskContainer**」(ヒューマン・タスク・コンテナーの場合) を選択します。
4. 「**追加プロパティー**」リストから、「**HTTP エンドポイント URL 情報を提供 (Provide HTTP endpoint URL information)**」を選択します。
5. リストからデフォルトのプレフィックスのいずれか 1 つを選択するか、カスタム・プレフィックスを入力します。 クライアント・アプリケーションを、Web サービス API をホスティングするアプリケーション・サーバーに直接接続する場合は、デフォルト・プレフィックス・リストのプレフィックスを使用します。そうでない場合は、カスタム・プレフィックスを指定します。
6. 「**適用**」をクリックして、選択したプレフィックスを SCA モジュールにコピーします。
7. 「**OK**」をクリックします。 URL 情報がワークスペースに保管されます。

**タスクの結果**

管理コンソールで現行値を表示できます (例えばビジネス・プロセス・コンテナーの場合は、「**エンタープライズ・アプリケーション**」 → 「**BPEContainer**」 → 「**デプロイメント記述子の表示**」)。

エクスポートされた WSDL ファイルでは、soap:address エレメントの location 属性に、指定した Web サービス・エンドポイント・アドレスが含まれています。以下に例を示します。

```
<wsdl:service name="BFMWSService">
  <wsdl:port name="BFMWSPort" binding="this:BFMWSBinding">
   <soap:address location=
         "https://myserver:9080/WebServicesAPIs/sca/com/ibm/bpe/api/BFMWS"/>
```

**WSDL ファイルの公開:**

Web サービス記述言語 (WSDL) ファイルには、Web サービス API で使用できるすべての操作の詳細な説明が含まれています。Business Flow Manager と Human Task Manager の Web サービス API では、別々の WSDL ファイルを使用できます。これらの WSDL ファイルは、まず公開して、その後 WebSphere 環境からご使用の開発環境にコピーし、プロキシー・クライアントの生成に使用することになります。

**始める前に**

WSDL ファイルを公開する前に、指定した Web サービス・エンドポイント・アドレスが正しいことを確認してください。このアドレスは、クライアント・アプリケーションが Web サービス API へのアクセスに使用する URL です。

**このタスクについて**

WSDL ファイルの公開が必要なのは一度だけです。

**注:** WebSphere Process Server クライアント CD を所有している場合は、代わりに、その CD からクライアント・プログラミング環境に直接ファイルをコピーすることもできます。

***ビジネス・プロセス WSDL の公開:***

管理コンソールを使用して WSDL ファイルを公開します。

**手順**

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」を選択します。

   **注:** 「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」を選択して、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから **BPEContainer** アプリケーションを選択します。
4. 「**追加プロパティー**」のリストから、「**WSDL ファイルの公開**」を選択します。
5. リスト中の ZIP ファイルをクリックします。
6. 表示されるファイルのダウンロード・ウィンドウで、「**保管**」をクリックします。
7. ローカル・フォルダーを参照し、「**保管**」をクリックします。

**タスクの結果**

エクスポートされる ZIP ファイルは、BPEContainer_WSDLFiles.zip と命名されます。 ZIP ファイルには、Web サービスを記述した WSDL ファイルと、WSDL ファイル内から参照されるすべての XSD ファイルが含まれています。

***ヒューマン・タスク WSDL の公開:***

管理コンソールを使用して WSDL ファイルを公開します。

**手順**

1. 管理者権限のあるユーザー ID で、管理コンソールにログオンします。
2. 「**アプリケーション**」 → 「**SCA モジュール**」を選択します。

   **注:** 「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」を選択して、使用可能なすべてのエンタープライズ・アプリケーションのリストを表示することもできます。
3. SCA モジュールまたはアプリケーションのリストから **TaskContainer** アプリケーションを選択します。
4. 「**追加プロパティー**」のリストから、「**WSDL ファイルの公開**」を選択します。
5. リスト中の ZIP ファイルをクリックします。
6. 表示されるファイルのダウンロード・ウィンドウで、「**保管**」をクリックします。
7. ローカル・フォルダーを参照し、「**保管**」をクリックします。

**タスクの結果**

エクスポートされる ZIP ファイルは、TaskContainer_WSDLFiles.zip と命名されます。 ZIP ファイルには、Web サービスを記述した WSDL ファイルと、WSDL ファイル内から参照されるすべての XSD ファイルが含まれています。

**ビジネス・オブジェクトのエクスポート:**

ビジネス・プロセスおよびヒューマン・タスクには、Web サービスとして外部にアクセスできるよう明確に定義されたインターフェースが用意されています。これらのインターフェースがビジネス・オブジェクトを参照する場合は、インターフェース定義とビジネス・オブジェクトをクライアント・プログラミング環境にエクスポートする必要があります。

**このタスクについて**

この手順は、クライアント・アプリケーションが対話する必要があるビジネス・オブジェクトごとに、繰り返してください。

WebSphere Process Server では、ビジネス・オブジェクトが、ビジネス・プロセスまたはヒューマン・タスクと対話する要求、応答、および障害メッセージの形式を定義します。これらのメッセージには、複合データ型の定義が含まれている場合もあります。

例えば、ヒューマン・タスクを作成して開始するには、以下の情報項目をタスク・インターフェースに渡す必要があります。

- タスク・テンプレート名
- タスク・テンプレート名前空間
- 入力メッセージ (フォーマット済みのビジネス・データを含む)
- 応答メッセージを戻すための応答ラッパー
- 障害および例外を戻すための障害メッセージ

以上の項目は、単一のビジネス・オブジェクト内でカプセル化されます。Web サービス・インターフェースのすべての操作は、「document/literal wrapped」操作としてモデル化されます。これらの操作の入出力パラメーターは、ラッパー文書の中にカプセル化されます。他のビジネス・オブジェクトは、それに対応する応答および障害のメッセージ形式を定義します。

Web サービスによってビジネス・プロセスまたはヒューマン・タスクを作成および開始するためには、クライアント・サイドのクライアント・アプリケーションがこれらのラッパー・オブジェクトを使用できるようにする必要があります。

そのためには、WebSphere 環境からビジネス・オブジェクトを Web サービス記述言語 (WSDL) ファイルおよび XML スキーマ定義 (XSD) ファイルとしてエクスポートし、データ型定義をクライアント・プログラミング環境にインポートしてから、それをクライアント・アプリケーションで使用できるヘルパー・クラスに変換します。

**手順**

1. WebSphere Integration Developer ワークスペースが稼働していない場合は、起動します。
2. エクスポートするビジネス・オブジェクトを含むライブラリー・モジュールを選択します。 ライブラリー・モジュールは、必要なビジネス・オブジェクトを含む圧縮ファイルです。
3. ライブラリー・モジュールをエクスポートします。
4. エクスポートしたファイルをクライアント・アプリケーション開発環境にコピーします。

**例**

ビジネス・プロセスが以下の Web サービス操作を公開するとします。

```
<wsdl:operation name="updateCustomer">
    <wsdl:input message="tns:updateCustomerRequestMsg"
                name="updateCustomerRequest"/>
    <wsdl:output message="tns:updateCustomerResponseMsg"
                 name="updateCustomerResponse"/>
    <wsdl:fault message="tns:updateCustomerFaultMsg"
                name="updateCustomerFault"/>
  </wsdl:operation>
```

公開は、以下のように定義された WSDL メッセージを介して行われるとします。

```
<wsdl:message name="updateCustomerRequestMsg">
    <wsdl:part element="types:updateCustomer"
               name="updateCustomerParameters"/>
  </wsdl:message>
```

```
<wsdl:message name="updateCustomerResponseMsg">
  <wsdl:part element="types:updateCustomerResponse"
             name="updateCustomerResult"/>
</wsdl:message>
<wsdl:message name="updateCustomerFaultMsg">
  <wsdl:part element="types:updateCustomerFault"
             name="updateCustomerFault"/>
</wsdl:message>
```

*具象* ユーザー定義エレメント「types:updateCustomer」、「types:updateCustomerResponse」、および「types:updateCustomerFault」は、クライアント・アプリケーションが実行するすべての*汎用* 命令 (call、sendMessage など) で、<xsd:any> パラメーターを使用して Web サービス API に渡したり Web サービス API から受け取ったりする必要があります。これらのユーザー定義エレメントは、エクスポートされた XSD ファイルから生成されるヘルパー・クラスを使用して、クライアント・アプリケーション上で作成、直列化、および非直列化されます。

## クライアント CD でのファイルの使用

WebSphere サーバー環境から成果物をエクスポートする代わりに、クライアント・アプリケーションの生成に必要なファイルを WebSphere Process Server クライアント CD からコピーすることができます。

この場合は、Business Flow Manager API または Human Task Manager API のデフォルトの Web サービス・エンドポイント・アドレスを手動で変更する必要があります。

クライアント・アプリケーションが両方の API にアクセスする場合は、両方の API のデフォルト・エンドポイント・アドレスを編集する必要があります。

**クライアント CD からのファイルのコピー:**

Web サービス API へのアクセスに必要なファイルは、WebSphere Process Server クライアント CD に収録されています。

**手順**

1. クライアント CD にアクセスして、ProcessChoreographer¥client ディレクトリーを参照します。
2. 必要なファイルをクライアント・アプリケーション開発環境にコピーします。

   Business Flow Manager API の場合は、次のファイルをコピーします。

   **BFMWS.wsdl**
   Business Flow Manager Web サービス API で使用可能な Web サービスが記述されています。このファイルには、エンドポイントのアドレスが含まれています。

   **BFMIF.wsdl**
   Business Flow Manager Web サービス API の各 Web サービスのパラメーターとデータ型が記述されています。

   **BFMIF.xsd**
   Business Flow Manager Web サービス API で使用されるデータ型が記述されています。

**BPCGEN.xsd**
Business Flow Manager と Human Task Manager の Web サービス API の間で共通のデータ型が記述されています。

Human Task Manager API の場合は、次のファイルをコピーします。

**HTMWS.wsdl**
Human Task Manager Web サービス API で使用可能な Web サービスが記述されています。このファイルには、エンドポイントのアドレスが含まれています。

**HTMIF.wsdl**
Human Task Manager Web サービス API の各 Web サービスのパラメーターとデータ型が記述されています。

**HTMIF.xsd**
Human Task Manager Web サービス API で使用されるデータ型が記述されています。

**BPCGEN.xsd**
Business Flow Manager と Human Task Manager の Web サービス API の間で共通のデータ型が記述されています。

**注:** BPCGen.xsd ファイルは、両方の API で共通です。

### 次のタスク

各ファイルをコピーした後は、BFMWS.wsdl または HTMWS.wsdl ファイルの Web サービス API のエンドポイント・アドレスを、Web サービス API をホストする WebSphere アプリケーション・サーバーのエンドポイント・アドレスに手動で変更する必要があります。

### Web サービス・エンドポイント・アドレスの手動変更:

クライアント CD からファイルをコピーする場合は、WSDL ファイルで指定されるデフォルトの Web サービスのエンドポイント・アドレスを、Web サービス API をホスティングするサーバーのエンドポイント・アドレスに変更する必要があります。

### このタスクについて

管理コンソールを使用すると、WSDL ファイルをエクスポートする前に Web サービス・エンドポイント・アドレスを設定することができます。ただし、WSDL ファイルを WebSphere Process Server クライアント CD からコピーする場合は、デフォルトの Web サービス・エンドポイント・アドレスを手動で変更する必要があります。

使用する Web サービス・エンドポイント・アドレスは、WebSphere サーバーの構成によって異なります。

- シナリオ 1。WebSphere サーバーが 1 台だけの場合。指定する WebSphere エンドポイント・アドレスは、サーバーのホスト名とポート番号です (例: **host1:9080**)。

- シナリオ 2。複数のサーバーで構成される WebSphere クラスターの場合。指定する WebSphere エンドポイント・アドレスは、Web サービス API をホスティングするサーバーのホスト名とポートです (例: **host2:9081**)。
- シナリオ 3。Web サーバーをフロントエンドとして使用する場合。指定する WebSphere エンドポイント・アドレスは、Web サーバーのホスト名とポートです (例: **host:80**)。

***Business Flow Manager API エンドポイントの変更:***

Business Flow Manager API ファイルを WebSphere Process Server クライアント CD からコピーする場合は、デフォルトのエンドポイント・アドレスを手動で編集する必要があります。

**手順**

1. クライアント CD からコピーしたファイルが格納されているディレクトリーにナビゲートします。
2. テキスト・エディターか XML エディターで BFMWS.wsdl ファイルを開きます。
3. `soap:address` エレメントを見つけます (ファイル下部)。
4. `location` 属性の値を、Web サービス API が実行されているサーバーの HTTP URL に変更します。 変更するには、次のステップを実行します。
   a. オプションとして `http` を `https` に置き換え、より安全な HTTPS プロトコルを使用できます。
   b. *localhost* を、Web サービス API サーバーのエンドポイント・アドレスのホスト名または IP アドレスに置き換えます。
   c. *9080* を、アプリケーション・サーバーのポート番号に置き換えます。
   d. *BPEContainer_N1_server1* を、Web サービス API を実行しているアプリケーションのコンテキスト・ルートで置換します。 デフォルトのコンテキスト・ルートは、次のエレメントで構成されています。
      - *BPEContainer*。アプリケーション名。
      - *N1*。ノード名。
      - *server1*。サーバー名。
   e. URL の固定部 (`/sca/com/ibm/bpe/api/BFMWS`) は変更しないでください。

   例えば、アプリケーションがサーバー **s1.n1.ibm.com** で実行されており、そのサーバーが **9080** ポートで SOAP/HTTP 要求を受け入れている場合は、`soap:address` エレメントを次のように変更します。

```
<soap:address location="http://si.n1.ibm.com:9080/
                  BPEContainer_N1_server1/sca/com/ibm/bpe/api/BFMWS"/>
```

***Human Task Manager API エンドポイントの変更:***

Human Task Manager API ファイルを WebSphere Process Server クライアント CD からコピーする場合は、デフォルトのエンドポイント・アドレスを手動で編集する必要があります。

**手順**

1. クライアント CD からコピーしたファイルが格納されているディレクトリーにナビゲートします。
2. テキスト・エディターか XML エディターで HTMWS.wsdl ファイルを開きます。
3. soap:address エレメントを見つけます (ファイル下部)。
4. location 属性の値を、正しいエンドポイント・アドレスに変更します。 変更するには、次のステップを実行します。
   a. オプションとして http を https に置き換え、より安全な HTTPS プロトコルを使用できます。
   b. *localhost* を、Web サービス API サーバーのエンドポイント・アドレスのホスト名または IP アドレスに置き換えます。
   c. *9080* を、アプリケーション・サーバーのポート番号に置き換えます。
   d. *HTMContainer_N1_server1* を、Web サービス API を実行しているアプリケーションのコンテキスト・ルートで置換します。 デフォルトのコンテキスト・ルートは、次のエレメントで構成されています。
      - *HTMContainer*。アプリケーション名。
      - *N1*。ノード名。
      - *server1*。サーバー名。
   e. URL の固定部 (/sca/com/ibm/task/api/HTMWS) は変更しないでください。

   例えば、アプリケーションがサーバー **s1.n1.ibm.com** で実行されており、そのサーバーが **9081** ポートで SOAP/HTTPS 要求を受け入れている場合は、soap:address エレメントを次のように変更します。

```
<soap:address location="https://si.n1.ibm.com:9081/
                  HTMContainer_N1_server1/sca/com/ibm/task/api/HTMWS"/>
```

# Java Web サービス環境でのクライアント・アプリケーションの開発

Java Web サービスと互換性のある Java ベースの開発環境を使用して、Web サービス API 用のクライアント・アプリケーションを開発できます。

## プロキシー・クライアントの生成 (Java Web サービス)

Java Web サービス・クライアント・アプリケーションは、*プロキシー・クライアント* を使用して Web サービス API と対話します。

### このタスクについて

Java Web サービスのプロキシー・クライアントには、Web サービス要求を実行するためにクライアント・アプリケーションが呼び出す Java Bean クラスが数多く含まれています。プロキシー・クライアントは、サービス・パラメーターを SOAP メッセージにアセンブルし、SOAP メッセージを HTTP 経由で Web サービスに送信し、Web サービスから応答を受信し、戻されたデータをクライアント・アプリケーションに引き渡します。

したがって、基本的にプロキシー・クライアントによってクライアント・アプリケーションは、ローカル機能のようにして Web サービスを呼び出すことができます。

**注:** プロキシー・クライアントの生成が必要なのは一度だけです。その後、同じ Web サービス API にアクセスするすべてのクライアント・アプリケーションは、同じプロキシー・クライアントを使用できます。

IBM Web サービス環境では、次の 2 つの方法でプロキシー・クライアントを生成します。

- Rational® Application Developer または WebSphere Integration Developer が統合された開発環境を使用する。
- WSDL2Java コマンド行ツールを使用する。

その他の Java Web サービス開発環境には通常、WSDL2Java ツールまたは独自のクライアント・アプリケーション生成機能が組み込まれています。

**Rational Application Developer によるプロキシー・クライアントの生成:**

Rational Application Developer の統合された開発環境により、クライアント・アプリケーションのプロキシー・クライアントを生成できます。

**始める前に**

プロキシー・クライアントを生成するには、その前に、ビジネス・プロセスまたはヒューマン・タスクの Web サービス・インターフェースを記述した WSDL ファイルを WebSphere 環境 (または WebSphere Process Server クライアント CD) からエクスポートし、それをクライアントのプログラミング環境にコピーしておく必要があります。

**手順**

1. 該当する WSDL ファイルをプロジェクトに追加します。
   - ビジネス・プロセスの場合:
     a. エクスポート・ファイル BPEContainer_*nodename_servername*_WSDLFiles.zip を一時ディレクトリーに unzip します。
     b. サブディレクトリー META-INF を、unzip されたディレクトリー BPEContainer_*nodename_servername*.ear/b.jar からインポートします。
   - ヒューマン・タスクの場合:
     a. エクスポート・ファイル TaskContainer_*nodename_servername*_WSDLFiles.zip を一時ディレクトリーに unzip します。
     b. サブディレクトリー META-INF を、unzip されたディレクトリー TaskContainer_*nodename_servername*.ear/h.jar からインポートします。

   新規ディレクトリー wsdl およびサブディレクトリー構造がプロジェクト内に作成されます。
2. Web サービス・ウィザード・プロパティーを変更します。
   a. Rational Application Developer で、「**設定**」 → 「**Web サービス**」 → 「**コード生成**」 → 「**IBM WebSphere ランタイム**」を選択します。
   b. 「**ラップ・スタイルを使用せずに WSDL から Java を生成 (Generate Java from WSDL using the no wrapped style)**」オプションを選択します。

**注:** 「**設定**」メニューで「**Web サービス**」オプションを選択できない場合は、まず「**ウィンドウ**」 → 「**設定**」 → 「**ワークベンチ**」 → 「**機能**」と進んで、必要な機能を有効にする必要があります。「**Web サービス開発者 (Web Service Developer)**」をクリックして、「**OK**」をクリックします。次いで「設定」ウィンドウを再び開き、「**コード生成**」オプションを変更します。

3. 新たに作成された wsdl ディレクトリー内にある BFMWS.WSDL または HTMWS.WSDL ファイルを選択します。
4. 右クリックして、「**Web サービス**」 → 「**クライアントを生成**」を選択します。

   残りのステップを続行する前に、サーバーが開始していることを確認します。
5. 「Web サービス」ウィンドウで「**次へ**」をクリックして、デフォルトをすべて受け入れます。
6. 「Web サービス選択」ウィンドウで「**次へ**」をクリックして、デフォルトをすべて受け入れます。
7. 「クライアント環境構成」ウィンドウで、以下のようにします。
   a. 「**編集**」をクリックして、「Web サービス・ランタイム」オプションを IBM WebSphere に変更します。
   b. 「J2EE のバージョン」オプションを 1.4 に変更します。
   c. 「**OK**」をクリックします。
   d. 「**次へ**」をクリックします。
8. このステップは、ビジネス・プロセス Web サービス API とヒューマン・タスク Web サービス API の両方を含む Web サービス・クライアントを生成する必要がある場合にのみ必要です。なぜなら、両方の WSDL ファイルに重複するメソッドがあるからです。
   a. 「Web サービス・プロキシー」ウィンドウで、「名前空間からパッケージへのカスタム・マッピングを定義」を選択して、「**OK**」をクリックします。
   b. 「Web サービス・クライアントの名前空間からパッケージへのマッピング」ウィンドウで、以下の名前空間およびパッケージを追加します。

      BFMWS.wsdl の場合:

| 名前空間 | パッケージ |
|---|---|
| http://www.ibm.com/xmlns/prod/websphere/business-process/types/6.0 | com.ibm.sca.bpe |
| http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0 | com.ibm.sca.bpe |
| http://www.ibm.com/xmlns/prod/websphere/business-process/services/6.0/Binding | com.ibm.sca.bpe |
| http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/6.0 | com.ibm.sca.bpe |

      HTMWS.wsdl の場合:

| 名前空間 | パッケージ |
|---|---|
| http://www.ibm.com/xmlns/prod/websphere/human-task/types/6.0 | com.ibm.sca.task |
| http://www.ibm.com/xmlns/prod/websphere/human-task/services/6.0 | com.ibm.sca.task |
| http://www.ibm.com/xmlns/prod/websphere/human-task/services/6.0/Binding | com.ibm.sca.task |

| 名前空間 | パッケージ |
| --- | --- |
| http://www.ibm.com/xmlns/prod/websphere/bpc-common/types/6.0 | com.ibm.sca.task |

上書きを確認するように求められる場合、「**YesToAll**」をクリックします。

9. 「**終了**」をクリックします。

**タスクの結果**

複数のプロキシー、ロケーター、ヘルパー Java クラスで構成されるプロキシー・クライアントが生成され、プロジェクトに追加されます。デプロイメント記述子も更新されます。

**WSDL2Java によるプロキシー・クライアントの生成:**

WSDL2Java は、プロキシー・クライアントを生成するコマンド行ツールです。プロキシー・クライアントによって、クライアント・アプリケーションのプログラミングが容易になります。

**始める前に**

プロキシー・クライアントを生成するには、その前に、ビジネス・プロセスまたはヒューマン・タスクの Web サービス API を記述した WSDL ファイルを WebSphere 環境 (または WebSphere Process Server クライアント CD) からエクスポートし、それをクライアントのプログラミング環境にコピーしておく必要があります。

**このタスクについて**

**手順**

1. WSDL2Java ツールを使用してプロキシー・クライアントを生成します。
   タイプ:

   **wsdl2java** *options WSDLfilepath*

   各部の意味は、次のとおりです。

   - *オプション* は以下のとおりです。

     **-noWrappedOperations (-w)**
     ラップ操作の検出を使用不可にします。要求および応答メッセージ用の Java Bean が生成されます。

     **注:** これはデフォルト値ではありません。

     **-role (-r)**
     値 **client** を指定すると、クライアント・サイド開発用のファイルおよびバインディング・ファイルが生成されます。

     **-container (-c)**
     使用するクライアント・サイド・コンテナー。有効な引数は以下のとおりです。

**クライアント**
クライアント・コンテナー

**ejb** Enterprise JavaBeans (EJB) コンテナー

**none** コンテナーなし

**web** Web コンテナー

**-output (-o)**
生成されたファイルを保管するフォルダー。

WSDL2Java パラメーターの完全なリストが必要な場合は、**/help** コマンド行スイッチを使用するか、WID/RAD で WSDL2Java ツールのオンライン・ヘルプを参照してください。

- *WSDLfilepath* は、WebSphere 環境からエクスポートしたか、クライアント CD からコピーした WSDL ファイルのパスとファイル名です。

次の例では、ヒューマン・タスク・アクティビティー Web サービス API 用のプロキシー・クライアントが生成されます。

```
call wsdl2java.bat -r client -c client -noWrappedOperations
                   -output c:¥ws¥proxyClient  c:¥ws¥bin¥HTMWS.wsdl
```

2. 生成されたクラス・ファイルをプロジェクトに組み込みます。

## BPEL プロセス用ヘルパー・クラスの作成 (Java Web サービス)

具象 API 要求 (sendMessage、call など) で参照されるビジネス・オブジェクトでは、クライアント・アプリケーションが「document/literal wrapped」スタイル・エレメントを使用する必要があります。クライアント・アプリケーションは、ヘルパー・クラスに、必要なラッパー・エレメントの生成を担当させます。

### 始める前に

ヘルパー・クラスを作成するには、WebSphere Process Server 環境からあらかじめ Web サービス API の WSDL ファイルをエクスポートしておく必要があります。

### このタスクについて

Web サービス API の call() 命令や sendMessage() 命令を実行すると、WebSphere Process Server 上で、BPEL プロセスとの対話ができるようになります。call() 命令の入力メッセージは、プロセスの入力メッセージの document/literal ラッパーが提供されると予想します。

BPEL プロセスまたはヒューマン・タスクのヘルパー・クラスを生成する技法は、次に示すように数多く存在します。

1. SoapElement オブジェクトを使用する。

   WebSphere Integration Developer で使用可能な Rational Application Developer 環境では、Web サービス・エンジンが JAX-RPC 1.1 環境をサポートします。JAX-RPC 1.1 では、SoapElement オブジェクトが Document Object Model (DOM) エレメントを拡張するので、DOM API を使用して SOAP メッセージを作成、読み取り、ロード、および保管することが可能です。

例えば、WSDL ファイルに、次に示すワークフロー・プロセスまたはヒューマン・タスクの入力メッセージが含まれていると想定します。

```
<xsd:element name="operation1">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="input1" nillable="true" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

プロセスまたはヒューマン・タスク・モジュールを開発する際に、WSDL ファイルが作成されます。

DOM API を使用して、クライアント・アプリケーション内の対応する SOAP メッセージを作成するには、次のようにします。

```
SOAPFactory soapfactoryinstance = SOAPFactory.newInstance();
SOAPElement soapmessage = soapfactoryinstance.createElement
                          ("operation1", namespaceprefix, interfaceURI);
SOAPElement inputelement = soapfactoryinstance.createElement("input1");
inputelement.addTextNode( message value);
soapmessage.addChildElement(outputelement);
```

以下の例では、クライアント・アプリケーションで sendMessage 操作用の入力パラメーターを作成する方法を示します。

```
SendMessage inWsend = new SendMessage();
inWsend.setProcessTemplateName(processtemplatename);
inWsend.setPortType(porttype);
inWsend.setOperation(operationname);
inWsend.set_any(soapmessage);
```

2. WebSphere カスタム・データ・バインディング機能を使用する。

   この技法については、次の developerWorks の記事で説明されています。

   - How to choose a custom mapping technology for Web services
   - Developing Web Services with EMF SDOs for complex XML schema

   Interoperability With Patterns and Strategies for Document-Based Web Services

   Web Services support for Schema/WSDL(s) containing optional JAX-RPC 1.0/1.1 XML Schema Types

## クライアント・アプリケーションの作成 (Java Web サービス)

クライアント・アプリケーションは、Web サービス API に要求を送信し、Web サービス API からの応答を受信します。プロキシー・クライアントを使用して、複合データ型のフォーマット設定を行う通信クラスおよびヘルパー・クラスを管理すると、クライアント・アプリケーションから、Web サービス・メソッドをローカル機能のように呼び出すことができます。

### 始める前に

クライアント・アプリケーションの作成を開始する前に、プロキシー・クライアントと必要なヘルパー・クラスを生成します。

### このタスクについて

Web サービス対応の開発ツール (IBM Rational Application Developer (RAD) など) を使用すれば、クライアント・アプリケーションを開発できます。どのタイプの Web サービス・アプリケーションをビルドしても、Web サービス API を呼び出すことができます。

#### 手順

1. 新規クライアント・アプリケーション・プロジェクトを作成します。
2. プロキシー・クライアントを生成し、Java ヘルパー・クラスをプロジェクトに追加します。
3. クライアント・アプリケーションをコーディングします。
4. プロジェクトをビルドします。
5. クライアント・アプリケーションを実行します。

### 例

以下の例では、Business Flow Manager Web サービス API の使用方法を示します。

```
// create the proxy
       BFMIFProxy proxy = new BFMIFProxy();
 // prepare the input data for the operation
      GetProcessTemplate iW = new GetProcessTemplate();
      iW.setIdentifier(your_process_template_name);

 // invoke the operation
      GetProcessTemplateResponse oW = proxy.getProcessTemplate(iW);

 // process output of the operation
      ProcessTemplateType ptd = oW.getProcessTemplate();
      System.out.println("getName= " + ptd.getName());
      System.out.println("getPtid= " + ptd.getPtid());
```

## セキュリティーの追加 (Java Web サービス)

Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことによって保護する必要があります。

### このタスクについて

WebSphere Application Server は現在、次に示す Web サービス API のセキュリティー・メカニズムをサポートしています。

- ユーザー名トークン
- Lightweight Third Party Authentication (LTPA)

#### ユーザー名トークンのインプリメント:

ユーザー名トークンのセキュリティー・メカニズムでは、ユーザー名とパスワード資格情報を提供します。

#### このタスクについて

ユーザー名トークンのセキュリティー・メカニズムにより、さまざまなコールバック・ハンドラー を選択してインプリメントできます。選択に応じて、次の違いがあります。

- クライアント・アプリケーションを実行するときに、ユーザー名とパスワードの入力を毎回求められます。
- ユーザー名とパスワードがデプロイメント記述子に書き込まれます。

どちらの場合でも、入力されるユーザー名とパスワードは、対応するビジネス・プロセス・コンテナーまたはヒューマン・タスク・コンテナーの許可ロールの値と一致する必要があります。

ユーザー名とパスワードは要求メッセージ・エンベロープの中にカプセル化されるので、SOAP メッセージ・ヘッダーに「はっきり」示されます。そのため、クライアント・アプリケーションで HTTPS (HTTP over SSL) 通信プロトコルを使用する構成にするよう強くお勧めします。これにより、すべての通信が暗号化されます。HTTPS 通信プロトコルは、Web サービス API のエンドポイント URL アドレスを指定するときに選択できます。

ユーザー名トークンを定義するには、次のステップを実行します。

**手順**

1. セキュリティー・トークンを作成します。
   a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。
   b. 「**WS 拡張**」タブをクリックします。
   c. 「**サービス参照**」の下に、次の「Web サービス参照 (Web Service References)」がリストされます。
      - service/BFMWSService (ビジネス・プロセスの場合)
      - service/HTMWSService (ヒューマン・タスクの場合)

      どちらがリストされるかは、プロキシー・クライアントの生成時に、BFMWS.wsdl (ビジネス・プロセスの場合)、HTMWS.wsdl (ヒューマン・タスクの場合)、またはその両方のうちのどれが追加されたかによって決まります。
   d. どちらのサービス参照の場合も、以下のようにします。
      1) いずれかの「**サービス参照**」を選択します。
      2) 「**要求生成プログラム構成**」セクションを展開します。
      3) 「**セキュリティー・トークン**」サブセクションを展開します。
      4) 「**追加**」をクリックします。「セキュリティー・トークン」ウィンドウが開きます。
      5) 「**名前**」フィールドに、新規のセキュリティー・トークンの名前として **UserNameTokenBFM** または **UserNameTokenHTM** を入力します。
      6) 「**トークン・タイプ**」ドロップダウン・リストで、「**ユーザー名**」を選択します (「**ローカル名**」フィールドにはデフォルト値が自動的に入力されます)。
      7) 「**URI**」 フィールドはブランクのままにします。ユーザー名トークンには URI 値は不要です。
      8) 「**OK**」をクリックします。
2. 以下のように、トークン生成プログラムを作成します。

a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。
b. 「**WS バインディング**」タブをクリックします。
c. 「**サービス参照**」の下に、前のステップと同じ Web サービス参照がリストされます。
   - service/BFMWSService (ビジネス・プロセスの場合)
   - service/HTMWSService (ヒューマン・タスクの場合)
d. どちらのサービス参照の場合も、以下のようにします。
   1) いずれかの「**サービス参照**」を選択します。
   2) 「**セキュリティー要求生成プログラムのバインディング構成**」セクションを展開します。
   3) 「**トークン生成プログラム**」サブセクションを展開します。
   4) 「**追加**」をクリックします。「トークン生成プログラム」ウィンドウが開きます。
   5) 「**名前**」 フィールドに、新規のトークン生成プログラムの名前 (「UserNameTokenGeneratorBFM」または「UserNameTokenGeneratorHTM」など) を入力します。
   6) 「**トークン生成クラス**」フィールドで、トークン生成プログラム・クラス **com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator** が選択されていることを確認します。
   7) 「**セキュリティー・トークン**」ドロップダウン・リストで、前に作成した適切なセキュリティー・トークンを選択します。
   8) 「**値タイプの使用**」チェック・ボックスを選択します。
   9) 「**値のタイプ**」フィールドで、「**ユーザー名トークン (Username Token)**」を選択します (「**ローカル名**」フィールドは、「**ユーザー名トークン**」の選択を反映して自動的に入力されます)。
   10) 「**コールバック・ハンドラー**」フィールドで、「com.ibm.wsspi.wssecurity.auth.callback.GUIPromptCallbackHandler」(クライアント・アプリケーションを実行すると、ユーザー名およびパスワードを入力するようプロンプトが表示される)、または「com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler」と入力します。
   11) **NonPromptCallbackHandler** を選択した場合は、有効なユーザー名とパスワードを、デプロイメント記述子の対応するフィールドに指定する必要があります。
   12) 「**OK**」をクリックします。

**関連情報**

IBM WebSphere Developer Technical Journal: Web services security with WebSphere Application Server V6

**LTPA セキュリティー・メカニズムのインプリメント:**

Lightweight Third Party Authentication (LTPA) セキュリティー・メカニズムは、クライアント・アプリケーションが以前に確立されたセキュリティー・コンテキストで実行されている場合に使用できます。

### このタスクについて

LTPA セキュリティー・メカニズムは、セキュリティー・コンテキストが既に確立されている保護された環境でクライアント・アプリケーションが実行されている場合にのみ使用できます。例えば、クライアント・アプリケーションが Enterprise JavaBeans(EJB) コンテナーで実行されている場合、EJB クライアントは、クライアント・アプリケーションを呼び出すためにあらかじめログインしておく必要があります。セキュリティー・コンテキストはそれから確立されます。その後 EJB クライアント・アプリケーションが Web サービスを呼び出すと、LTPA コールバック・ハンドラーは、セキュリティー・コンテキストから LTPA トークンを取得して SOAP 要求メッセージに追加します。サーバー・サイドでは、LTPA トークンが LTPA メカニズムによって処理されます。

LTPA セキュリティー・メカニズムをインプリメントするには、次のステップを実行します。

### 手順

1. WebSphere Integration Developer で使用可能な Rational Application Developer 環境で、「**WS バインディング**」 → 「**セキュリティー要求生成プログラムのバインディング構成**」 → 「**トークン生成プログラム**」を選択します。
2. セキュリティー・トークンを作成します。
   a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。
   b. 「**WS 拡張**」タブをクリックします。
   c. 「**サービス参照**」の下に、次の「**Web サービス参照 (Web Service References)**」がリストされます。
      - service/BFMWSService (ビジネス・プロセスの場合)
      - service/HTMWSService (ヒューマン・タスクの場合)

      どちらがリストされるかは、プロキシー・クライアントの生成時に、BFMWS.wsdl (ビジネス・プロセスの場合)、HTMWS.wsdl (ヒューマン・タスクの場合)、またはその両方のうちのどれが追加されたかによって決まります。
   d. どちらのサービス参照の場合も、以下のようにします。
      1) いずれかの「**サービス参照**」を選択します。
      2) 「**要求生成プログラム構成**」セクションを展開します。
      3) 「**セキュリティー・トークン**」サブセクションを展開します。
      4) 「**追加**」をクリックします。「セキュリティー・トークン」ウィンドウが開きます。
      5) 「**名前**」フィールドに、新規のセキュリティー・トークンの名前として **LTPATokenBFM** または **LTPATokenHTM** を入力します。

6) 「**トークン・タイプ**」ドロップダウン・リストで、「**LTPAToken**」を選択します (「**URI**」および「**ローカル名**」フィールドには、自動的にデフォルト値が入力されます)。

7) 「**OK**」をクリックします。

3. 以下のように、トークン生成プログラムを作成します。

a. モジュールの「**デプロイメント・エディター (Deployment Editor)**」を開きます。

b. 「**WS バインディング**」タブをクリックします。

c. 「**サービス参照**」の下に、前のステップと同じ Web サービス参照がリストされます。

- service/BFMWSService (ビジネス・プロセスの場合)
- service/HTMWSService (ヒューマン・タスクの場合)

d. どちらのサービス参照の場合も、以下のようにします。

1) いずれかの「**サービス参照**」を選択します。

2) 「**セキュリティー要求生成プログラムのバインディング構成**」セクションを展開します。

3) 「**トークン生成プログラム**」サブセクションを展開します。

4) 「**追加**」をクリックします。「トークン生成プログラム」ウィンドウが開きます。

5) 「**名前**」 フィールドに、新規のトークン生成プログラムの名前 (「LTPATokenGeneratorBFM」または「LTPATokenGeneratorHTM」など) を入力します。

6) 「**トークン生成クラス**」フィールドで、トークン生成プログラム・クラス **com.ibm.wsspi.wssecurity.token.LTPATokenGenerator** が選択されていることを確認します。

7) 「**セキュリティー・トークン**」ドロップダウン・リストで、前に作成した適切なセキュリティー・トークンを選択します。

8) 「**値タイプの使用**」チェック・ボックスを選択します。

9) 「**値のタイプ**」フィールドで、「**LTPAToken**」を選択します (「**URI**」および「**ローカル名**」フィールドは、「**LTPA トークン (LTPA Token)**」の選択を反映して自動的に入力されます)。

10) 「**コールバック・ハンドラー**」フィールドに、「com.ibm.wsspi.wssecurity.auth.callback.LTPATokenCallbackHandler」と入力します。

11) 「**OK**」をクリックします。

**タスクの結果**

実行時に **LTPATokenCallbackHandler** は、既存のセキュリティー・コンテキストから LTPA トークンを取得して SOAP 要求メッセージに追加します。

## トランザクション・サポートの追加 (Java Web サービス)

Java Web サービス・クライアント・アプリケーションを構成して、クライアント・アプリケーションのコンテキストをサービス要求の一部として引き渡すことによってクライアントのトランザクションにサーバー・サイド要求の処理が参加するのを

許可することができます。このアトミック・トランザクション・サポートは、Web Services-Atomic Transaction (WS-AT) 仕様で定義されています。

### このタスクについて

WebSphere Application Server は、各 Web サービスの API 要求を別々のアトミック・トランザクションとして実行します。クライアント・アプリケーションは、次のいずれかの方法でトランザクション・サポートを使用するように構成できます。

- トランザクションに参加する。サーバー・サイド要求の処理は、クライアント・アプリケーションのトランザクション・コンテキスト内で実行されます。この場合、Web サービス API 要求の実行中やロールバック中にサーバーに問題が発生すると、クライアント・アプリケーションの要求もロールバックされます。
- トランザクション・サポートを使用しない。WebSphere Application Server は、要求を実行するための新規トランザクションを引き続き作成しますが、サーバー・サイド要求の処理は、クライアント・アプリケーションのトランザクション・コンテキストでは実行されません。

## .NET 環境でのクライアント・アプリケーションの開発

Microsoft .NET は、Web サービスを使用してアプリケーションを接続する強力な開発環境を提供します。

### プロキシー・クライアントの生成 (.NET)

.NET クライアント・アプリケーションは、*プロキシー・クライアント* を使用して Web サービス API と対話します。プロキシー・クライアントのおかげで、クライアント・アプリケーションで、複雑な Web サービス・メッセージング・プロトコルを意識する必要がなくなります。

### 始める前に

プロキシー・クライアントを作成するには、まず WebSphere 環境からいくつかの WSDL ファイルをエクスポートして、それをクライアントのプログラミング環境にコピーする必要があります。

**注:** WebSphere Process Server クライアント CD を所有している場合は、代わりにその CD からファイルをコピーできます。

### このタスクについて

プロキシー・クライアントは、C# Bean クラスのセットで構成されています。各クラスには、単一の Web サービスで公開されるメソッドおよびオブジェクトがすべて含まれています。サービス・メソッドは、パラメーターを完全な SOAP メッセージにアセンブルし、その SOAP メッセージを HTTP 経由で Web サービスに送信し、Web サービスから応答を受信して、戻されたデータを処理します。

**注:** プロキシー・クライアントの生成が必要なのは一度だけです。 Web サービス API にアクセスするクライアント・アプリケーションはすべて、以後は同じプロキシー・クライアントを使用できます。

**手順**

1. プロキシー・クライアントの生成には WSDL コマンドを使用します。 タイプ:

**wsdl** *options WSDLfilepath*

各部の意味は、次のとおりです。

- オプション は以下のとおりです。

  **/language**
  プロキシー・クラスの作成に使用する言語を指定できます。デフォルトは C# です。言語引数として、**VB** (Visual Basic)、**JS** (JScript)、または **VJS** (Visual J#) を指定することもできます。

  **/output**
  該当するサフィックスの付いた出力ファイルの名前。例えば、proxy.cs です。

  **/protocol**
  プロキシー・クラスでインプリメントされるプロトコル。**SOAP** がデフォルトの設定です。

  **WSDL.exe** パラメーターの完全なリストが必要な場合は、**/?** コマンド行スイッチを使用するか、Visual Studio で WSDL ツールのオンライン・ヘルプを参照してください。

- *WSDLfilepath* は、WebSphere 環境からエクスポートしたか、クライアント CD からコピーした WSDL ファイルのパスとファイル名です。

次の例では、Human Task Manager Web サービス API 用のプロキシー・クライアントが生成されます。

```
wsdl /language:cs /output:proxyclient.cs c:¥ws¥bin¥HTMWS.wsdl
```

2. プロキシー・クライアントをダイナミック・リンク・ライブラリー (DLL) ファイルとしてコンパイルします。

## BPEL プロセス用ヘルパー・クラスの作成 (.NET)

特定の Web サービス API 操作では、クライアント・アプリケーションが「document/literal」スタイルのラップ・エレメントを使用する必要があります。クライアント・アプリケーションは、ヘルパー・クラスに、必要なラッパー・エレメントの生成を担当させます。

### 始める前に

ヘルパー・クラスを作成するには、WebSphere Process Server 環境からあらかじめ Web サービス API の WSDL ファイルをエクスポートしておく必要があります。

### このタスクについて

Web サービス API の call() 命令と sendMessage() 命令により、BPEL プロセスが WebSphere Process Server 内で起動します。call() 命令の入力メッセージは、BPEL プロセスの入力メッセージの document/literal ラッパーが提供されると予想します。BPEL プロセスの必要な Bean とクラスを生成するには、`<wsdl:types>` エレメントを新規の XSD ファイルにコピーし、xsd.exe ツールを使用してヘルパー・クラスを生成します。

### 手順

1. まだ実行していない場合は、WebSphere Integration Developer から BPEL プロセス・インターフェースの WSDL ファイルをエクスポートします。
2. テキスト・エディターまたは XML エディターで WSDL ファイルを開きます。
3. <wsdl:types> エレメントのすべての子エレメントの内容をコピーし、新しいスケルトン XSD ファイルに貼り付けます。
4. XSD ファイル上で xsd.exe ツールを実行します。

   **call xsd.exe file.xsd /classes /o**

   各部の意味は、次のとおりです。

   **file.xsd** 変換する XML スキーマ定義ファイル。

   **/classes (/c)**
   指定した XSD ファイル (複数も可) の内容に対応するヘルパー・クラスを生成します。

   **/output (/o)**
   生成したファイルの出力ディレクトリーを指定します。このディレクトリーを省略すると、デフォルトでは現行ディレクトリーになります。

   以下に例を示します。

   **call xsd.exe ProcessCustomer.xsd /classes /output:c:¥temp**
5. 生成されるクラス・ファイルをクライアント・アプリケーションに追加します。例えば Visual Studio を使用する場合は、「**プロジェクト**」 → 「**既存項目の追加**」メニュー・オプションを実行します。

## 例

ProcessCustomer.wsdl ファイルの内容が以下のような場合:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:bons1="http://com/ibm/bpe/unittest/sca"
                  xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  name="ProcessCustomer"
                  targetNamespace="http://ProcessTypes/bpel/ProcessCustomer">
  <wsdl:types>
    <xsd:schema targetNamespace="http://ProcessTypes/bpel/ProcessCustomer"
                xmlns:bons1="http://com/ibm/bpe/unittest/sca"
                xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:import namespace="http://com/ibm/bpe/unittest/sca"
                schemaLocation="xsd-includes/http.com.ibm.bpe.unittest.sca.xsd"/>
      <xsd:element name="doit">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="input1" nillable="true" type="bons1:Customer"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="doitResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="output1" nillable="true" type="bons1:Customer"/>
          </xsd:sequence>
        </xsd:complexType>
```

```
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
    <wsdl:message name="doitRequestMsg">
    <wsdl:part element="tns:doit" name="doitParameters"/>
  </wsdl:message>
    <wsdl:message name="doitResponseMsg">
    <wsdl:part element="tns:doitResponse" name="doitResult"/>
  </wsdl:message>
    <wsdl:portType name="ProcessCustomer">
    <wsdl:operation name="doit">
      <wsdl:input message="tns:doitRequestMsg" name="doitRequest"/>
      <wsdl:output message="tns:doitResponseMsg" name="doitResponse"/>
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

結果として出力される XSD ファイルは以下のようになります。

```
<xsd:schema xmlns:bons1="http://com/ibm/bpe/unittest/sca"
                      xmlns:tns="http://ProcessTypes/bpel/ProcessCustomer"
                      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                      targetNamespace="http://ProcessTypes/bpel/ProcessCustomer">
 <xsd:import namespace="http://com/ibm/bpe/unittest/sca"
              schemaLocation="Customer.xsd"/>
 <xsd:element name="doit">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="input1" type="bons1:Customer" nillable="true"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:element name="doitResponse">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="output1" type="bons1:Customer" nillable="true"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

**関連情報**

XML Schema Definition Tool (XSD.EXE) に関する Microsoft の資料

## クライアント・アプリケーションの作成 (.NET)

クライアント・アプリケーションは、Web サービス API に要求を送信し、Web サービス API からの応答を受信します。プロキシー・クライアントを使用して、複合データ型のフォーマット設定を行う通信クラスおよびヘルパー・クラスを管理すると、クライアント・アプリケーションから、Web サービス・メソッドをローカル機能のように呼び出すことができます。

### 始める前に

クライアント・アプリケーションの作成を開始する前に、プロキシー・クライアントと必要なヘルパー・クラスを生成します。

### このタスクについて

.NET 対応の開発ツール (Visual Studio .NET など) を使用すれば、.NET クライアント・アプリケーションを開発できます。どのタイプの .NET アプリケーションを

ビルドしても、汎用の Web サービス API を呼び出すことができます。

**手順**

1. 新規クライアント・アプリケーション・プロジェクトを作成します。 例えば、Visual Studio で **WinFX Windows® Application** を作成します。
2. プロジェクト・オプションで、プロキシー・クライアントのダイナミック・リンク・ライブラリー (DLL) ファイルに参照を追加します。 ビジネス・オブジェクト定義を含むヘルパー・クラスすべてをプロジェクトに追加します。例えば Visual Studio では、「**プロジェクト**」 → 「**既存項目の追加**」オプションを実行します。
3. プロキシー・クライアント・オブジェクトを作成します。 例:

```
HTMClient.HTMReference.HumanTaskManagerComponent1Export_HumanTaskManagerHttpService service =
    new HTMClient.HTMReference.HumanTaskManagerComponent1Export_HumanTaskManagerHttpService();
```

4. メッセージで使用され、Web サービスとの間で送受信されるビジネス・オブジェクトのデータ型を宣言します。 例:

```
HTMClient.HTMReference.TKIID id = new HTMClient.HTMReference.TKIID();

ClipBG bg = new ClipBG();
Clip clip = new Clip();
```

5. 特定の Web サービス関数を呼び出し、必要なパラメーターを指定します。 例えば、ヒューマン・タスクを作成して開始するには、以下のようにします。

```
HTMClient.HTMReference.createAndStartTask task =
   new HTMClient.HTMReference.createAndStartTask();
HTMClient.HTMReference.StartTask sTask = new HTMClient.HTMReference.StartTask();

sTask.taskName = "SimpleTask";
sTask.taskNamespace = "http://myProcess/com/acme/task";
sTask.inputMessage = bg;
task.inputTask = sTask;

id = service.createAndStartTask(task).outputTask;
```

6. リモートのプロセスおよびタスクは、永続 ID (上記の例では `id`) で識別されます。 例えば、上記で作成したヒューマン・タスクを要求するには、以下のようにします。

```
HTMClient.HTMReference.claimTask claim = new HTMClient.HTMReference.claimTask();
claim.inputTask = id;
```

## セキュリティーの追加 (.NET)

Web サービス通信は、クライアント・アプリケーションにセキュリティー・メカニズムを組み込むことにより保護できます。

### このタスクについて

これらのセキュリティー・メカニズムには、ユーザー名トークン (ユーザー名とパスワード)、またはカスタム・バイナリーと XML ベースのセキュリティー・トークンなどがあります。

**手順**

1. Web Services Enhancements (WSE) 2.0 SP3 for Microsoft .NET をダウンロードしてインストールします。 入手先は以下のとおりです。

http://www.microsoft.com/downloads/details.aspx?familyid=1ba1f631-c3e7-420a-bc1e-ef18bab66122&displaylang=en

2. 生成されたプロキシー・クライアント・コードを以下のように変更します。

   変更前:

```
public class Export1_MyMicroflowHttpService : System.Web.Services.Protocols.SoapHttpClientProtocol {
```

   変更後:

```
public class Export1_MyMicroflowHttpService : Microsoft.Web.Services2.WebServicesClientProtocol {
```

   **注:** これらの変更は、WSDL.exe ツールを実行してプロキシー・クライアントを再生成すると失われます。

3. ファイルの先頭に以下の行を追加して、クライアント・アプリケーション・コードを変更します。

```
using System.Web.Services.Protocols;
using Microsoft.Web.Services2;
using Microsoft.Web.Services2.Security.Tokens;
...
```

4. 必要なセキュリティー・メカニズムをインプリメントするコードを追加します。例えば、ユーザー名とパスワード保護を追加するコードは次のとおりです。

```
string user = "U1";
string pwd = "password";
UsernameToken token =
   new UsernameToken(user, pwd, PasswordOption.SendPlainText);

me._proxy.RequestSoapContext.Security.Tokens.Clear();
me._proxy.RequestSoapContext.Security.Tokens.Add(token);
```

## ビジネス・プロセスおよびタスク関連のオブジェクトの照会

Web サービス API を使用すると、Business Process Choreographer データベース内のビジネス・プロセス・オブジェクトおよびタスク関連オブジェクトを照会して、これらのオブジェクトの特定のプロパティーを取得することができます。

### このタスクについて

Business Process Choreographer データベースは、ビジネス・プロセスおよびタスクの管理用のテンプレート (モデル) とインスタンス (ランタイム) のデータを保管します。

クライアント・アプリケーションは、Web サービス API 経由で照会を発行し、データベースからビジネス・プロセスおよびビジネス・タスクに関する情報を取得することができます。

クライアント・アプリケーションは、1 回限りの照会を発行して、オブジェクトの特定のプロパティーを取得することができます。使用頻度の高い照会は、保管しておくことができます。クライアント・アプリケーションは、このような保管照会文を後で取り出して使用することができます。

## Web サービス API による、ビジネス・プロセスおよびタスク関連オブジェクトに対する照会

Web サービス API の QUERY インターフェースを使用して、ビジネス・プロセスおよびタスクに関する情報を取得します。

クライアント・アプリケーションは、SQL 形式の構文を使用して、データベースを照会します。

### Java Web サービスの例

```
string processTemplateName = "ProcessCustomerLR";
query query1 = new query();
query1.selectClause = "DISTINCT PROCESS_INSTANCE.STARTED, PROCESS_INSTANCE.PIID";
query1.whereClause =
       "PROCESS_INSTANCE.TEMPLATE_NAME = '" + processTemplateName + "'";
query1.orderByClause = "PROCESS_INSTANCE.STARTED";
query1.threshold = null;
query1.timeZone = "UTC"; query1.skipTuples = null;
queryResponse queryResponse1 = proxy.query(query1);
```

データベースから取り出した情報は、Web サービス API を使用して*照会結果セット*として戻されます。

以下に例を挙げます。

```
QueryResultSetType queryResultSet = queryResponse1.queryResultSet;
if (queryResultSet != null) {
   Console.WriteLine("--> QueryResultSetType");
   Console.WriteLine(" . size= " + queryResultSet.size);
   Console.WriteLine(" . numberColumns= " + queryResultSet.numberColumns);
   string indent = " . ";

   // -- the query column info
   QueryColumnInfoType[] queryColumnInfo = queryResultSet.QueryColumnInfo;
   if (queryColumnInfo.Length > 0) {
      Console.WriteLine();
      Console.WriteLine("= . QueryColumnInfoType size= " + queryColumnInfo.Length);
      Console.Write( " | tableName ");
      for (int i = 0; i < queryColumnInfo.Length ; i++) {
         Console.Write( " | " + queryColumnInfo[i].tableName.PadLeft(20) );
      }
      Console.WriteLine();
      Console.Write( " | columnName ");
      for (int i = 0; i < queryColumnInfo.Length ; i++) {
         Console.Write( " | " + queryColumnInfo[i].columnName.PadLeft(20) );
      }
      Console.WriteLine();
      Console.Write( " | data type ");
      for (int i = 0; i < queryColumnInfo.Length ; i++) {
         QueryColumnInfoTypeType tt = queryColumnInfo[i].type;
         Console.WriteLine( " | " + tt.ToString());
      }
         Console.WriteLine();
   }
   else {
      Console.WriteLine("--> queryColumnInfo= <null>");
   }

   // - the query result values
   string[][] result = queryResultSet.result;
   if (result !=null) {
      Console.WriteLine();
      Console.WriteLine("= . result size= " + result.Length);
      for (int i = 0; i &lt; result.Length; i++) {
```

```
        Console.Write(indent +i );
        string[] row = result[i];
        for (int j = 0; j &lt; row.Length; j++ ) {
           Console.Write(" | " + row[j]);
        }
        Console.WriteLine();
      }
   }
   else {
      Console.WriteLine("--> result= <null>");
   }
}
else {
   Console.WriteLine("--> QueryResultSetType= <null>");
}
```

照会関数は、呼び出し元の権限に応じてオブジェクトを戻します。照会結果セットには、呼び出し元が表示する許可を持つオブジェクトのプロパティーが含まれるのみです。

オブジェクトのプロパティーを照会するために、事前定義データベース・ビューが提供されています。プロセス・テンプレートの場合、照会関数には以下の構文があります。

```
ProcessTemplateData[] queryProcessTemplates
                      (java.lang.String whereClause,
                       java.lang.String orderByClause,
                       java.lang.Integer threshold,
                       java.util.TimeZone timezone);
```

タスク・テンプレートの場合、照会関数には以下の構文があります。

```
TaskTemplate[] queryTaskTemplates
               (java.lang.String whereClause,
                java.lang.String orderByClause,
                java.lang.Integer threshold,
                java.util.TimeZone timezone);
```

他のビジネス・プロセスおよびタスク関連オブジェクトの場合、照会関数には以下の構文があります。

```
QueryResultSet query (java.lang.String selectClause,
                      java.lang.String whereClause,
                      java.lang.String orderByClause,
                      java.lang.Integer skipTuples
                      java.lang.Integer threshold,
                      java.util.TimeZone timezone);
```

QUERY インターフェースには、queryAll メソッドも含まれています。このメソッドを使用して、オブジェクトに関係のあるデータすべてを、モニターなどの目的で取得することができます。queryAll メソッドの呼び出し元には、Java 2 Platform Enterprise Edition (J2EE) ロールの、BPESystemAdministrator、BPESystemMonitor、TaskSystemAdministrator、または TaskSystemMonitor のいずれかが必要です。オブジェクトの対応する作業項目を使用した許可検査は適用されません。

**.NET の例**

```
ProcessTemplateType[] templates = null;

   try {
```

```
  queryProcessTemplates iW = new queryProcessTemplates();
  iW.whereClause = "PROCESS_TEMPLATE.STATE=PROCESS_TEMPLATE.STATE.STATE_STARTED";
  iW.orderByClause = null;
  iW.threshold = null;
  iW.timeZone = null;

  Console.WriteLine("--> queryProcessTemplates ... ");
  Console.WriteLine("--> query: WHERE " + iW.whereClause + " ORDER BY " +
    iW.orderByClause + " THRESHOLD " + iW.threshold + " TIMEZONE" + iW.timeZone);

  templates = proxy.queryProcessTemplates(iW);

  if (templates.Length < 1) {
    Console.WriteLine("--> No templates found :-(");
  }
  else {
    for (int i = 0; i < templates.Length ; i++) {
      Console.Write("--> found template with ptid: " + templates[i].ptid);
      Console.WriteLine(" and name: " + templates[i].name);
     /*  ... other properties of ProcessTemplateType ... */
    }
  }
}
catch (Exception e) {
  Console.WriteLine("exception= " + e);
}
```

## 保管照会文の管理

保管照会文は、頻繁に実行される照会を保管するための方法です。保管照会文は、すべてのユーザーが使用可能な照会 (共通照会) か、特定のユーザーに属する照会 (専用照会) のいずれかです。

### このタスクについて

保管照会文は、データベースに保管され、名前で識別される照会のことです。専用の保管照会文と共通の保管照会文の名前を同じにすることができます。異なる複数の所有者の専用保管照会文を同じ名前にすることもできます。

保管照会文は、ビジネス・プロセス・オブジェクト、タスク・オブジェクト、またはこの 2 つのオブジェクト・タイプの組み合わせたものを対象とします。

共通保管照会文の管理
: 共通保管照会文がシステム管理者によって作成されます。 この照会は、全ユーザーが使用できます。

他のユーザーの専用保管照会文の管理
: 専用照会はどのユーザーでも作成できます。この照会は、 照会の所有者とシステム管理者しか使用できません。

専用保管照会文の操作
: システム管理者でなくても、自分専用の保管照会文は作成、実行、および削除できます。また、システム管理者が作成した共通の保管照会文を使用することもできます。

# Business Process Choreographer JMS API を使用したクライアント・アプリケーションの開発

Java Messaging Service (JMS) API を介してビジネス・プロセス・アプリケーションに非同期でアクセスするクライアント・アプリケーションを開発できます。

## このタスクについて

JMS クライアント・アプリケーションは要求および応答メッセージを JMS API と交換します。要求メッセージを作成するために、クライアント・アプリケーションは JMS TextMessage メッセージ本文に、対応する操作の文書/リテラル・ラッパーを表す XML エレメントを入力します。

**関連概念**

187 ページの『ビジネス・プロセスおよびヒューマン・タスクと対話するためのプログラミング・インターフェースの比較』
ビジネス・プロセスおよびヒューマン・タスクと対話するクライアント・アプリケーションの作成には、Enterprise JavaBeans (EJB)、Web サービス、Java Message Service (JMS) および Representational State Transfer (REST) サービスなどの汎用プログラミング・インターフェースを使用できます。これらのインターフェースには、それぞれ異なる特性があります。

## ビジネス・プロセスの要件

Business Process Choreographer 上で実行するように WebSphere Integration Developer で開発されたビジネス・プロセスが、JMS API によってアクセス可能となるためには、特定の規則に準拠する必要があります。

要件は以下のとおりです。

1. ビジネス・プロセスのインターフェースは、Java API for XML-based RPC (JAX-RPC 1.1) 仕様で定義された「document/literal wrapped」スタイルを使用して定義する必要があります。これは、WebSphere Integration Developer で開発するすべてのビジネス・プロセスとヒューマン・タスクのデフォルト・スタイルです。
2. Web サービス操作のビジネス・プロセスとヒューマン・タスクによって公開される障害メッセージは、XML スキーマ・エレメントにより定義される単一の WSDL メッセージ部を構成する必要があります。以下に例を挙げます。

```
<wsdl:part name="myFault" element="myNamespace:myFaultElement"/>
```

**関連情報**

Java API for XML based RPC (JAX-RPC) のダウンロード・ページ

Which style of WSDL should I use?

## JMS レンダリングの許可

JMS インターフェースの使用を許可するには、WebSphere Application Server でセキュリティー設定が使用可能になっている必要があります。

ビジネス・プロセス・コンテナーがインストールされている場合、ロール **JMSAPIUser** をユーザー ID にマップする必要があります。このユーザー ID を使用して、すべての JMS API 要求を発行します。例えば、**JMSAPIUser** が「User A」にマップされる場合、すべての JMS API 要求はプロセス・エンジンにとって「User A」から発生しているように見えます。

**JMSAPIUser** ロールには以下の権限を割り当てる必要があります。

| 要求 | 必要な許可 |
|---|---|
| forceTerminate | プロセス管理者 |
| sendEvent | 可能なアクティビティー所有者またはプロセス管理者 |

**注:** その他のすべての要求の場合、特殊権限は必要ありません。

特殊権限は、ビジネス・プロセス管理者のロールを持つ人物に付与されます。ビジネス・プロセス管理者は特殊なロールです。これは、プロセス・インスタンスのプロセス管理者とは異なります。ビジネス・プロセス管理者はすべての特権を持っています。

プロセス・スターターのユーザー ID は、プロセス・インスタンスが存在している場合、ユーザー・レジストリーから削除できません。このユーザー ID を削除する場合、このプロセスのナビゲーションが継続できません。 システム・ログ・ファイルに、次のような例外が書き込まれます。

```
no unique ID for: <user ID>
```

## JMS インターフェースへのアクセス

JMS インターフェースを介してメッセージを送受信するには、まずアプリケーションで BPC.cellname.Bus への接続を作成し、セッションを作成し、メッセージ・プロデューサーおよびコンシューマーを生成する必要があります。

### このタスクについて

プロセス・サーバーは、point-to-point パラダイムに従う Java Message Service (JMS) メッセージを受け入れます。 JMS メッセージを送受信するアプリケーションは、以下のアクションに従う必要があります。

以下の例では、JMS クライアントは管理対象環境 (EJB、アプリケーション・クライアント、または Web クライアント・コンテナー) で実行していると想定しています。 JMS クライアントを J2SE 環境で実行する場合は、http://www-1.ibm.com/support/docview.wss?uid=swg24012804 の「IBM Client for JMS on J2SE with IBM WebSphere Application Server」を参照してください。

**手順**

1. BPC.*cellname*.Bus への接続を作成します。 クライアント・アプリケーションの要求用の事前構成された接続ファクトリーは存在しません。つまり、クライアント・アプリケーションは、JMS API の ReplyConnectionFactory を使用するか、または固有の接続ファクトリーを作成するかのいずれかが可能です。作成する場合は接続ファクトリーを検索するために、Java Naming and Directory Interface

(JNDI) 参照を使用できます。 JNDI 参照名は、Business Process Choreographer の外部要求キューの構成時に指定したものと同じ名前である必要があります。以下の例では、クライアント・アプリケーションが「jms/clientCF」という固有の接続ファクトリーを作成すると想定しています。

```
//Obtain the default initial JNDI context.
Context initialContext = new InitialContext();

// Look up the connection factory.
// Create a connection factory that connects to the BPC bus.
// Call it, for example, "jms/clientCF".
// Also configure an appropriate authentication alias.
ConnectionFactory connectionFactory =
            (ConnectionFactory)initialcontext.lookup("jms/clientCF");

// Create the connection.
Connection connection = connectionFactory.createConnection();
```

2. セッションを作成し、メッセージ・プロデューサーおよびコンシューマーが作成されるようにします。

```
// Create a transaction session using auto-acknowledgement.
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

3. メッセージを送信するメッセージ・プロデューサーを作成します。 JNDI 参照名は、Business Process Choreographer の外部要求キューの構成時に指定したものと同じ名前である必要があります。

```
// Look up the destination of the Business Process Choreographer input queue to
// send messages to.
Queue sendQueue = (Queue) initialcontext.lookup("jms/BFMJMSAPIQueue");

// Create a message producer.
MessageProducer producer = session.createProducer(sendQueue);
```

4. 応答を受信するメッセージ・コンシューマーを作成します。 応答宛先の JNDI 参照名は、ユーザー定義の宛先を指定できますが、デフォルトの (Business Process Choreographer 定義の) 応答宛先 jms/BFMJMSReplyQueue も指定できます。どちらの場合も、応答宛先は BPC.<cellname>.Bus にしておく必要があります。

```
// Look up the destination of the reply queue.
Queue  replyQueue = (Queue) initialcontext.lookup("jms/BFMJMSReplyQueue");

// Create a message consumer.
MessageConsumer consumer = session.createConsumer(replyQueue);
```

5. メッセージを送信します。

```
// Start the connection.
connection.start();

// Create a message - see the task descriptions for examples - and send it.
// This method is defined elsewhere ...
String payload = createXMLDocumentForRequest();
TextMessage requestMessage = session.createTextMessage(payload);

// Set mandatory JMS header.
// targetFunctionName is the operation name of JMS API
// (for example, getProcessTemplate, sendMessage)
requestMessage.setStringProperty("TargetFunctionName", targetFunctionName);

// Set the reply queue; this is mandatory if the replyQueue
// is not the default queue (as it is in this example).
requestMessage.setJMSReplyTo(replyQueue);

// Send the message.
```

```
producer.send(requestMessage);

// Get the message ID.
String jmsMessageID = requestMessage.getJMSMessageID();

session.commit();
```

6. 返信を受信します。

```
// Receive the reply message and analyse the reply.
TextMessage  replyMessage = (TextMessage) consumer.receive();

// Get the payload.
String payload = replyMessage.getText();

session.commit();
```

7. 接続を閉じ、リソースを解放します。

```
 // Final housekeeping; free the resources.
session.close();
connection.close();
```

**注:** 各トランザクションの後に接続を閉じることは不要です。接続が開始したら、接続が閉じるまでに、要求および応答メッセージはいくつでも交換できます。例では、単一のビジネス・メソッド内に単一の呼び出しがある単純なケースを示しています。

## Business Process Choreographer JMS メッセージの構造

各 JMS メッセージのヘッダーおよび本文には事前定義構造がある必要があります。

Java Message Service (JMS) メッセージは以下のもので構成されます。

- メッセージ識別およびルーティング情報用のメッセージ・ヘッダー。
- 目次を保持するメッセージの本文 (ペイロード)。

Business Process Choreographer はテキスト・メッセージ形式のみサポートします。

### メッセージ・ヘッダー

JMS は、クライアントが多くのメッセージ・ヘッダー・フィールドにアクセスできるようにします。

Business Process Choreographer JMS クライアントは以下のヘッダー・フィールドを設定できます。

- **JMSReplyTo**

  要求に対する応答を送信する宛先。このフィールドが要求メッセージで指定されていない場合、応答は Export インターフェースのデフォルトの応答宛先に送信されます (Export は、ビジネス・プロセス・コンポーネントのクライアント・インターフェース・レンダリングです)。この宛先は、`initialContext.lookup ("jms/BFMJMSReplyQueue");` を使用して取得できます。
- **TargetFunctionName**

  WSDL 操作の名前 (例えば、"queryProcessTemplates")。このフィールドは常に設定する必要があります。 TargetFunctionName は、ここで説明されている汎用の JMS メッセージ・インターフェースの操作を指定することに注意してください。

これを、例えば **call** または **sendMessage** 操作を使用して、間接的に呼び出すことができる具体的なプロセスまたはタスクによって提供される操作と混同しないでください。

また、Business Process Choreographer クライアントは以下のヘッダー・フィールドにアクセスできます。

- **JMSMessageID**

  メッセージを一意的に識別します。メッセージが送信されると、JMS プロバイダーによって設定されます。メッセージの送信前にクライアントが JMSMessageID を設定する場合、JMS プロバイダーによって上書きされます。メッセージの ID が認証目的で必要とされる場合、クライアントはメッセージの送信後に JMSMessageID を取得できます。
- **JMSCorrelationID**

  メッセージをリンクします。このフィールドは設定しないでください。 Business Process Choreographer 応答メッセージには、要求メッセージの JMSMessageID が含まれています。

各応答メッセージには以下の JMS ヘッダー・フィールドが含まれています。

- **IsBusinessException**

  WSDL 出力メッセージの場合は ″false″ で、WSDL 障害メッセージの場合は ″true″ です。

ServiceRuntimeExceptions は非同期クライアント・アプリケーションに戻されません。 JMS 要求メッセージの処理中に重大な例外が発生すると、それはランタイム障害となり、この要求メッセージを処理しているトランザクションはロールバックします。その後、JMS 要求メッセージが再度送達されます。メッセージを SCA Export の一部として処理している間 (例えば、メッセージの非直列化中) に障害が早期に発生する場合、SCA Export の受信宛先によって指定された失敗した送達の最大数まで再試行されます。送達の失敗最大数に達した後、要求メッセージは Business Process Choreographer バスのシステム例外宛先に追加されます。しかし、Business Flow Manager の SCA コンポーネントによる要求を実際に処理している間に障害が発生する場合、失敗した要求メッセージは WebSphere Process Server の失敗したイベント管理インフラストラクチャーによって処理されます。つまり、再試行しても例外状態が解決しない場合は、最終的に、失敗したイベント管理データベースに入れられることがあります。

### メッセージ本文

JMS メッセージ本文は、操作の文書/リテラル・ラッパー・エレメントを表す XML 文書が入ったストリングです。

有効な要求メッセージ本文の単純な例を以下に示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<_6:queryProcessTemplates xmlns:_6="http://www.ibm.com/xmlns/prod/
      websphere/business-process/services/6.0">
<whereClause>PROCESS_TEMPLATE.STATE IN (1)</whereClause>
</_6:queryProcessTemplates>
```

## JMS クライアント・アプリケーションの成果物のコピー

JMS クライアント・アプリケーションを容易に作成するために、WebSphere Process Server 環境から多数の成果物をコピーすることができます。

### このタスクについて

これらの成果物は、JMS メッセージ本文の作成に BOXMLSerializer を使用する場合にのみ必須です。 JMS API の場合、成果物は以下のとおりです。

BFMIF.wsdl

BFMIF.xsd

BPCGen.xsd

wsa.xsd

これらの成果物は、以下のようにして取得できます。

- 成果物を WebSphere Process Server 環境から公開およびエクスポートします。

  これらのクライアント成果物は、*install_root*¥ProcessChoreographer¥client ディレクトリーにあります。
- WebSphere Process Server クライアント CD の *install_root*¥ProcessChoreographer¥client ディレクトリーからファイルをコピーします。

### タスクの結果

## 応答メッセージでのビジネス例外の検査

JMS クライアント・アプリケーションは、すべての応答メッセージのメッセージ・ヘッダーを検査して、ビジネス例外を調べる必要があります。

### このタスクについて

JMS クライアント・アプリケーションはまず、応答メッセージのヘッダーの **IsBusinessException** プロパティーを検査する必要があります。

以下に例を示します。

### 例

```
// receive response message
Message receivedMessage = ((JmsProxy) getToBeInvokedUponObject().receiveMessage();
String strResponse = ((TextMessage) receivedMessage).getText();

if (receivedMessage.getStringProperty("IsBusinessException") {
  // strResponse is a bussiness fault
  // any api can end w/a processFaultMsg
  // the call api also w/a businessFaultMsg
 }
else {
  // strResponse is the output message
}
```

# 例: Business Process Choreographer JMS API を使用して長期実行プロセスを実行する

この例では、長期実行プロセスで処理するために JMS API を使用する汎用のクライアント・アプリケーションの作成方法を示します。

**手順**

1. 316 ページの『JMS インターフェースへのアクセス』の説明に従って、JMS 環境をセットアップします。
2. 以下のように、インストールされたプロセス定義のリストを取得します。
   - `queryProcessTemplates` を送信します。
   - これにより、ProcessTemplate オブジェクトのリストが戻されます。
3. 以下のように、開始アクティビティー (`createInstance="yes"` を指定した receive または pick) のリストを取得します。
   - `getStartActivities` を送信します。
   - これにより、InboundOperationTemplate オブジェクトのリストが戻されます。
4. 入力メッセージを作成します。 これは環境に固有で、事前に配置された、プロセス固有の成果物を使用しなければならないことがあります。
5. プロセス・インスタンスを作成します。
   - `sendMessage` を発行します。

   JMS API とともに `call` 操作を使用して、ビジネス・プロセスによって提供される長期実行の要求/応答操作と対話することもできます。この操作では、長期間たった後でも、操作の結果または障害を指定された応答宛先に戻します。そのため、`call` 操作を使用する場合、`query` および `getOutputMessage` 操作を使用してプロセスの出力または障害メッセージを取得する必要はありません。
6. オプション: 以下のステップを繰り返して、プロセス・インスタンスから出力メッセージを取得します。
   a. `query` を発行して、終了状態のプロセス・インスタンスを取得します。
   b. `getOutputMessage` を発行します。
7. オプション: 以下のように、プロセスによって公開された追加の操作を実行します。
   a. `getWaitingActivities` または `getActiveEventHandlers` を発行して、InboundOperationTemplate オブジェクトのリストを取得します。
   b. 入力メッセージを作成します。
   c. `sendMessage` を発行してメッセージを送信します。
8. オプション: プロセスに対して定義された、またはアクティビティーを含むカスタム・プロパティーを、`getCustomProperties` および `setCustomProperties` で取得および設定します。
9. 以下のように、プロセス・インスタンスでの作業を終了します。
   a. `delete` および `terminate` を送信して、長期実行プロセスでの作業を終了します。

# JSF コンポーネントを使用した、ビジネス・プロセスおよびヒューマン・タスク用 Web アプリケーションの開発

Business Process Choreographer は、いくつかの JavaServer Faces (JSF) コンポーネントを提供します。これらのコンポーネントを拡張および統合して、ビジネス・プロセスおよびヒューマン・タスク機能を Web アプリケーションに追加することができます。

## このタスクについて

WebSphere Integration Developer を使用して Web アプリケーションを作成することができます。ヒューマン・タスクを含むアプリケーションでは、JSF カスタム・クライアントを生成できます。 JSF クライアントの生成について詳しくは、WebSphere Integration Developer のインフォメーション・センターを参照してください。

Business Process Choreographer で提供される JSF コンポーネントを使用して Web クライアントを開発することもできます。

**手順**

1. 動的プロジェクトを作成し、Web プロジェクト・フィーチャー・プロパティーを変更して JSF 基本コンポーネントを組み込みます。

   Web プロジェクトの作成の詳細については、WebSphere Integration Developer インフォメーション・センターにアクセスしてください。
2. 前提条件である Business Process Choreographer Explorer Java アーカイブ (JAR ファイル) を追加します。

   以下のファイルをプロジェクトの WEB-INF/lib ディレクトリーに追加してください。

   - bpcclientcore.jar
   - bfmclientmodel.jar
   - htmclientmodel.jar
   - bpcjsfcomponents.jar

   Web アプリケーションをリモート・サーバーにデプロイする場合、以下のファイルも追加してください。 Business Process Choreographer API にリモート側でアクセスするには、以下のファイルが必要です。

   - bpe137650.jar
   - task137650.jar

   WebSphere Process Server では、これらのすべてのファイルは以下のディレクトリーにあります。

   - Windows システムの場合: *install_root*¥ProcessChoreographer¥client
   - UNIX®、Linux®、および i5/OS® システムの場合: *install_root*/ProcessChoreographer/client
3. 必要な EJB 参照を、Web アプリケーション・デプロイメント記述子 web.xml ファイルに追加します。

```
<ejb-ref id="EjbRef_1">
  <ejb-ref-name>ejb/BusinessProcessHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.ibm.bpe.api.BusinessFlowManagerHome</home>
  <remote>com.ibm.bpe.api.BusinessFlowManager</remote>
</ejb-ref>
<ejb-ref id="EjbRef_2">
  <ejb-ref-name>ejb/HumanTaskManagerEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <home>com.ibm.task.api.HumanTaskManagerHome</home>
  <remote>com.ibm.task.api.HumanTaskManager</remote>
</ejb-ref>
<ejb-local-ref id="EjbLocalRef_1">
  <ejb-ref-name>ejb/LocalBusinessProcessHome</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.ibm.bpe.api.LocalBusinessFlowManagerHome</local-home>
  <local>com.ibm.bpe.api.LocalBusinessFlowManager</local>
</ejb-local-ref>
<ejb-local-ref id="EjbLocalRef_2">
  <ejb-ref-name>ejb/LocalHumanTaskManagerEJB</ejb-ref-name>
  <ejb-ref-type>Session</ejb-ref-type>
  <local-home>com.ibm.task.api.LocalHumanTaskManagerHome</local-home>
  <local>com.ibm.task.api.LocalHumanTaskManager</local>
</ejb-local-ref>
```

4. Business Process Choreographer Explorer JSF コンポーネントを JSF アプリケーションに追加します。
   a. アプリケーションに必要なタグ・ライブラリー参照を JavaServer Pages (JSP) ファイルに追加します。 通常、JSF および HTML タグ・ライブラリーと、JSF コンポーネントに必要とされるタグ・ライブラリーが必要です。
      - `<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>`
      - `<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>`
      - `<%@ taglib uri="http://com.ibm.bpe.jsf/taglib" prefix="bpe" %>`
   b. JSP ページの本体に `<f:view>` タグを追加し、`<f:view>` タグに `<h:form>` タグを追加します。
   c. JSP ファイルに JSF コンポーネントを追加します。

      アプリケーションに応じて、List コンポーネント、Details コンポーネント、CommandBar コンポーネント、または Message コンポーネントを JSP ファイル内に追加します。各コンポーネントの複数のインスタンスを追加することができます。
   d. JSF 構成ファイル内で管理対象 Bean を構成します。

      デフォルトでは、構成ファイルは faces-config.xml ファイルです。このファイルは、Web アプリケーションの WEB-INF ディレクトリーにあります。

      JSP ファイルに追加するコンポーネントに応じて、照会およびその他のラッパー・オブジェクトへの参照を JSF 構成ファイルに追加する必要もあります。的確なエラー処理が行われるようにするには、JSF 構成ファイルで、エラー Bean とナビゲーション・ターゲットの両方をエラー・ページ用に定義する必要もあります。エラー Bean の名前には `BPCError`、エラー・ページのナビゲーション・ターゲットの名前には `error` が使用されていることを確認します。

```
<faces-config>
...
<managed-bean>
  <managed-bean-name>BPCError</managed-bean-name>
  <managed-bean-class>com.ibm.bpc.clientcore.util.ErrorBeanImpl
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

...
<navigation-rule>
...
<navigation-case>
<description>
The general error page.
</description>
<from-outcome>error</from-outcome>
<to-view-id>/Error.jsp</to-view-id>
</navigation-case>
...
</navigation-rule>
</faces-config>
```

エラー・ページをトリガーするエラー状態では、例外がエラー Bean で設定されます。

e. JSF コンポーネントをサポートするために必要なカスタム・コードをインプリメントします。

5. アプリケーションをデプロイします。

アプリケーションを Network Deployment 環境でデプロイしている場合、ターゲット・リソースの Java Naming and Directory Interface (JNDI) 名を、Business Flow Manager および Human Task Manager API をセル内で検出するための値に変更してください。

- ビジネス・プロセス・コンテナーが同じ管理対象セル内の別のサーバー上で構成されている場合、名前には以下の構造があります。

  *cell*/*nodes*/*nodename*/*servers*/*servername*/com/ibm/bpe/api/BusinessManagerHome
  *cell*/*nodes*/*nodename*/*servers*/*servername*/com/ibm/task/api/HumanTaskManagerHome

- ビジネス・プロセス・コンテナーが同じセル内のクラスターで構成されている場合、名前には以下の構造があります。

  *cell*/*clusters*/*clustername*/com/ibm/bpe/api/BusinessFlowManagerHome
  *cell*/*clusters*/*clustername*/com/ibm/task/api/HumanTaskManagerHome

EJB 参照を JNDI 名へマップするか、参照を ibm-web-bnd.xmi ファイルへ手動で追加します。

以下の表に、参照バインディングおよびそのデフォルト・マッピングを示します。

*表 16. 参照バインディングから JNDI 名へのマッピング*

| 参照バインディング | JNDI 名 | コメント |
|---|---|---|
| ejb/BusinessProcessHome | com/ibm/bpe/api/BusinessFlowManagerHome | リモート・セッション Bean |
| ejb/LocalBusinessProcessHome | com/ibm/bpe/api/BusinessFlowManagerHome | ローカル・セッション Bean |
| ejb/HumanTaskManagerEJB | com/ibm/task/api/HumanTaskManagerHome | リモート・セッション Bean |
| ejb/LocalHumanTaskManagerEJB | com/ibm/task/api/HumanTaskManagerHome | ローカル・セッション Bean |

### タスクの結果

デプロイした Web アプリケーションには、Business Process Choreographer Explorer コンポーネントが提供する機能が含まれています。

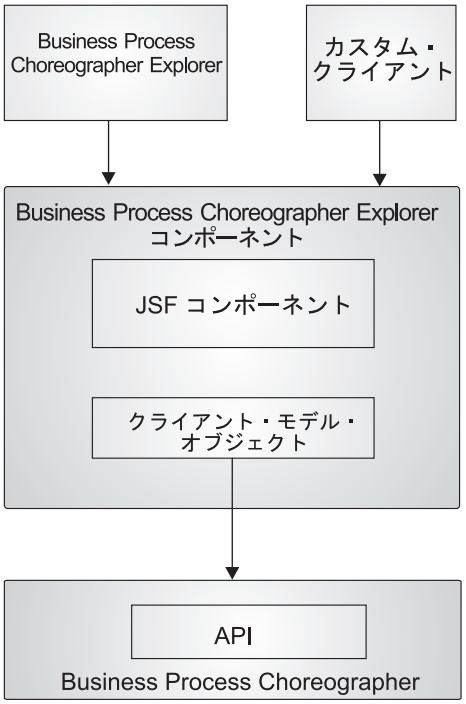
### 次のタスク

プロセスおよびタスク・メッセージにカスタム JSP を使用している場合、JSP をデプロイするために使用される Web モジュールを、カスタム JSF クライアントがマップされるのと同じサーバーにマップする必要があります。

## Business Process Choreographer Explorer コンポーネント

Business Process Choreographer Explorer コンポーネントは、JavaServer Faces (JSF) テクノロジーに基づく構成可能かつ再使用可能なエレメントの集合です。これらのエレメントを Web アプリケーションに組み込むことができます。これにより、Web アプリケーションは、インストール済みビジネス・プロセスおよびヒューマン・タスク・アプリケーションにアクセスできるようになります。

コンポーネントは、JSF コンポーネントのセットおよびクライアント・モデル・オブジェクトのセットで構成されています。コンポーネントから Business Process Choreographer、Business Process Choreographer Explorer、およびその他のカスタム・クライアントへの関係を、次の図に示します。




### JSF コンポーネント

Business Process Choreographer Explorer コンポーネントには、以下の JSF コンポーネントが含まれます。ビジネス・プロセスおよびヒューマン・タスクを操作するための Web アプリケーションをビルドするときに、これらの JSF コンポーネントを JavaServer Pages (JSP) ファイルに組み込みます。

- List コンポーネント

  List コンポーネントは、例えば、タスク、アクティビティー、プロセス・インスタンス、プロセス・テンプレート、作業項目、またはエスカレーションなどの、アプリケーション・オブジェクトのリストをテーブル内に表示します。このコンポーネントには、関連付けられたリスト・ハンドラーがあります。
- Details コンポーネント

  Details コンポーネントは、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。このコンポーネントには、関連付けられた詳細ハンドラーがあります。
- CommandBar コンポーネント

  CommandBar コンポーネントは、ボタンを含むバーを表示します。これらのボタンは、詳細ビュー内のオブジェクトまたはリスト内の選択されたオブジェクトのいずれかに作動するコマンドを表します。これらのオブジェクトは、リスト・ハンドラーまたは詳細ハンドラーによって提供されます。
- Message コンポーネント

  Message コンポーネントは、サービス・データ・オブジェクト (SDO) または単純型のいずれかを含むことのできるメッセージを表示します。

### クライアント・モデル・オブジェクト

クライアント・モデル・オブジェクトは、JSF コンポーネントとともに使用されます。これらのオブジェクトは、基盤となる Business Process Choreographer API のインターフェースの一部をインプリメントし、元のオブジェクトをラップします。クライアント・モデル・オブジェクトは、ラベルの各国語サポートと、一部のプロパティーのコンバーターを提供します。

## JSF コンポーネントでのエラー処理

JavaServer Faces (JSF) コンポーネントはエラー処理の際に、事前定義された管理対象 Bean である BPCError を活用します。エラー・ページをトリガーするエラー状態では、例外がエラー Bean で設定されます。

この Bean は、com.ibm.bpc.clientcore.util.ErrorBean インターフェースをインプリメントします。エラー・ページが表示されるのは、次のような状態のときです。

- リスト・ハンドラー用に定義された照会の実行中にエラーが発生し、エラーがコマンドの execute メソッドによって ClientException エラーとして生成された場合
- ClientException エラーがコマンドの execute メソッドによって生成され、このエラーが ErrorsInCommandException エラーではなく、CommandBarMessage インターフェースのインプリメントもしない場合
- コンポーネント内でエラー・メッセージが表示され、メッセージのハイパーリンクに従っている場合

com.ibm.bpc.clientcore.util.ErrorBeanImpl インターフェースのデフォルト実装を使用できます。

インターフェースは次のように定義されます。

```
public interface ErrorBean {

    public void setException(Exception ex);

    /*
     * This setter method call allows a locale and
     * the exception to be passed. This allows the
     * getExceptionMessage methods to return localized Strings
     *
     */
    public void setException(Exception ex, Locale locale);

    public Exception getException();
    public String getStack();
    public String getNestedExceptionMessage();
    public String getNestedExceptionStack();
    public String getRootExceptionMessage();
    public String getRootExceptionStack();

    /*
     * This method returns the exception message
     * concatenated recursively with the messages of all
     * the nested exceptions.
     */
    public String getAllExceptionMessages();

    /*
     * This method is returns the exception stack
     * concatenated recursively with the stacks of all
     * the nested exceptions.
     */
    public String getAllExceptionStacks();
}
```

## クライアント・モデル・オブジェクトのデフォルトのコンバーターおよびラベル

クライアント・モデル・オブジェクトは、Business Process Choreographer API の対応するインターフェースをインプリメントします。

List コンポーネントと Details コンポーネントはあらゆる Bean で機能します。Bean のプロパティーをすべて表示できます。ただし、Bean のプロパティーに使用されるコンバーターとラベルを設定する場合は、 List コンポーネントの `column` タグまたは Details コンポーネントの `property` タグを使用する必要があります。コンバーターとラベルを設定する代わりに、プロパティーのデフォルトのコンバーターとラベルを定義できます。これらを定義するには、次の静的メソッドを定義します。定義できる静的メソッドを次に示します。

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
       getConverter(String property);
```

次の表に、対応する Business Flow Manager API クラスと Human Task Manager API クラスを実装し、そのプロパティーにデフォルトのラベルとコンバーターを提供するクライアント・モデル・オブジェクトを示します。インターフェースをこのようにラップすることにより、ロケールを区別するラベルと、プロパティーのセット用のコンバーターが提供されます。以下の表に、Business Process Choreographer インターフェースから対応するクライアント・モデル・オブジェクトへのマッピングを示します。

表 17. *Business Process Choreographer* インターフェースからクライアント・モデル・オブジェクトへのマッピング

| Business Process Choreographer インターフェース | クライアント・モデル・オブジェクト・クラス |
|---|---|
| com.ibm.bpe.api.ActivityInstanceData | com.ibm.bpe.clientmodel.bean.ActivityInstanceBean |
| com.ibm.bpe.api.ActivityServiceTemplateData | com.ibm.bpe.clientmodel.bean.ActivityServiceTemplateBean |
| com.ibm.bpe.api.ProcessInstanceData | com.ibm.bpe.clientmodel.bean.ProcessInstanceBean |
| com.ibm.bpe.api.ProcessTemplateData | com.ibm.bpe.clientmodel.bean.ProcessTemplateBean |
| com.ibm.task.api.Escalation | com.ibm.task.clientmodel.bean.EscalationBean |
| com.ibm.task.api.Task | com.ibm.task.clientmodel.bean.TaskInstanceBean |
| com.ibm.task.api.TaskTemplate | com.ibm.task.clientmodel.bean.TaskTemplateBean |

## JSF アプリケーションへの List コンポーネントの追加

Business Process Choreographer Explorer List コンポーネントを使用して、例えばビジネス・プロセス・インスタンスやタスク・インスタンスなどの、クライアント・モデル・オブジェクトのリストを表示します。

**手順**

1. List コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   `bpe:list` タグを `h:form` タグに追加します。 `bpe:list` タグには、モデル属性が含まれていなければなりません。`bpe:column` タグを `bpe:list` に追加して、リスト内の各行に表示されるオブジェクトのプロパティーを追加します。

   以下の例では、List コンポーネントを追加してタスク・インスタンスを表示する方法を示します。

   ```
   <h:form>

      <bpe:list model="#{TaskPool}">
         <bpe:column name="name" action="taskInstanceDetails" />
         <bpe:column name="state" />
         <bpe:column name="kind" />
         <bpe:column name="owner" />
         <bpe:column name="originator" />
      </bpe:list>

   </h:form>
   ```

   モデル属性は、TaskPool という管理対象 Bean を参照します。管理対象 Bean は、リストが操作を繰り返す対象となる Java オブジェクトのリストを提供し、次に個々の行を表示します。

2. `bpe:list` タグで参照されている管理対象 Bean を構成します。

   List コンポーネントの場合、この管理対象 Bean は、com.ibm.bpe.jsf.handler.BPCListHandler クラスのインスタンスでなければなりません。

   以下の例では、TaskPool 管理対象 Bean を構成ファイルに追加する方法を示します。

   ```
   <managed-bean>
   <managed-bean-name>TaskPool</managed-bean-name>
   <managed-bean-class>com.ibm.bpe.jsf.handler.BPCListHandler</managed-bean-class>
   ```

```
<managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>query</property-name>
      <value>#{TaskPoolQuery}</value>
   </managed-property>
   <managed-property>
      <property-name>type</property-name>
      <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>
</managed-bean>

<managed-bean>
<managed-bean-name>TaskPoolQuery</managed-bean-name>
<managed-bean-class>sample.TaskPoolQuery</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
   <managed-property>
     <property-name>type</property-name>
     <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>
</managed-bean>

<managed-bean>
<managed-bean-name>htmConnection</managed-bean-name>
<managed-bean-class>com.ibm.task.clientmodel.HTMConnection</managed-bean-class>
<managed-bean-scope>application</managed-bean-scope>
   <managed-property>
      <property-name>jndiName</property-name>
      <value>java:comp/env/ejb/LocalHumanTaskManagerEJB</value>
   </managed-property>
</managed-bean>
```

例では、照会およびタイプの 2 つの構成可能プロパティーが TaskPool に含まれていることを示しています。照会プロパティーの値は、TaskPoolQuery という別の管理対象 Bean を参照しています。タイプ・プロパティーの値は、Bean クラスを指定します。そのクラスのプロパティーは、表示されたリストの列に示されます。関連する照会インスタンスは、プロパティー型を持つことも可能です。プロパティー型が指定された場合、それはリスト・ハンドラーに指定された型と同一でなければなりません。

照会の結果を強い型の Bean のリストとして表すことができる限り、任意のタイプの照会ロジックを JSF アプリケーションに追加できます。例えば、TaskPoolQuery は com.ibm.task.clientmodel.bean.TaskInstanceBean オブジェクトのリストを使用して実装されます。

3. リスト・ハンドラーによって参照される管理対象 Bean 用のカスタム・コードを追加します。

   以下の例では、TaskPool 管理対象 Bean 用のカスタム・コードを追加する方法を示します。

```
public class TaskPoolQuery implements Query {

  public List execute throws ClientException {

    // Examine the faces-config file for a managed bean "htmConnection".
    //
    FacesContext ctx = FacesContext.getCurrentInstance();
    Application  app = ctx.getApplication();
    ValueBinding htmVb = app.createValueBinding("#{htmConnection}");
    htmConnection = (HTMConnection) htmVb.getValue(ctx);
    HumanTaskManagerService taskService =
         htmConnection.getHumanTaskManagerService();
```

```
    // Then call the actual query method on the Human Task Manager service.
    //
    // リストに表示するすべてのプロパティーのデータベース列を
    // select ステートメントに追加します
    //
    QueryResultSet queryResult = taskService.query(
     "DISTINCT TASK.TKIID, TASK.NAME, TASK.KIND, TASK.STATE, TASK.TYPE,"
      + "TASK.STARTER, TASK.OWNER, TASK.STARTED, TASK.ACTIVATED, TASK.DUE,
         TASK.EXPIRES, TASK.PRIORITY",
      "TASK.KIND IN(101,102,105) AND TASK.STATE IN(2)
        AND WORK_ITEM.REASON IN (1)",
      (String)null,
      (Integer)null,
      (TimeZone)null);
    List applicationObjects = transformToTaskList ( queryResult );
    return applicationObjects ;
  }


  private List transformToTaskList(QueryResultSet result) {

ArrayList array = null;
int entries = result.size();
array = new ArrayList( entries );

// Transforms each row in the QueryResultSet to a task instance beans.
 for (int i = 0; i < entries; i++) {
   result.next();
   array.add( new TaskInstanceBean( result, connection ));
 }
 return array ;
  }
}
```

TaskPoolQuery Bean は、Java オブジェクトのプロパティーを照会します。この Bean は、com.ibm.bpc.clientcore.Query インターフェースをインプリメントする必要があります。リスト・ハンドラーは、内容を最新表示するときに、照会の execute メソッドを呼び出します。呼び出しによって、Java オブジェクトのリストが戻されます。getType メソッドは、戻された Java オブジェクトのクラス名を戻す必要があります。

## タスクの結果

これで、JSF アプリケーションは、例えば状態、種類、所有者、ユーザーが使用可能なタスク・インスタンスのオリジネーターなどの、要求されたオブジェクトのリストのプロパティーを表示する JavaServer ページを含むようになります。

## リストの処理方法

List コンポーネントのインスタンスはすべて com.ibm.bpe.jsf.handler.BPCListHandler クラスのインスタンスに関連しています。

このリスト・ハンドラーは関連するリスト内で選択された項目をトラッキングし、さまざまな種類の項目用の詳細ページにリスト項目を関連付ける通知メカニズムを提供します。リスト・ハンドラーは、bpe:list タグの**モデル**属性を介して List コンポーネントにバインドされます。

リスト・ハンドラーの通知メカニズムは、 com.ibm.bpe.jsf.handler.ItemListener インターフェースを使用してインプリメントされます。このインターフェースの実装は、JavaServer Faces (JSF) アプリケーションの構成ファイルに登録できます。

リスト内のリンクがクリックされると、通知がトリガーされます。 **action** 属性が設定されているすべての列についてリンクがレンダリングされます。 **action** 属性の値は JSF ナビゲーション・ターゲットか、JSF ナビゲーション・ターゲットを戻す JSF アクション・メソッドのいずれかです。

BPCListHandler クラスは、refreshList メソッドを提供します。このメソッドを JSF メソッド・バインディングで使用して、照会を再実行するためのユーザー・インターフェース制御をインプリメントすることができます。

### 照会の実装

リスト・ハンドラーを使用すると、すべての種類のオブジェクトおよびそれらのプロパティーを表示できます。表示されるリストの内容は、リスト・ハンドラー用に構成された com.ibm.bpc.clientcore.Query インターフェースの実装によって戻されるオブジェクトのリストによって異なります。照会は、BPCListHandler クラスの setQuery メソッドを使用してプログラマチックに設定することもできますし、アプリケーションの JSF 構成ファイルで構成することもできます。

照会は、Business Process Choreographer API に対してだけでなく、コンテンツ管理システムやデータベースなど、アプリケーションからアクセスできるその他の情報ソースに対しても実行できます。要件は、照会の結果が execute メソッドでオブジェクトの `java.util.List` として戻されることだけです。

戻されるオブジェクトのタイプは、照会が定義されたリストの列に表示されるすべてのプロパティーに対して適切な getter メソッドを使用できることを保証する必要があります。戻されるオブジェクトのタイプがリスト定義に適合することを確認するには、faces 構成ファイルで定義されている BPCListHandler インスタンス上のタイプ・プロパティーの値を、戻されるオブジェクトの完全修飾クラス名に設定します。この名前は、照会実装の getType 呼び出しで戻すことができます。実行時に、リスト・ハンドラーはオブジェクト・タイプが定義に準拠していることを確認します。

リスト内の特定の項目にエラー・メッセージをマップするには、照会によって戻されたオブジェクトが署名 `public Object getID()` でメソッドをインプリメントする必要があります。

### デフォルトのコンバーターおよびラベル

照会によって戻される項目は Bean である必要があり、そのクラスは BPCListHandler クラスまたは com.ibm.bpc.clientcore.Query インターフェースの定義でタイプとして指定されたクラスと一致している必要があります。さらに、List コンポーネントは、項目クラスまたはスーパークラスが以下のメソッドを実装しているかどうかを検査します。

```
static public String getLabel(String property,Locale locale);
static public com.ibm.bpc.clientcore.converter.SimpleConverter
       getConverter(String property);
```

これらのメソッドが Bean に対して定義されている場合、List コンポーネントはリストのデフォルト・ラベルとして label を使用し、プロパティーのデフォルト・コンバーターとして SimpleConverter を使用します。これらの設定は、bpe:list タグの **label** および **converterID** 属性で上書きできます。詳しくは、SimpleConverter インターフェースおよび ColumnTag クラスの Javadoc を参照してください。

## ユーザー固有の時間帯情報

JavaServer Faces (JSF) コンポーネントは、List コンポーネントのユーザー指定の時間帯情報を処理するためのユーティリティーを提供します。

BPCListHandler クラスは、com.ibm.bpc.clientcore.util.User インターフェースを使用して、各ユーザーの時間帯およびロケールに関する情報を取得します。List コンポーネントは、JavaServer Faces (JSF) 構成ファイルでインターフェースの実装が **user** を管理対象 Bean 名として設定されていると予期します。構成ファイル内でこの項目が欠けている場合は、WebSphere Process Server が動作している時間帯が戻されます。

com.ibm.bpc.clientcore.util.User インターフェースは次のように定義されています。

```
public interface User {

    /**
     * The locale used by the client of the user.
     * @return Locale.
     */
    public Locale getLocale();
   /**
    * The time zone used by the client of the user.
    * @return TimeZone.
    */
    public TimeZone getTimeZone();

   /**
    * The name of the user.
    * @return name of the user.
    */
    public String getName();
}
```

## List コンポーネントでのエラー処理

List コンポーネントを使用して、JSF アプリケーション内のリストを表示する場合、com.ibm.bpe.jsf.handler.BPCListHandler クラスが提供するエラー処理機能を利用できます。

### 照会の実行時またはコマンドの実行時に発生するエラー

照会の実行中にエラーが発生した場合、BPCListHandler クラスは、不十分なアクセス権限によるエラーとその他の例外とを区別します。不十分なアクセス権限によるエラーをキャッチするには、照会の execute メソッドによってスローされる ClientException の **rootCause** パラメーターが com.ibm.bpe.api.EngineNotAuthorizedException または com.ibm.task.api.NotAuthorizedException 例外である必要があります。List コンポーネントは、照会の結果の代わりにエラー・メッセージを表示します。

エラーが不十分なアクセス権限によるものでない場合、BPCListHandler クラスは例外オブジェクトを、JSF アプリケーション構成ファイルの BPCError キーで定義した com.ibm.bpc.clientcore.util.ErrorBean インターフェースの実装に渡します。例外が設定されている場合は、エラー・ナビゲーション・ターゲットが呼び出されます。

### リストに表示される項目の処理時に発生するエラー

BPCListHandler クラスは、com.ibm.bpe.jsf.handler.ErrorHandler インターフェースをインプリメントします。setErrors メソッドでタイプ java.util.Map のマップ・パラメーターを使用して、これらのエラーに関する情報を提供することができます。このマップには、キーとして ID が、値として例外が含まれています。 ID は、エラーの原因となったオブジェクトの getID メソッドによって戻された値である必要があります。マップが設定されていて、リスト内に表示されている項目のいずれかに ID のいずれかが一致する場合は、リスト・ハンドラーによって、エラー・メッセージを含む列が自動的にリストに追加されます。

リスト内のエラー・メッセージが古くなるのを避けるため、エラー・マップをリセットしてください。次の状況では、マップは自動的にリセットされます。

- refreshList メソッドの BPCListHandler クラスが呼び出される。
- BPCListHandler クラスで新規照会が設定されている。
- CommandBar コンポーネントを使用して、リストの項目でアクションがトリガーされている。CommandBar コンポーネントは、エラー処理のメソッドの 1 つとしてこのメカニズムを使用します。

## List コンポーネント: タグ定義

Business Process Choreographer Explorer List コンポーネントは、例えば、タスク、アクティビティー、プロセス・インスタンス、プロセス・テンプレート、作業項目、およびエスカレーションなどの、オブジェクトのリストをテーブル内に表示します。

List コンポーネントは、JSF コンポーネント・タグである `bpe:list` と `bpe:column` から構成されます。`bpe:column` タグは、`bpe:list` タグのサブエレメントです。

### コンポーネント・クラス

com.ibm.bpe.jsf.component.ListComponent

### 構文例

```
<bpe:list model="#{ProcessTemplateList}">
          rows="20"
          styleClass="list"
          headerStyleClass="listHeader"
          rowClasses="normal">

    <bpe:column name="name" action="processTemplateDetails"/>
    <bpe:column name="validFromTime"/>
    <bpe:column name="executionMode" label="Execution mode"/>
    <bpe:column name="state" converterID="my.state.converter"/>
    <bpe:column name="autoDelete"/>
    <bpe:column name="description"/>

</bpe:list>
```

## タグ属性

bpe:list タグの本体には、bpe:column タグのみを含めることができます。テーブルがレンダリングされるとき、List コンポーネントは、アプリケーション・オブジェクトのリストで処理を繰り返し、オブジェクトごとにすべての列をレンダリングします。

*表 18. bpe:list 属性*

| 属性 | 必須 | 説明 |
|---|---|---|
| buttonStyleClass | いいえ | フッター領域内のボタンのレンダリング用のカスケーディング・スタイル・シート (CSS) スタイル・クラス。 |
| cellStyleClass | いいえ | 個々のテーブル・セルのレンダリング用の CSS スタイル・クラス。 |
| checkbox | いいえ | 複数の項目を選択するためのチェック・ボックスを提供するかどうかを決定します。この属性には true または false のいずれかの値が使用されます。値が true に設定されている場合は、チェック・ボックス列がレンダリングされます。 |
| headerStyleClass | いいえ | テーブル・ヘッダーのレンダリング用の CSS スタイル・クラス。 |
| model | はい | com.ibm.bpe.jsf.handler.BPCListHandler クラスの管理対象 Bean 用の値バインディング。 |
| rows | いいえ | ページに表示される行数。項目数が行数を超える場合は、テーブルの最後にページ送りボタンが表示されます。この属性では、値の式はサポートされていません。 |
| rowClasses | いいえ | テーブル内の行のレンダリング用の CSS スタイル・クラス。 |
| selectAll | いいえ | この属性が true に設定されている場合は、リスト内のすべての項目がデフォルトで選択されます。 |
| styleClass | いいえ | タイトル、行、およびページ送りボタンを含むテーブル全体のレンダリングの CSS スタイル・クラス。 |

*表 19. bpe:column 属性*

| 属性 | 必須 | 説明 |
|---|---|---|
| action | いいえ | この属性が指定されている場合は、列でリンクがレンダリングされます。リンクがクリックされると、JavaServer Faces アクション・メソッドまたは Faces ナビゲーション・ターゲットが起動されます。シグニチャーが String method() である JavaServer Faces アクション・メソッド。 |

*表 19. bpe:column 属性 (続き)*

| 属性 | 必須 | 説明 |
|---|---|---|
| converterID | いいえ | プロパティー値の変換に使用する Faces コンバーター ID。この属性が設定されていない場合、モデルによりこのプロパティーに設定された Faces コンバーター ID が使用されます。 |
| label | いいえ | 列のヘッダーのラベルまたはテーブル・ヘッダー行のセルのラベルとして使用されるリテラルまたは値バインディング式。この属性が設定されていない場合、モデルによりこのプロパティーに設定されたラベルが使用されます。 |
| name | はい | この列に表示されるプロパティーの名前。 |

## JSF アプリケーションへの Details コンポーネントの追加

Business Process Choreographer Explorer Details コンポーネントを使用して、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。

**手順**

1. Details コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   `bpe:details` タグを `<h:form>` タグに追加します。`bpe:details` タグには、**model** 属性が含まれていなければなりません。`bpe:property` タグを使用して Details コンポーネントにプロパティーを追加することができます。

   以下の例では、Details コンポーネントを追加して、タスク・インスタンスのプロパティーのいくつかを表示する方法を示します。

   ```
   <h:form>

      <bpe:details model="#{TaskInstanceDetails}">
         <bpe:property name="displayName" />
         <bpe:property name="owner" />
         <bpe:property name="kind" />
         <bpe:property name="state" />
         <bpe:property name="escalated" />
         <bpe:property name="suspended" />
         <bpe:property name="originator" />
         <bpe:property name="activationTime" />
         <bpe:property name="expirationTime" />
      </bpe:details>

   </h:form>
   ```

   **model** 属性は、TaskInstanceDetails という管理対象 Bean を参照します。Bean は、Java オブジェクトのプロパティーを提供します。

2. `bpe:details` タグで参照されている管理対象 Bean を構成します。

   Details コンポーネントの場合、この管理対象 Bean は、com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスのインスタンスでなければなり

ません。このハンドラー・クラスは、Java オブジェクトをラップし、そのパブリック・プロパティーを Details コンポーネントに公開します。

以下の例では、TaskInstanceDetails 管理対象 Bean を構成ファイルに追加する方法を示します。

```
<managed-bean>
   <managed-bean-name>TaskInstanceDetails</managed-bean-name>
   <managed-bean-class>com.ibm.bpe.jsf.handler.BPCDetailsHandler</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
   <managed-property>
      <property-name>type</property-name>
      <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
   </managed-property>
</managed-bean>
```

例では、TaskInstanceDetails Bean に構成可能な type プロパティーが含まれることを示しています。タイプ・プロパティーの値は、Bean クラス (com.ibm.task.clientmodel.bean.TaskInstanceBean) を指定します。そのクラスのプロパティーは、表示された詳細の行に示されます。Bean クラスは任意の JavaBeans クラスにすることができます。Bean がデフォルト・コンバーターおよびプロパティー・ラベルを提供する場合、そのコンバーターおよびラベルが、List コンポーネントの場合と同じようにしてレンダリングに使用されます。

### タスクの結果

これで、JSF アプリケーションは、例えばタスク・インスタンスの詳細などの、指定されたオブジェクトの詳細を表示する JavaServer ページを含むようになります。

## Details コンポーネント: タグ定義

Business Process Choreographer Explorer Details コンポーネントは、タスク、作業項目、アクティビティー、プロセス・インスタンス、およびプロセス・テンプレートのプロパティーを表示します。

Details コンポーネントは、JSF コンポーネント・タグである bpe:details と bpe:property から構成されます。bpe:property タグは、bpe:details タグのサブエレメントです。

### コンポーネント・クラス

com.ibm.bpe.jsf.component.DetailsComponent

### 構文例

```
<bpe:details model="#{MyActivityDetails}">
    <bpe:property name="name"/>
    <bpe:property name="owner"/>
    <bpe:property name="activated"/>
</bpe:details>
```

```
<bpe:details model="#{MyActivityDetails}" style="style" styleClass="cssStyle">
             style="style"
             styleClass="cssStyle"
</bpe:details>
```

### タグ属性

bpe:property タグを使用して、表示される属性のサブセットおよびこれらの属性が表示される順序の両方を指定します。詳細タグに属性タブが含まれていない場合、モデル・オブジェクトの使用可能な属性がすべてレンダリングされます。

*表 20. bpe:details 属性*

| 属性 | 必須 | 説明 |
|---|---|---|
| columnClasses | いいえ | 列のレンダリング用のカスケーディング・スタイル・シート (CSS) のスタイル・クラスをコンマで区切ったリスト。 |
| id | いいえ | コンポーネントの JavaServer Faces ID。 |
| model | はい | com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスの管理対象 Bean 用の値バインディング。 |
| rowClasses | いいえ | 行のレンダリング用の、コンマで区切られた CSS スタイル・クラスのリスト。 |
| styleClass | いいえ | HTML エレメントのレンダリングに使用される CSS クラス。 |

*表 21. bpe:property 属性*

| 属性 | 必須 | 説明 |
|---|---|---|
| converterID | いいえ | JavaServer Faces (JSF) 構成ファイルでコンバーターを登録するために使用される ID。 |
| label | いいえ | プロパティーのラベル。この属性が設定されていない場合、クライアント・モデル・クラスによってデフォルト・ラベルが提供されます。 |
| name | はい | 表示されるプロパティーの名前。この名前は、対応するクライアント・モデル・クラスで定義されているように、名前付きプロパティーに対応していなければなりません。 |

## JSF アプリケーションへの CommandBar コンポーネントの追加

Business Process Choreographer Explorer CommandBar コンポーネントを使用して、ボタンを含むバーを表示します。これらのボタンは、オブジェクトの詳細ビューまたはリスト内の選択されたオブジェクトで作動するコマンドを表します。

### このタスクについて

ユーザーがユーザー・インターフェースのボタンをクリックすると、対応するコマンドが選択されたオブジェクトで実行されます。 CommandBar コンポーネントは、JavaServer Faces (JSF) アプリケーションに追加して拡張することができます。

#### 手順

1. CommandBar コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

bpe:commandbar タグを <h:form> タグに追加します。bpe:commandbar タグには、モデル属性が含まれていなければなりません。

以下の例では、タスク・インスタンス・リストに refresh コマンドと claim コマンドを提供する CommandBar コンポーネントを追加する方法を示します。

```
<h:form>

   <bpe:commandbar model="#{TaskInstanceList}">
      <bpe:command commandID="Refresh" >
                   action="#{TaskInstanceList.refreshList}"
                   label="Refresh"/>

      <bpe:command commandID="MyClaimCommand" >
                   label="Claim" >
                   commandClass="<customcode>"/>
   </bpe:commandbar>

</h:form>
```

**model** 属性は、管理対象 Bean を参照します。この Bean は、ItemProvider インターフェースをインプリメントし、選択された Java オブジェクトを提供する必要があります。CommandBar コンポーネントは、通常、同一の JSP ファイル内の List コンポーネントまたは Details コンポーネントのいずれかとともに使用されます。一般に、タグで指定されたモデルは、同一ページの List コンポーネントまたは Details コンポーネントで指定されたモデルと同じです。そのため、例えば List コンポーネントの場合、コマンドはリスト内の選択された項目に対して作動します。

この例では、**model** 属性は TaskInstanceList 管理対象 Bean を参照します。この Bean は、タスク・インスタンス・リストの選択されたオブジェクトを提供します。この Bean は、ItemProvider インターフェースをインプリメントする必要があります。このインターフェースは、BPCListHandler クラスおよび BPCDetailsHandler クラスによってインプリメントされます。

2. オプション: bpe:commandbar タグで参照されている管理対象 Bean を構成します。

   CommandBar **model** 属性が、例えばリスト・ハンドラーまたは詳細ハンドラー用に既に構成済みの管理対象 Bean を参照する場合、それ以上の構成は必要ありません。モデルで BPCListHandler クラスも BPCDetailsHandler クラスも使用しない場合は、ItemProvider インターフェースを実装したクラスを所有する別のオブジェクトを参照する必要があります。

3. カスタム・コマンドをインプリメントするコードを JSF アプリケーションに追加します。

   以下のコード断片は、Command インターフェースを実装するコマンド・クラスの作成方法を示します。このコマンド・クラス (MyClaimCommand) は、JSP ファイル内の bpe:command タグで参照されます。

```
public class MyClaimCommand implements Command {

  public String execute(List selectedObjects) throws ClientException {
     if( selectedObjects != null && selectedObjects.size() > 0 ) {
        try {
         // Determine HumanTaskManagerService from an HTMConnection bean.
         // Configure the bean in the faces-config.xml for easy access
```

```
        // in the JSF application.
        FacesContext ctx = FacesContext.getCurrentInstance();
        ValueBinding vb =
          ctx.getApplication().createValueBinding("{htmConnection}");
        HTMConnection htmConnection = (HTMConnection) htmVB.getValue(ctx);
        HumanTaskManagerService htm =
           htmConnection.getHumanTaskManagerService();

        Iterator iter = selectedObjects.iterator() ;
        while( iter.hasNext() ) {
          try {
               TaskInstanceBean task = (TaskInstanceBean) iter.next() ;
               TKIID tiid = task.getID() ;

               htm.claim( tiid ) ;
               task.setState( new Integer(TaskInstanceBean.STATE_CLAIMED ) ) ;

          }
          catch( Exception e ) {
            ;   // Error while iterating or claiming task instance.
                 // Ignore for better understanding of the sample.
          }
        }
      }
      catch( Exception e ) {
        ;  // Configuration or communication error.
          // Ignore for better understanding of the sample
      }
    }
    return null;
  }

 // Default implementations
 public boolean isMultiSelectEnabled() { return false; }
 public boolean[] isApplicable(List itemsOnList) {return null; }
 public void setContext(Object targetModel) {; // Not used here }
}
```

コマンドは以下のように処理されます。

a. ユーザーがコマンド・バーの対応するボタンをクリックすると、コマンドが起動されます。CommandBar コンポーネントは、**model** 属性で指定された項目プロバイダーから選択された項目を検索し、選択されたオブジェクトのリストを commandClass インスタンスの execute メソッドに渡します。

b. **commandClass** 属性は、コマンド・インターフェースをインプリメントするカスタム・コマンド実装を参照します。つまりこのコマンドは、public String execute(List selectedObjects) throws ClientException メソッドをインプリメントする必要があります。コマンドが戻した結果は、JSF アプリケーションの次のナビゲーション規則を決定するために使用されます。

c. コマンドの完了後、CommandBar コンポーネントは **action** 属性を評価します。**action** 属性は、静的ストリングである場合も、`public String Method()` というシグニチャーの JSF アクション・メソッドへのメソッド・バインディングである場合もあります。**action** 属性を使用して、コマンド・クラスの結果をオーバーライドするか、またはナビゲーション規則の結果を明示的に指定します。コマンドが ErrorsInCommandException 例外以外の例外を生成した場合、**action** 属性は処理されません。

d. **commandClass** 属性にコマンド・クラスが指定されていない場合、アクションが即時に呼び出されます。例えば、例の中のリフレッシュ・コマンドの場合、JSF 値式 `#{TaskInstanceList.refreshList}` がコマンドの代わりに呼び出されます。

## タスクの結果

これで、JSF アプリケーションは、カスタマイズされたコマンド・バーをインプリメントする JavaServer ページを含むようになります。

## コマンドの処理方法

CommandBar コンポーネントを使用して、アプリケーションにアクション・ボタンを追加します。コンポーネントは、ユーザー・インターフェースでのアクション用のボタンを作成し、ボタンがクリックされたときに作成されるイベントを処理します。

これらのボタンは、BPCListHandler クラスや BPCDetailsHandler クラスなど、com.ibm.bpe.jsf.handler.ItemProvider インターフェースによって戻されるオブジェクトで動作する機能を起動します。 CommandBar コンポーネントは、`bpe:commandbar` タグで**モデル**属性の値によって定義された項目プロバイダーを使用します。

アプリケーションのユーザー・インターフェースのコマンド・バー・セクションにあるボタンをクリックすると、関連するイベントが CommandBar コンポーネントによって次のように処理されます。

1. CommandBar コンポーネントは、イベントを生成したボタンに対して指定された com.ibm.bpc.clientcore.Command インターフェースの実装を示します。
2. CommandBar コンポーネントに関連するモデルが com.ibm.bpe.jsf.handler.ErrorHandler インターフェースをインプリメントすると、前のイベントからのエラー・メッセージを削除するため、clearErrorMap メソッドが呼び出されます。
3. ItemProvider インターフェースの getSelectedItems メソッドが呼び出されます。戻された項目のリストは、コマンドの execute メソッドに渡され、コマンドが呼び出されます。
4. CommandBar コンポーネントは、JavaServer Faces (JSF) ナビゲーション・ターゲットを決定します。`bpe:commandbar` タグで**アクション**属性が指定されていない場合は、execute メソッドの戻り値によってナビゲーション・ターゲットが指定されます。**アクション**属性が JSF メソッド・バインディングに設定されている場合は、メソッドによって戻されたストリングがナビゲーション・ターゲットと解釈されます。**アクション**属性は、明示的なナビゲーション・ターゲットも指定します。

## CommandBar コンポーネント: タグ定義

Business Process Choreographer Explorer CommandBar コンポーネントは、ボタンを含むバーを表示します。これらのボタンは、詳細ビュー内のオブジェクトまたはリスト内の選択されたオブジェクトに作動します。

CommandBar コンポーネントは、JSF コンポーネント・タグである `bpe:commandbar` と `bpe:command` から構成されます。 `bpe:command` タグは、`bpe:commandbar` タグのサブエレメントです。

## コンポーネント・クラス

com.ibm.bpe.jsf.component.CommandBarComponent

## 構文例

```
<bpe:commandbar model="#{TaskInstanceList}">

    <bpe:command
       commandID="Work on"
       label="Work on..."
       commandClass="com.ibm.bpc.explorer.command.WorkOnTaskCommand"
       context="#{TaskInstanceDetailsBean}"/>

    <bpe:command
       commandID="Cancel"
       label="Cancel"
       commandClass="com.ibm.task.clientmodel.command.CancelClaimTaskCommand"
       context="#{TaskInstanceList}"/>

</bpe:commandbar>
```

## タグ属性

*表 22. bpe:commandbar 属性*

| 属性 | 必須 | 説明 |
|---|---|---|
| buttonStyleClass | いいえ | コマンド・バー内のボタンのレンダリングに使用されるカスケーディング・スタイル・シート (CSS) スタイル・クラス。 |
| id | いいえ | コンポーネントの JavaServer Faces ID。 |
| model | はい | ItemProvider インターフェースをインプリメントする管理対象 Bean に対する値バインディング式。通常、この管理対象 Bean は、同一の JavaServer Pages (JSP) ファイル内の List コンポーネントまたは Details コンポーネントによって CommandBar コンポーネントとして使用される com.ibm.bpe.jsf.handler.BPCListHandler クラスまたは com.ibm.bpe.jsf.handler.BPCDetailsHandler クラスです。 |
| styleClass | いいえ | コマンド・バーのレンダリングに使用される CSS スタイル・クラス。 |

*表 23. bpe:command 属性*

| 属性 | 必須 | 説明 |
|---|---|---|
| action | いいえ | JavaServer Faces アクション・メソッドまたはコマンド・ボタンにより起動される Faces ナビゲーション・ターゲット。アクションにより戻されるナビゲーション・ターゲットは、その他のナビゲーション・ルールをすべて上書きします。このアクションは、例外がスローされない場合、またはコマンドから ErrorsInCommandException 例外がスローされる場合に呼び出されます。 |
| commandClass | いいえ | コマンド・クラスの名前。このクラスのインスタンスは CommandBar コンポーネントにより作成され、コマンド・ボタンが選択されると実行されます。 |
| commandID | はい | コマンドの ID。 |
| context | いいえ | **commandClass** 属性を使用して指定されたコマンドのコンテキストを提供するオブジェクト。コマンド・バーが初めてアクセスされると、コンテキスト・オブジェクトが取得されます。 |
| immediate | いいえ | コマンドが起動される時期を指定します。この属性の値が `true` の場合は、ページの入力が処理される前にコマンドが起動されます。デフォルトは `false` です。 |
| label | はい | コマンド・バーでレンダリングされるボタンのラベル。 |
| rendered | いいえ | ボタンがレンダリングされるかどうかを判別します。属性の値は、ブール値または値の式のいずれかにすることができます。 |
| styleClass | いいえ | ボタンのレンダリングに使用される CSS スタイル・クラス。このスタイルは、コマンド・バーに定義されたボタン・スタイルをオーバーライドします。 |

# JSF アプリケーションへの Message コンポーネントの追加

Business Process Choreographer Explorer Message コンポーネントを使用して、JavaServer Faces (JSF) アプリケーション内で、データ・オブジェクトおよびプリミティブ型をレンダリングします。

## このタスクについて

メッセージ型がプリミティブ型である場合、ラベルおよび入力フィールドがレンダリングされます。メッセージ型がデータ・オブジェクトである場合、コンポーネントはオブジェクトを全探索し、オブジェクト内のエレメントをレンダリングします。

**手順**

1. Message コンポーネントを JavaServer Pages (JSP) ファイルに追加します。

   bpe:form タグを <h:form> タグに追加します。 bpe:form タグには、model 属性が含まれていなければなりません。

   以下の例では、Message コンポーネントを追加する方法を示します。

   ```
   <h:form>

      <h:outputText value="Input Message" />
      <bpe:form model="#{MyHandler.inputMessage}" readOnly="true" />

      <h:outputText value="Output Message" />
      <bpe:form model="#{MyHandler.outputMessage}" />

   </h:form>
   ```

   Message コンポーネントの **model** 属性は、com.ibm.bpc.clientcore.MessageWrapper オブジェクトを参照します。このラッパー・オブジェクトは、サービス・データ・オブジェクト (SDO) オブジェクトか、または int や boolean などの Java プリミティブ型のいずれかをラップします。例では、メッセージは MyHandler 管理対象 Bean のプロパティーによって提供されます。

2. bpe:form タグで参照されている管理対象 Bean を構成します。

   以下の例では、MyHandler 管理対象 Bean を構成ファイルに追加する方法を示します。

   ```
   <managed-bean>
   <managed-bean-name>MyHandler</managed-bean-name>
   <managed-bean-class>com.ibm.bpe.sample.jsf.MyHandler</managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>

      <managed-property>
         <property-name>type</property-name>
         <value>com.ibm.task.clientmodel.bean.TaskInstanceBean</value>
      </managed-property>

   </managed-bean>
   ```

3. JSF アプリケーションにカスタム・コードを追加します。

   以下の例では、入力メッセージおよび出力メッセージをインプリメントする方法を示します。

   ```
   public class MyHandler implements ItemListener {

     private TaskInstanceBean taskBean;
     private MessageWrapper inputMessage, outputMessage

     /* Listener method, e.g. when a task instance was selected in a list handler.
      * Ensure that the handler is registered in the faces-config.xml or manually.
      */
     public void itemChanged(Object item) {
       if( item instanceof TaskInstanceBean ) {
           taskBean = (TaskInstanceBean) item ;
       }
     }

     /* Get the input message wrapper
      */
     public MessageWrapper getInputMessage() {
   ```

```
    try{
        inputMessage = taskBean.getInputMessageWrapper() ;
    }
    catch( Exception e ) {
        ;       //...ignore errors for simplicity
    }
    return inputMessage;
  }

  /* Get the output message wrapper
   */
  public MessageWrapper getOutputMessage() {
    // Retrieve the message from the bean. If there is no message, create
    // one if the task has been claimed by the user. Ensure that only
    // potential owners or owners can manipulate the output message.
    try{
        outputMessage = taskBean.getOutputMessageWrapper();
        if( outputMessage == null
         && taskBean.getState() == TaskInstanceBean.STATE_CLAIMED ) {
            HumanTaskManagerService htm = getHumanTaskManagerService();
            outputMessage = new MessageWrapperImpl();
            outputMessage.setMessage(
               htm.createOutputMessage( taskBean.getID() ).getObject()
            );
         }
    }
    catch( Exception e ) {
        ;       //...ignore errors for simplicity
    }
    return outputMessage
  }
}
```

MyHandler 管理対象 Bean は、リスト・ハンドラーへの項目リスナーとして登録できるように、com.ibm.jsf.handler.ItemListener インターフェースをインプリメントします。ユーザーがリスト内の項目をクリックすると、選択された項目について MyHandler Bean が itemChanged(Object item) メソッドで通知されます。ハンドラーは、項目タイプを検査してから、関連した TaskInstanceBean オブジェクトへの参照を保管します。このインターフェースを使用するには、faces-config.xml ファイル内の適切なリスト・ハンドラーの itemListener リストにエントリーを追加します。

MyHandler Bean は、getInputMessage および getOutputMessage メソッドを提供します。これらのメソッドはどちらも、MessageWrapper オブジェクトを戻します。メソッドは、参照されたタスク・インスタンス Bean への呼び出しを委任します。例えばメッセージが設定されていないなどの理由で、タスク・インスタンス Bean がヌルを戻した場合、ハンドラーは新規に空のメッセージを作成して保管します。Message コンポーネントは MyHandler Bean が提供するメッセージを表示します。

## タスクの結果

これで、JSF アプリケーションは、データ・オブジェクトおよびプリミティブ型をレンダリング可能な JavaServer ページを含むようになります。

## Message コンポーネント: タグ定義

Business Process Choreographer Explorer Message コンポーネントは、JavaServer Faces (JSF) アプリケーション内で、commonj.sdo.DataObject オブジェクトと、整数およびストリングなどのプリミティブ型をレンダリングします。

Message コンポーネントは、JSF コンポーネント・タグである bpe:form から構成されます。

### コンポーネント・クラス

com.ibm.bpe.jsf.component.MessageComponent

### 構文例

```
<bpe:form model="#{TaskInstanceDetailsBean.inputMessageWrapper}"
          simplification="true" readOnly="true"
          styleClass4table="messageData"
          styleClass4output="messageDataOutput">
</bpe:form>
```

### タグ属性

*表 24. bpe:form 属性*

| 属性 | 必須 | 説明 |
|---|---|---|
| id | いいえ | コンポーネントの JavaServer Faces ID。 |
| model | はい | commonj.sdo.DataObject オブジェクトまたは com.ibm.bpc.clientcore.MessageWrapper オブジェクトを参照する値バインディング式。 |
| readOnly | いいえ | この属性が true に設定されている場合は、読み取り専用フォームのみがレンダリングされます。デフォルトでは、この属性は false に設定されています。 |
| simplification | いいえ | この属性が true に設定されている場合は、単純型が含まれているプロパティーおよびカーディナリティーがゼロまたは 1 のプロパティーが表示されます。デフォルトでは、この属性は true に設定されています。 |
| style4validinput | いいえ | 有効な入力のレンダリング用のカスケーディング・スタイル・シート (CSS) スタイル。 |
| style4invalidinput | いいえ | 無効な入力のレンダリング用の CSS スタイル。 |
| styleClass4invalidInput | いいえ | 無効な入力のレンダリング用の CSS スタイル・クラス名。 |
| styleClass4output | いいえ | 出力エレメントのレンダリング用の CSS スタイル・クラス名。 |

*表 24. bpe:form 属性 (続き)*

| 属性 | 必須 | 説明 |
|---|---|---|
| styleClass4table | いいえ | Message コンポーネントによって提供されるテーブルのレンダリング用の、CSS テーブル・スタイルのクラス名。 |
| styleClass4validInput | いいえ | 有効な入力のレンダリング用の CSS スタイル・クラス名。 |

# タスクおよびプロセス・メッセージ用の JSP ページの開発

Business Process Choreographer Explorer インターフェースには、ビジネス・データの表示や入力のためのデフォルトの入力フォームおよび出力フォームが用意されています。 JSP ページを使用して、カスタマイズされた入力フォームおよび出力フォームをカスタマイズできます。

## このタスクについて

ユーザー定義の JavaServer Pages (JSP) ページを Web クライアントに組み込むには、WebSphere Integration Developer でヒューマン・タスクをモデル化するときにこれらのページを指定する必要があります。例えば、JSP ページの指定先は、特定のタスクやその入出力メッセージ、特定のユーザーのロールまたはすべてのユーザーのロールのいずれでも構いません。ユーザー定義 JSP ページは、出力データを表示して入力データを収集するために実行時にユーザー・インターフェースに組み込まれます。

カスタマイズ・フォームは自己完結した Web ページではなく、Business Process Choreographer Explorer によって HTML フォームに組み込まれる HTML フラグメントです。例えば、メッセージのすべてのラベルや入力フィールドのフラグメントがこれに該当します。

カスタマイズ・フォームがあるページでボタンをクリックすると、入力データは Business Process Choreographer Explorer に送信されて検証されます。検証は、提供されたプロパティーのタイプとブラウザーで使用されているロケールに基づいて行われます。 入力データを検証できない場合は同じページがもう一度表示され、検証エラーの情報が messageValidationErrors 要求属性に書き込まれます。情報はマップとして提供され、このマップは、無効なプロパティーの XML Path Expression (XPath) を、発生した妥当性検査例外にマップします。

カスタマイズ・フォームを Business Process Choreographer Explorer に追加するには、WebSphere Integration Developer を使用して以下のステップを実行します。

**手順**

1. カスタマイズ・フォームを作成します。

   Web インターフェースで使用される、入出力フォーム用のユーザー定義 JSP ページは、メッセージ・データにアクセスする必要があります。 JSP 内の Java 断片または JSP 実行言語を使用して、メッセージ・データにアクセスします。フォーム内のデータは要求コンテキストを介して使用可能です。
2. JSP ページにタスクを割り当てます。

ヒューマン・タスク・エディターでヒューマン・タスクを開きます。 クライアント設定で、ユーザー定義 JSP ページの場所と、カスタマイズ・フォームの適用先のロール (例: 管理者) を指定します。Business Process Choreographer Explorer のクライアント設定は、タスク・テンプレートに格納されます。これらの設定は、実行時にタスク・テンプレートを使用して取得されます。

3. ユーザー定義 JSP ページを Web アーカイブ (WAR ファイル) にパッケージ化します。

   WAR ファイルは、タスクが格納されているモジュールと一緒にエンタープライズ・アーカイブに組み込むことも、個別に配置することもできます。JSP が個別にデプロイされる場合、Business Process Choreographer Explorer またはカスタム・クライアントがデプロイされるサーバー上で JSP を使用可能にしてください。

   プロセスおよびタスク・メッセージにカスタム JSP を使用している場合、JSP をデプロイするために使用される Web モジュールを、カスタム JSF クライアントがマップされるのと同じサーバーにマップする必要があります。

### タスクの結果

カスタマイズ・フォームは、Business Process Choreographer Explorer で実行時にレンダリングされます。

## ユーザー定義 JSP フラグメント

ユーザー定義 JavaServer Pages (JSP) フラグメントは、HTML フォーム・タグに埋め込まれます。Business Process Choreographer Explorer は、実行時にこれらのフラグメントを、レンダリングされるページに埋め込みます。

入力メッセージのユーザー定義 JSP フラグメントは、出力メッセージの JSP フラグメントより先に埋め込まれます。

```
<html....>
   ...
   <form...>
     Input JSP (display task input message)

     Output JSP (display task output message)

   </form>
   ...
</html>
```

ユーザー定義 JSP フラグメントは、HTML フォーム・タグに埋め込まれているため、入力要素を追加できます。 入力要素の名前は、データ要素の XML Path Language (XPath) 表現と一致する必要があります。入力要素の名前には、以下に示す提供されたプレフィックス値を使用してプレフィックスを付けることが重要です。

```
<input id="address"
       type="text"
       name="${prefix}/selectPromotionalGiftResponse/address"
       value="${messageMap['/selectPromotionalGiftResponse/address"]}
       size="60"
       align="left" />
```

プレフィックス値は要求属性として提供されます。この属性により、引用符で囲まれた書式内の入力名は固有であることが保証されます。プレフィックスは Business Process Choreographer Explorer によって次のように生成されるため、変更しないでください。

```
String prefix = (String)request.getAttribute("prefix");
```

プレフィックス要素が設定されるのは、与えられたコンテキストにおいてメッセージが編集できる場合に限ります。 出力データは、ヒューマン・タスクの状態に応じてさまざまな方法で表示できます。例えば、タスクが要求状態にある場合は、出力データを変更できます。ただし、タスクが完了状態にある場合、出力データは表示専用になります。JSP フラグメントでは、プレフィックス要素が存在し、それに応じてメッセージを表示するかどうかをテストできます。 以下の JSTL ステートメントでは、プレフィックス要素が設定されているかどうかをテストする 1 つの方法を示しています。

```
...
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
...
<c:choose>
  <c:when test="${not empty prefix}">
    <!--Read/write mode-->
  </c:when>
  <c:otherwise>
    <!--Read-only mode-->
  </c:otherwise>
</c:choose>
```

# ヒューマン・タスク機能をカスタマイズするプラグインの作成

Business Process Choreographer では、ヒューマン・タスクの処理中に発生するイベントのイベント処理インフラストラクチャーを提供します。ニーズに合わせて機能を適応させることができるように、プラグイン点も設けられます。サービス・プロバイダー・インターフェース (SPI) を使用すると、イベントの処理および担当者照会結果の後処理を行うようにカスタマイズされたプラグインを作成できます。

## このタスクについて

ヒューマン・タスク API イベントとエスカレーション通知イベント用のプラグインを作成することができます。また、担当者解決から戻される結果を処理するプラグインを作成することもできます。例えば、ピーク時に結果リストにユーザーを追加して、ワークロードのバランスを取ることもできます。

プラグインを使用する前に、それらのプラグインをインストールして登録する必要があります。担当者照会結果を後処理するプラグインは TaskContainer アプリケーションに登録できます。これにより、プラグインはすべてのタスクで使用できるようになります。

## API イベント・ハンドラーの作成

API イベントは、API メソッドがヒューマン・タスクを操作するときに発生します。 API イベント・ハンドラー・プラグインのサービス・プロバイダー・インターフェース (SPI) を使用して、API イベントまたは同等の API イベントを持つ内部イベントが送信するタスク・イベントを処理するプラグインを作成します。

## このタスクについて

API イベント・ハンドラーを作成するには、次のステップを実行します。

**手順**

1. APIEventHandlerPlugin3 インターフェースをインプリメントするクラス、または APIEventHandler 実装クラスを拡張するクラスを作成します。 このクラスは、他のクラスのメソッドを呼び出すことができます。
   - APIEventHandlerPlugin3 インターフェースを使用する場合は、APIEventHandlerPlugin3 インターフェースおよび APIEventHandlerPlugin インターフェースのすべてのメソッドをインプリメントする必要があります。
   - APIEventHandler 実装クラスを拡張する場合は、必要なメソッドを上書きしてください。

   このクラスは、Java 2 Enterprise Edition (J2EE) Enterprise アプリケーションのコンテキストで実行します。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

   **注:** このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクを更新するメソッドは呼び出さないでください。このアクションを行うと、データベース内のタスク・データが矛盾するおそれがあります。
2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   次のいずれかの方法で、JAR ファイルを使用可能にすることができます。
   - アプリケーション EAR ファイル内のユーティリティー JAR ファイルとする。
   - アプリケーション EAR ファイルとともにインストールされる共用ライブラリーとする。
   - TaskContainer アプリケーションとともにインストールされる共用ライブラリーとする。この場合、プラグインはすべてのタスクで使用できるようになります。
3. JAR ファイルの META-INF/services/ ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

   この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java 2 サービス・プロバイダー・インターフェース仕様に準拠します。

   a. `com.ibm.task.spi.`*`plug-in_name`*`APIEventHandlerPlugin` という名前のファイルを作成します。 *plug-in_name* はプラグインの名前です。

      例えば、`Customer` という名前のプラグインが com.ibm.task.spi.APIEventHandlerPlugin3 インターフェースをインプリメントする場合、構成ファイルの名前は `com.ibm.task.spi.`**`Customer`**`APIEventHandlerPlugin` になります。

b. このファイルのコメント行 (番号記号 (#) で始まる行) とブランク行を除く最初の行に、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

   例えば、`MyAPIEventHandler` というプラグイン・クラスが `com.customer.plugins` パッケージにある場合は、構成ファイルの最初の行に `com.customer.plugins.MyAPIEventHandler` という項目が含まれていなければなりません。

## タスクの結果

ファイルには、API イベントを処理するプラグインを含むインストール可能な JAR ファイルと、プラグインのロードで使用できるサービス・プロバイダー構成ファイルがあります。

**注:** API イベント・ハンドラーおよび通知イベント・ハンドラーの両方登録するために使用可能な eventHandlerName プロパティーは 1 つだけです。API イベント・ハンドラーおよび通知イベント・ハンドラーを両方使用する場合、プラグイン実装は同じ名前でなければなりません (例えば、SPI 実装のイベント・ハンドラー名として `Customer` など)。

いずれのプラグインも、単一のクラスまたは 2 つのクラスを使用して実装できます。いずれの場合も、JAR ファイルの META-INF/services/ ディレクトリーに 2 つのファイルを作成する必要があります (com.ibm.task.spi.CustomerNotificationEventHandlerPlugin と com.ibm.task.spi.CustomerAPIEventHandlerPlugin など)。

プラグインの実装とヘルパー・クラスを単一の JAR ファイルにパッケージします。

実装への変更を有効にするには、共用ライブラリー内の JAR ファイルを置き換えて、関連 EAR ファイルを再デプロイし、サーバーを再始動します。

## 次のタスク

次に、プラグインをインストールして登録し、実行時にヒューマン・タスク・コンテナーが使用できるようにする必要があります。API イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

## API イベント・ハンドラー

API イベントは、ヒューマン・タスクが変更されるか、その状態が変化するときに発生します。これらの API イベントを処理するために、タスクが変更される前 (pre-event メソッド)、および API 呼び出しが戻る直前 (post-event メソッド) に、イベント・ハンドラーが直接呼び出されます。

pre-event メソッドが ApplicationVetoException 例外をスローすると、API アクションが実行されずに API の呼び出し元にその例外が戻され、イベントに関連付けられたトランザクションがロールバックされます。pre-event メソッドが内部イベントによってトリガーされ、ApplicationVetoException 例外がスローされた場合は、自動要求などの内部イベントが実行されなくても、例外はクライアント・アプリケーションに戻されません。この場合、情報メッセージが SystemOut.log ファイルに書き込

まれます。API メソッドが処理中に例外をスローすると、その例外はキャッチされ、post-event メソッドに渡されます。その例外は、post-event メソッドが戻った後に再び呼び出し元に渡されます。

以下の規則が、pre-event メソッドに適用されます。

- pre-event メソッドは、関連した API メソッドまたは内部イベントのパラメーターを受け取ります。
- pre-event メソッドは、ApplicationVetoException 例外をスローして処理の続行を阻止することができます。

以下の規則が、post-event メソッドに適用されます。

- post-event メソッドは、API 呼び出しに提供されたパラメーター、および戻り値を受け取ります。API メソッド実装によって例外がスローされると、post-event メソッドも例外を受け取ります。
- post-event メソッドは、戻り値を変更できません。
- post-event メソッドは例外をスローできません。ランタイム例外はログに記録されますが、無視されます。

API イベント・ハンドラーをインプリメントするには、APIEventHandlerPlugin インターフェースを拡張する APIEventHandlerPlugin3 インターフェースをインプリメントするか、またはデフォルトの com.ibm.task.spi.APIEventHandler SPI 実装クラスを拡張します。イベント・ハンドラーは、デフォルトの実装クラスから継承する場合、常に最新バージョンの SPI をインプリメントします。新しいバージョンの Business Process Choreographer にアップグレードする場合は、新しい SPI メソッドを利用すると必要な変更も少なくなります。

通知イベント・ハンドラーと API イベント・ハンドラーの両方がある場合、イベント・ハンドラー名は 1 つしか登録できないので、両方のハンドラーの名前は同じでなければなりません。

## 通知イベント・ハンドラーの作成

通知イベントは、ヒューマン・タスクがエスカレートされるときに生成されます。Business Process Choreographer には、エスカレーション作業項目の作成や電子メールの送信などのエスカレーションを処理する機能があります。通知イベント・ハンドラーを作成して、エスカレーションの処理方法をカスタマイズすることができます。

### このタスクについて

通知イベント・ハンドラーをインプリメントするには、NotificationEventHandlerPlugin インターフェースをインプリメントする方法と、デフォルトの com.ibm.task.spi.NotificationEventHandler サービス・プロバイダー・インターフェース (SPI) 実装クラスを拡張する方法があります。

通知イベント・ハンドラーを作成するには、次のステップを実行します。

**手順**

1. NotificationEventHandlerPlugin インターフェースをインプリメントするクラス、または NotificationEventHandler 実装クラスを拡張するクラスを作成します。 このクラスは、他のクラスのメソッドを呼び出すことができます。

   NotificationEventHandlerPlugin インターフェースを使用する場合は、すべてのインターフェース・メソッドをインプリメントする必要があります。SPI 実装クラスを拡張する場合は、必要なメソッドを上書きしてください。

   このクラスは、Java 2 Enterprise Edition (J2EE) Enterprise アプリケーションのコンテキストで実行します。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

   プラグインは、EscalationUser ロールの権限で呼び出されます。このロールは、ヒューマン・タスク・コンテナーの構成時に定義されます。

   **注:** このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクを更新するメソッドは呼び出さないでください。このアクションを行うと、データベース内のタスク・データが矛盾するおそれがあります。
2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   次のいずれかの方法で、JAR ファイルを使用可能にすることができます。
   - アプリケーション EAR ファイル内のユーティリティー JAR ファイルとする。
   - アプリケーション EAR ファイルとともにインストールされる共用ライブラリーとする。
   - TaskContainer アプリケーションとともにインストールされる共用ライブラリーとする。この場合、プラグインはすべてのタスクで使用できるようになります。
3. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   ヘルパー・クラスが数個の J2EE アプリケーションによって使用される場合は、共用ライブラリーとして登録する別の JAR ファイルにこれらのクラスをパッケージできます。
4. JAR ファイルの META-INF/services/ ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

   この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java 2 サービス・プロバイダー・インターフェース仕様に準拠します。

   a. `com.ibm.task.spi.`*`plug-in_name`*`NotificationEventHandlerPlugin` という名前のファイルを作成します。 *plug-in_name* はプラグインの名前です。

      例えば、`HelpDeskRequest` (イベント・ハンドラー名) という名前のプラグインが com.ibm.task.spi.NotificationEventHandlerPlugin インターフェースをイン

プリメントする場合、構成ファイルの名前は
com.ibm.task.spi.**HelpDeskRequest**NotificationEventHandlerPlugin になります。

b. このファイルのコメント行 (番号記号 (#) で始まる行) とブランク行を除く最初の行に、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

例えば、MyEventHandler というプラグイン・クラスが com.customer.plugins パッケージにある場合は、構成ファイルの最初の行に com.customer.plugins.MyEventHandler という項目が含まれていなければなりません。

### タスクの結果

ファイルには、通知イベントを処理するプラグインを含むインストール可能な JAR ファイルと、プラグインのロードで使用できるサービス・プロバイダー構成ファイルがあります。API イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

**注:** API イベント・ハンドラーおよび通知イベント・ハンドラーの両方登録するために使用可能な eventHandlerName プロパティーは 1 つだけです。API イベント・ハンドラーおよび通知イベント・ハンドラーを両方使用する場合、プラグイン実装は同じ名前でなければなりません (例えば、SPI 実装のイベント・ハンドラー名として Customer など)。

いずれのプラグインも、単一のクラスまたは 2 つのクラスを使用して実装できます。いずれの場合も、JAR ファイルの META-INF/services/ ディレクトリーに 2 つのファイルを作成する必要があります (com.ibm.task.spi.CustomerNotificationEventHandlerPlugin と com.ibm.task.spi.CustomerAPIEventHandlerPlugin など)。

プラグインの実装とヘルパー・クラスを単一の JAR ファイルにパッケージします。

実装への変更を有効にするには、共用ライブラリー内の JAR ファイルを置き換えて、関連 EAR ファイルを再デプロイし、サーバーを再始動します。

### 次のタスク

次に、プラグインをインストールして登録し、実行時にヒューマン・タスク・コンテナーが使用できるようにする必要があります。通知イベント・ハンドラーは、タスク・インスタンス、タスク・テンプレート、またはアプリケーション・コンポーネントに登録できます。

## API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインのインストール

API イベント・ハンドラーまたは通知イベント・ハンドラーのプラグインを使用するには、タスク・コンテナーからアクセスできるようにプラグインをインストールする必要があります。

### このタスクについて

プラグインのインストール方法は、そのプラグインが 1 つの Java 2 Enterprise Edition (J2EE) アプリケーションでのみ使用されるか、複数のアプリケーションで使用されるかによって異なります。

次のステップのいずれかを実行してプラグインをインストールします。

- 1 つの J2EE アプリケーションによって使用される場合のプラグインのインストール

  プラグインの JAR ファイルをアプリケーションの EAR ファイルに追加します。WebSphere Integration Developer のデプロイメント記述子エディターで、プラグインの JAR ファイルを、主要な Enterprise JavaBeans (EJB) モジュールの J2EE アプリケーション用のプロジェクト・ユーティリティー JAR ファイルとしてインストールします。
- 複数の J2EE アプリケーションによって使用される場合のプラグインのインストール

  JAR ファイルを WebSphere Application Server 共用ライブラリーに入れ、そのライブラリーを、プラグインへのアクセスが必要なアプリケーションと関連付けます。JAR ファイルをネットワーク・デプロイメント環境で使用できるようにするには、アプリケーションがデプロイされているサーバーまたはクラスター・メンバーをホストする各ノードに手動で JAR ファイルを配布します。アプリケーションのデプロイメント・ターゲット・スコープ (アプリケーションがデプロイされているサーバーまたはクラスター)、またはセル・スコープを使用できます。これによりプラグイン・クラスは、選択されたデプロイメント・スコープ全体で可視になることに注意してください。

### 次のタスク

これで、プラグインを登録できます。

## API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインをタスク・テンプレート、タスク・モデル、およびタスクに登録する

API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインをタスク、タスク・テンプレート、およびタスク・モデルに登録できます。この登録は、随時タスクの作成、既存タスクの更新、随時タスク・モデルの作成、またはタスク・テンプレートの定義のときに行うことができます。

### このタスクについて

API イベント・ハンドラーおよび通知イベント・ハンドラーのプラグインは、以下のレベルのタスクに登録できます。

**タスク・テンプレート**
: このテンプレートを使用して作成されるすべてのタスクは同じハンドラーを使用します

**随時タスク・モデル**
このモデルを使用して作成されるタスクは同じハンドラーを使用します

**随時タスク**
作成されたタスクは、指定されたハンドラーを使用します

**既存タスク**
タスクは、指定されたハンドラーを使用します

プラグインは、次のいずれかの方法で登録できます。

- WebSphere Integration Developer でモデル化されたタスク・テンプレートの場合は、タスク・モデル内にプラグインを指定します。
- 随時タスクまたは随時タスク・モデルの場合は、タスクまたはタスク・モデルの作成時にプラグインを指定します。

  TTask クラスの setEventHandlerName メソッドを使用して、イベント・ハンドラーの名前を登録します。
- 実行時に、タスク・インスタンスのイベント・ハンドラーを変更します。

  update(Task task) メソッドを使用して、実行時にタスク・インスタンスの別のイベント・ハンドラーを使用します。呼び出し元には、このプロパティーを更新するタスク管理者権限が必要です。

# 担当者照会結果の後処理を行うプラグインの作成、インストール、および実行

担当者解決は、特定のロール (例えばタスクの潜在的な所有者) に割り当てられているユーザーのリストを戻します。担当者解決で戻される担当者照会の結果を変更するプラグインを作成することができます。例えば、ワークロード・バランシングを改善するために、既にワークロードが高いユーザーを照会結果から除去するプラグインを使用することがあります。

## このタスクについて

後処理プラグインは 1 つしか所有できません。つまり、そのプラグインですべてのタスクの担当者照会結果を処理する必要があります。 このプラグインでは、ユーザーの追加または除去、あるいはユーザーまたはグループ情報の変更ができます。また、結果タイプを、例えばユーザー・リストからグループに、あるいは全員に、変更することもできます。

プラグインは担当者解決の完了後に実行されるので、機密性またはセキュリティーを保持するための規則が既に適用されています。プラグインは、担当者の解決中に除去されたユーザーについての情報を受け取ります (HTM_REMOVED_USERS マップ・キーで)。 プラグインでこのコンテキスト情報を確実に使用し、規定されている機密性規則またはセキュリティー規則が保持されるようにする必要があります。

担当者照会結果の後処理を実装するには、StaffQueryResultPostProcessorPlugin インターフェースを使用します。このインターフェースには、タスク、エスカレーション、タスク・テンプレート、およびアプリケーション・コンポーネントの照会結果を変更するメソッドが含まれています。

担当者照会結果の後処理を行うプラグインを作成するには、次のステップを実行します。

**手順**

1. StaffQueryResultPostProcessorPlugin インターフェースを実装するクラスを作成します。

   このクラスは、Java 2 Enterprise Edition (J2EE) Enterprise アプリケーションのコンテキストで実行します。このクラスは、他のクラスのメソッドを呼び出すことができます。このクラスとそのヘルパー・クラスが、EJB 仕様に従っていることを確認してください。

   **注:** このクラスから HumanTaskManagerService インターフェースを呼び出す場合は、イベントを作成したタスクを更新するメソッドは呼び出さないでください。このアクションを行うと、データベース内のタスク・データが矛盾するおそれがあります。

   インターフェースのメソッドをすべて実装する必要があります。これらのメソッドには、特定のタスク・テンプレート、タスク、またはエスカレーション・ロールの担当者割り当て基準に関連する情報が含まれています。

   - 担当者割り当て基準定義は、タイプ `Map` の **context** パラメーターのエントリーとして指定されます。この情報にアクセスするには、以下のようにします。

     ```
     Map pacAsMap = (Map) context.get("HTM_VERB");

     // PAC の名前を取得します
     String pacName = (String) pacAsMap.get("HTM_VERB_NAME");

     // PAC パラメーター名を取得します
     Set paramNames = pacAsMap.keySet();

     // 特定のパラメーターの値を取得します
     String paramValue = (String) pacAsMap.get(paramName);
     ```

   - 担当者割り当て基準パラメーターの値として指定された置換変数は、タイプ `Map` の **context** パラメーターのエントリーです。この情報にアクセスするには、以下のようにします。

     ```
     Object replVarObj = pacAsMap.get(replVarName);
     if (replVarObj instanceof String)
              String replVarValue = (String) replVarObj;
     if (replVarObj instanceof String[])
              String[] replVarValues = (String[]) replVarObj;
     ```

   - 担当者解決時に担当者ディレクトリーにアクセスして (例えば、virtual member manager の担当者ディレクトリーにアクセスして) 作成された StaffQueryResult オブジェクト。

     StaffQueryResult オブジェクトには、担当者解決時に取得されたユーザー・エントリーに関する情報が含まれています。詳しくは、StaffQueryResultPostProcessorPlugin インターフェースの Javadoc 参照情報を参照してください。

   - 担当者解決によって明示的に除外されたユーザーのリストは、タイプ `Map` の **context** パラメーターのエントリーとして組み込まれます。この情報にアクセスするには、以下のようにします。

     ```
     String[] removedUserIDs = (String[]) context.get("HTM_REMOVED_USERS");
     ```

次の例では、SpecialTask というタスクのエディター・ロールを変更する方法を示しています。

```
public StaffQueryResult processStaffQueryResult
                        (StaffQueryResult originalStaffQueryResult,
                                          Task task,
                                          int  role,
                                          Map  context)
{
  StaffQueryResult newStaffQueryResult = originalStaffQueryResult;
  StaffQueryResultFactory staffResultFactory =
                          StaffQueryResultFactory.newInstance();
  if (role == com.ibm.task.api.WorkItem.REASON_EDITOR &&
      task.getName() != null &&
      task.getName().equals("SpecialTask"))
  {
      UserData user = staffResultFactory.newUserData
                           ("SuperEditor",
                             new Locale("en-US"),
                            "SuperEditor@company.com");
      ArrayList userList = new ArrayList();
      userList.add(user);

      newStaffQueryResult = staffResultFactory.newStaffQueryResult(userList);
   }
   return(newStaffQueryResult);
}
```

2. プラグイン・クラスとそのヘルパー・クラスを JAR ファイルにアセンブルします。

   JAR を共用ライブラリーとして使用できるようにして、それをタスク・コンテナーに関連付けることができます。このようにすると、プラグインはすべてのタスクで使用できるようになります。

3. JAR ファイルの META-INF/services/ ディレクトリーに、プラグインのサービス・プロバイダー構成ファイルを作成します。

   この構成ファイルによって、プラグインを識別し、ロードするメカニズムが提供されます。このファイルは、Java 2 サービス・プロバイダー・インターフェース仕様に準拠します。

   a. com.ibm.task.spi.*plug-in_name*StaffQueryResultPostProcessorPlugin という名前のファイルを作成します。*plug-in_name* はプラグインの名前です。

      例えば、MyHandler という名前のプラグインが com.ibm.task.spi.StaffQueryResultPostProcessorPlugin インターフェースをインプリメントする場合、構成ファイルの名前は com.ibm.task.spi.**MyHandler**StaffQueryResultPostProcessorPlugin になります。

   b. このファイルのコメント行 (番号記号 (#) で始まる行) とブランク行を除く最初の行に、ステップ 1 で作成したプラグイン・クラスの完全修飾名を指定します。

      例えば、StaffPostProcessor というプラグイン・クラスが com.customer.plugins パッケージにある場合は、構成ファイルの最初の行に com.customer.plugins.StaffPostProcessor という項目が含まれていなけれ

ばなりません。 担当者照会結果を後処理するプラグインを含むインストール可能 JAR ファイルと、プラグインのロードに使用できるサービス・プロバイダー構成ファイルができます。

4. プラグインをインストールします。

   担当者照会結果の後処理プラグインは 1 つしか所有できません。プラグインは、共用ライブラリーとしてインストールする必要があります。

   a. このプラグインの WebSphere Application Server 共用ライブラリーを定義します。 Business Process Choreographer が構成されているサーバーまたはクラスターのスコープで共用ライブラリーを定義します。次に、この共用ライブラリーを TaskContainer アプリケーションと関連付けます。このステップは一度だけ実行する必要があります。

   b. サーバーまたはクラスター・メンバーをホストしていて、影響を受ける各 WebSphere Process Server インストール済み環境に対して、プラグイン JAR ファイルを使用可能にします。

5. プラグインを登録します。

   a. 管理コンソールで、Human Task Manager の「カスタム・プロパティー」ページに移動します。

      スタンドアロン環境の場合は、「**サーバー**」 → 「**アプリケーション・サーバー**」 → 「***server_name***」をクリックします。Business Process Choreographer がクラスターに構成されている場合は、「**サーバー**」 → 「**クラスター**」 → 「***cluster_name***」をクリックします。「**ビジネス・インテグレーション**」の下の「**Human Task Manager**」を選択します。「**追加プロパティー**」の下の「**カスタム・プロパティー**」を選択します。

   b. **Staff.PostProcessorPlugin** という名前と、プラグインにつけた名前の値 (この例では、`MyHandler`) を持つカスタム・プロパティーを追加します。

   これで、プラグインは担当者照会結果の後処理に使用できるようになりました。JAR ファイルを変更する場合は、共用ライブラリー内のファイルを置き換えて、サーバーを再始動してください。

6. プラグインを実行します。 後処理プラグインは、担当者の割り当てと担当者の代替の両方を実行した後に呼び出されます。このプラグインは、StaffQueryResultPostProcessorPlugin インターフェースで指定した情報を使用して呼び出されます。

# 第 2 部 アプリケーションのデプロイ

© Copyright IBM Corp. 2005, 2008

# 第 14 章 モジュールの準備とインストールの概要

モジュールのインストール (デプロイとも呼ばれる) では、モジュールをテスト環境または実稼働環境のいずれかで活動化します。この概要では、テストおよび実稼働環境と、モジュールのインストールに必要な手順の一部について簡単に説明します。

**注:** アプリケーションを実稼働環境でインストールするプロセスは、WebSphere Application Server Network Deployment バージョン 6 インフォメーション・センターの『アプリケーションの開発とデプロイ』の項で説明されているプロセスと同様です。これらのトピックをお読みになったことがない場合、まず目を通してください。

モジュールを実稼働環境にインストールする前に、必ずテスト環境での変更内容を確認してください。モジュールをテスト環境にインストールする場合は、WebSphere Integration Developer を使用します。詳しくは、WebSphere Integration Developer インフォメーション・センターを参照してください。モジュールを実稼働環境にインストールする場合は、WebSphere Process Server を使用します。

このトピックでは、モジュールの実稼働環境へのインストールおよびその準備に必要な概念と作業について説明します。モジュールで使用されるオブジェクトを収容するファイルや、モジュールをテスト環境から実稼働環境へ移行する方法について説明しているその他のトピックがあります。モジュールを正しくインストールするためには、これらのファイルとファイルに格納されている内容を理解することが大切です。

## ライブラリーと JAR ファイルの概要

モジュールでは、ライブラリー内の成果物を使用することがよくあります。成果物およびライブラリーは、モジュールをデプロイするときに指定する Java アーカイブ (JAR) ファイルに含まれています。

モジュールの開発時に、そのモジュールのさまざまな部分で使用する特定のリソースまたはコンポーネントを指定する場合があります。これらのリソースまたはコンポーネントは、モジュールの開発時に作成したオブジェクトである場合と、既にサーバー上にデプロイされているライブラリー内のオブジェクトである場合があります。ここでは、アプリケーションのインストール時に必要となるライブラリーおよびファイルについて説明します。

### ライブラリーの概要

ライブラリーには、WebSphere Integration Developer 内の複数のモジュールで使用されるオブジェクトおよびリソースが格納されています。これらの成果物は、JAR ファイル、リソース・アーカイブ (RAR) ファイル、または Web サービス・アーカイブ (WAR) ファイルに格納されています。これらの成果物の例としては、次のものがあります。

- インターフェースまたは Web サービス記述子 (拡張子 .wsdl のファイル)

© Copyright IBM Corp. 2005, 2008

- ビジネス・オブジェクトの XML スキーマ定義 (拡張子 .xsd のファイル)
- ビジネス・オブジェクト・マップ (拡張子 .map のファイル)
- リレーションシップ定義とロール定義 (拡張子 .rel および .rol のファイル)

ある成果物がモジュールで必要になると、EAR クラス・パスに基づいてサーバーがこれを探し出し、メモリーにまだロードされていない場合は、ロードします。それ以降は、成果物が置き換えられないかぎりは、その成果物に対する要求では、メモリーにロードしたコピーが使用されます。図 23 に、アプリケーションとコンポーネントおよび関連するライブラリーの包含関係を示します。

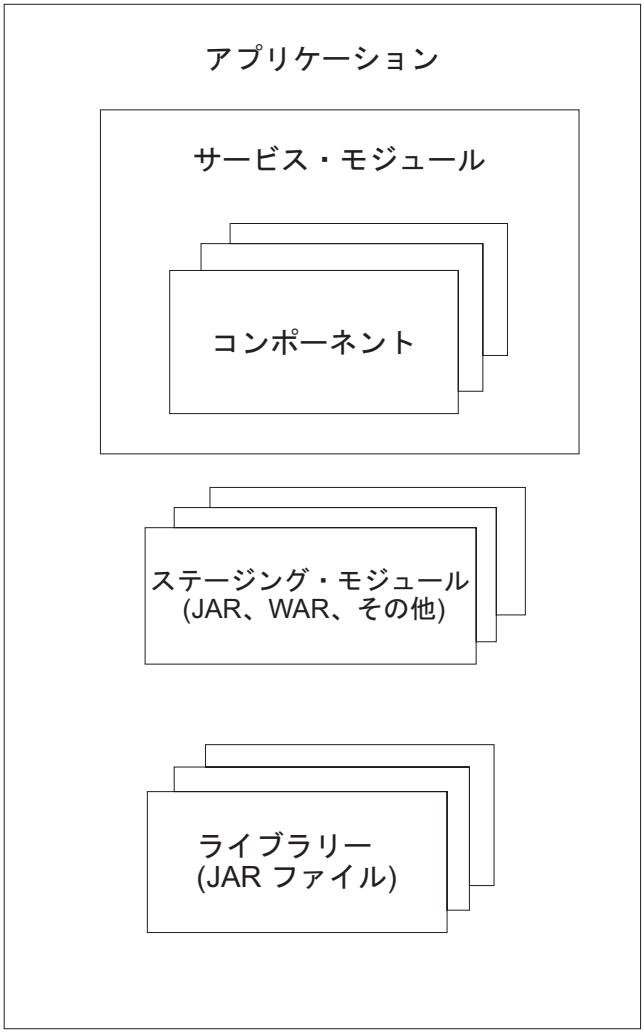



*図 23. モジュール、コンポーネント、およびライブラリー間の関係*

## JAR、RAR、および WAR ファイルの概要

モジュールのコンポーネントを格納できるファイルは複数存在します。これらのファイルについては、Java プラットフォーム、Enterprise Edition 仕様に詳しい説明があります。 JAR ファイルの詳細については、JAR 仕様に説明があります。

WebSphere Process Server では、JAR ファイルにアプリケーション (モジュールで使用するほかのサービス・コンポーネントへの支援的な参照およびインターフェー

スをすべて含んだ、モジュールのアセンブル・バージョン) も格納されています。アプリケーションを完全にインストールするには、この JAR ファイル、その他のすべてのライブラリー (JAR ファイル、Web サービス・アーカイブ (WAR) ファイル、リソース・アーカイブ (RAR) ファイル、ステージング・ライブラリー (Enterprise Java Beans - EJB) JAR ファイル、またはその他のアーカイブなど) が必要です。serviceDeploy コマンドを使用して、インストール可能な EAR ファイルを作成します。

### ステージング・モジュールの命名規則

ライブラリー内では、ステージング・モジュールの名前についての要件があります。ステージング・モジュールの名前は、それぞれのモジュールで固有でなければなりません。アプリケーションをデプロイするために必要なその他のモジュールには、ステージング・モジュールの名前との間に競合が発生しないような名前を付けてください。*myService* という名前のモジュールの場合、ステージング・モジュール名は、次のようになります。

- *myService*App
- *myService*EJB
- *myService*EJBClient
- *myService*Web

**注:** serviceDeploy コマンドは、サービスに WSDL ポート・タイプ・サービスが含まれている場合にのみ、*myService* Web ステージング・モジュールを作成します。

### ライブラリー使用時の考慮事項

ライブラリーを使用することにより、ビジネス・オブジェクトの整合性およびモジュール間での処理の整合性が保証されます。なぜなら、それぞれの呼び出し側モジュールは、特定のコンポーネントの専用コピーを所有するからです。不整合や障害が発生しないようにするために、呼び出し側モジュールで使用するコンポーネントおよびビジネス・オブジェクトに対する変更は、すべての呼び出し側モジュール間で整合がとれている必要があります。呼び出し側モジュールを更新するには、次の手順を実行します。

1. モジュールおよびライブラリーの最新コピーを実動サーバーにコピーします。
2. serviceDeploy コマンドを使用して、インストール可能な EAR ファイルを再作成します。
3. 呼び出し側モジュールを含む実行中のアプリケーションを停止し、再インストールします。
4. 呼び出し側モジュールを含むアプリケーションを再始動します。

## EAR ファイルの概要

EAR ファイルは、サービス・アプリケーションの実動サーバーへのデプロイにおいて、重要な要素です。

エンタープライズ・アーカイブ (EAR) ファイルとは、アプリケーションがデプロイメントに必要とするライブラリー、エンタープライズ Bean、および JAR ファイルを格納した圧縮ファイルです。

アプリケーション・モジュールを WebSphere Integration Developer からエクスポートするときに、JAR ファイルを作成します。この JAR ファイルおよびその他の成果物ライブラリーまたはオブジェクトを、インストール・プロセスの入力として使用します。 serviceDeploy コマンドは、アプリケーションを構成する、コンポーネントの記述と Java コードを含む入力ファイルから EAR ファイルを作成します。

## サーバーへのデプロイの準備

モジュールの開発およびテストが終了したら、モジュールをテスト・システムからエクスポートし、実稼働環境にデプロイする必要があります。アプリケーションをインストールするには、モジュールのエクスポート時に必要となるパスと、モジュールが必要とするライブラリーを認識している必要があります。

### 始める前に

このタスクを開始する前に、テスト・サーバーでモジュールの開発およびテストを完了し、パフォーマンスの問題などの問題点を解決しておく必要があります。

**重要:** デプロイメント環境で既に稼働中のアプリケーションまたはモジュールの置換を防ぐために、モジュールまたはアプリケーションの名前が既にインストール済みのものとは異なる固有の名前であることを確認してください。

### このタスクについて

このタスクでは、アプリケーションの必要な部分がすべて使用可能かどうか、および実動サーバーにデプロイできるように正しいファイルにパッケージされているかどうかを検証します。

**注:** WebSphere Integration Developer からエンタープライズ・アーカイブ (EAR) ファイルをエクスポートし、そのファイルを直接 WebSphere Process Server にインストールすることもできます。

**重要:** コンポーネント内部のサービスがデータベースを使用する場合は、データベースに直接接続されたサーバーにアプリケーションをインストールします。

**手順**

1. デプロイするモジュール用のコンポーネントを含むフォルダーを探します。

   コンポーネント・フォルダーの名前は *module-name* で、その中に *module*.module という名前のファイル (基本モジュール) があります。
2. モジュールに含まれるすべてのコンポーネントが、モジュール・フォルダーの下のコンポーネント・サブフォルダー内にあることを確認します。

   使いやすくするために、サブフォルダーには *module/component* のような名前を付けます。
3. 各コンポーネントを構成するすべてのファイルが適切なコンポーネント・サブフォルダーに格納されていて、*component-file-name*.component のような名前が付けられていることを確認します。

   コンポーネント・ファイルには、モジュール内の個々のコンポーネントの定義が記述されています。

4. 他のすべてのコンポーネントおよび成果物が、それらを必要とするコンポーネントのサブフォルダーに格納されていることを確認します。

   このステップで、コンポーネントが必要とする成果物への参照がすべて使用可能であることを確認します。serviceDeploy コマンドがステージング・モジュールに対して使用する名前と、これらのコンポーネントの名前が競合しないようにしてください。『ステージング・モジュールの命名規則』を参照してください。

5. 参照ファイル (*module*.references) がステップ 1 (364 ページ) のモジュール・フォルダー内に存在することを確認します。

   参照ファイルは、モジュール内の参照およびインターフェースを定義します。

6. ワイヤー・ファイル (*module*.wires) がコンポーネント・フォルダー内に存在することを確認します。

   ワイヤー・ファイルは、モジュール内の参照とインターフェース間の接続を確立します。

7. マニフェスト・ファイル (*module*.manifest) がコンポーネント・フォルダー内に存在することを確認します。

   マニフェストは、モジュールおよびモジュールを構成するすべてのコンポーネントをリストします。マニフェストには、クラスパス・ステートメントも記述されています。これは、serviceDeploy コマンドが、モジュールが必要とする他のモジュールを検出できるようにするためです。

8. モジュールの圧縮ファイルまたは JAR ファイルを作成します。このファイルは、実動サーバーにインストールするモジュールを準備するために使用する serviceDeploy コマンドへの入力として使用します。

## デプロイメント前の MyValue モジュールのフォルダー構造の例

以下の例は、MyValue、CustomerInfo、および StockQuote というコンポーネントから構成される MyValueModule モジュールのディレクトリー構造を示しています。

```
MyValueModule
   MyValueModule.manifest
   MyValueModule.references
   MyValueModule.wiring
   MyValueClient.jsp
process/myvalue
   MyValue.component
   MyValue.java
   MyValueImpl.java
service/customerinfo
   CustomerInfo.component
   CustomerInfo.java
   Customer.java
   CustomerInfoImpl.java
service/stockquote
   StockQuote.component
   StockQuote.java
   StockQuoteAsynch.java
   StockQuoteCallback.java
   StockQuoteImpl.java
```

### 次のタスク

実動サーバーへのモジュールのインストールの説明に従って、モジュールを実動システムにインストールします。

## クラスター上のサービス・アプリケーションのインストールに関する考慮事項

クラスターにサービス・アプリケーションをインストールする場合、追加要件が発生します。クラスターにサービス・アプリケーションをインストールする際に、これらの考慮事項に注意することが重要です。

クラスターは、スケール・メリットを提供して、サーバー間の要求ワークロードのバランスを取り、アプリケーションのクライアントに対して一定レベルの可用性を提供できるようにすることで、処理環境に多くのメリットをもたらす可能性があります。クラスター上で、サービスを含むアプリケーションをインストールする前に、以下の事項を検討してください。

- アプリケーションのユーザーが、クラスタリングにより提供される処理能力と可用性を必要としているか。

  その場合、クラスタリングが正しい解決策です。クラスタリングにより、アプリケーションの可用性と能力が向上します。
- クラスターがサービス・アプリケーション用に正しく準備されているか。

  サービスを含む最初のアプリケーションをインストールして開始する前に、クラスターを正しく構成する必要があります。クラスターを正しく構成していない場合、アプリケーションは要求を正しく処理できません。
- クラスターのバックアップはあるか。

  バックアップ・クラスターにもアプリケーションをインストールする必要があります。

# 第 15 章 ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール

ビジネス・プロセスまたはヒューマン・タスク、あるいはこの両方を含む Service Component Architecture (SCA) モジュールをデプロイメント・ターゲットに配布することができます。サーバーまたはクラスターをデプロイメント・ターゲットとすることができます。

### 始める前に

アプリケーションをインストールするアプリケーション・サーバーまたはクラスターごとに、Business Flow Manager と Human Task Manager がインストールされ、構成されていることを確認します。

### このタスクについて

ビジネス・プロセスおよびタスク・アプリケーションを、管理コンソールやコマンド行から、または管理スクリプトを実行してインストールすることができます。

### タスクの結果

ビジネス・プロセス・アプリケーションまたはヒューマン・タスク・アプリケーションがインストールされると、すべてのビジネス・プロセス・テンプレートおよびヒューマン・タスク・テンプレートは開始状態になります。これらのテンプレートから、プロセス・インスタンスとタスク・インスタンスを作成できます。

### 次のタスク

プロセス・インスタンスまたはタスク・インスタンスを作成するには、アプリケーションを始動する必要があります。

## Network Deployment 環境へのビジネス・プロセスおよびヒューマン・タスク・アプリケーションのインストール方法

プロセス・テンプレートまたはヒューマン・タスク・テンプレートを Network Deployment 環境にインストールするときに、アプリケーションのインストール機能により以下のアクションが自動的に実行されます。

アプリケーションは、段階的にインストールされます。次の段階を開始する前に、前の段階が正常に完了している必要があります。

1. アプリケーションのインストールは Deployment Manager で開始されます。

   この段階では、ビジネス・プロセス・テンプレートとヒューマン・タスク・テンプレートが、WebSphere 構成リポジトリーで構成されます。アプリケーションの検証も行われます。エラーが発生した場合、エラーは System.out ファイルまたは System.err ファイルに報告されるか、あるいは Deployment Manager で FFDC エントリーとして報告されます。

© IBM Corporation 2005, 2008 © IBM 2005, 2008

2. アプリケーションのインストールはノード・エージェントで続行されます。

   この段階では、1 つのアプリケーション・サーバー・インスタンスでのアプリケーションのインストールが開始されます。このアプリケーション・サーバー・インスタンスは、デプロイメント・ターゲットであるか、またはその一部です。デプロイメント・ターゲットが、複数のクラスター・メンバーで構成されるクラスターの場合は、このクラスターのクラスター・メンバーからサーバー・インスタンスが任意に選択されます。この段階でエラーが発生した場合、エラーは SystemOut.log ファイルまたは SystemErr.log ファイルに報告されるか、あるいはノード・エージェントで FFDC エントリーとして報告されます。

3. サーバー・インスタンスでアプリケーションが実行されます。

   この段階では、プロセス・テンプレートとヒューマン・テンプレートがデプロイメント・ターゲットの Business Process Choreographer データベースに配置されます。エラーが発生した場合、System.out ファイルまたは SystemErr.log ファイルに報告されるか、あるいはこのサーバー・インスタンスで FFDC エントリーとして報告されます。

## ビジネス・プロセスとヒューマン・タスクのデプロイメント

WebSphere Integration Developer またはサービス・デプロイメントによりプロセスまたはタスクのデプロイメント・コードが生成されるときに、各プロセス・コンポーネントまたはタスク・コンポーネントが 1 つのセッション・エンタープライズ Bean にマップされます。すべてのデプロイメント・コードは、エンタープライズ・アプリケーション (EAR) ファイルにパッケージ化されます。また、エンタープライズ・アプリケーションのインストール中に、プロセスごとに、そのプロセスの Java コードを表す Java クラスが生成され、EAR ファイルに埋め込まれます。デプロイするモデルの新規バージョンごとに、新しいエンタープライズ・アプリケーションにパッケージ化する必要があります。

ビジネス・プロセスまたはヒューマン・タスクが含まれているエンタープライズ・アプリケーションをインストールすると、これらのビジネス・プロセスまたはヒューマン・タスクは、ビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートとして Business Process Choreographer データベースに格納されます。デフォルトでは、新しくインストールされたテンプレートは、開始済み状態となります。ただし、新しくインストールされたエンタープライズ・アプリケーションの場合は、停止状態になります。インストール済みのエンタープライズ・アプリケーションは、個々に開始したり停止したりすることができます。

異なるエンタープライズ・アプリケーションそれぞれに、多数のバージョンのプロセス・テンプレートやタスク・テンプレートをデプロイできます。新規エンタープライズ・アプリケーションのインストール時に、インストールされるテンプレートのバージョンが次のように決定されます。

- テンプレートの名前とターゲット名前空間がまだ存在していない場合は、新規テンプレートがインストールされます。
- テンプレート名とターゲット名前空間が、既存のテンプレートと同じであるが、有効開始日が異なる場合は、既存のテンプレートの新規バージョンがインストールされます。

**注:** テンプレート名は、ビジネス・プロセスまたはヒューマン・タスクではなく、コンポーネントの名前から派生します。

有効開始日を指定しない場合、日付は次のように決定されます。

- WebSphere Integration Developer を使用する場合、有効開始日はヒューマン・タスクまたはビジネス・プロセスがモデル化された日付になります。
- サービス・デプロイメントを使用する場合、有効開始日は、serviceDeploy コマンドが実行された日付になります。アプリケーションがインストールされた日付を有効開始日として取得するのは、コラボレーション・タスクだけです。

## ビジネス・プロセス・アプリケーションおよびヒューマン・タスク・アプリケーションの対話式インストール

アプリケーションは、wsadmin ツールと installInteractive スクリプトを使用して、実行時に対話式でインストールできます。このスクリプトにより、管理コンソールを使用してアプリケーションをインストールする場合には変更することのできない設定を変更できます。

### このタスクについて

ビジネス・プロセス・アプリケーションを対話式にインストールするには、次のステップを実行します。

**手順**

1. wsadmin ツールを開始します。

   *profile_root*/bin ディレクトリーで、`wsadmin` と入力します。
2. アプリケーションをインストールします。

   wsadmin コマンド行プロンプトで、次のコマンドを入力します。

   ```
   $AdminApp installInteractive application.ear
   ```

   ここで、*application.ear* は、ご使用のプロセス・アプリケーションを含むエンタープライズ・アーカイブ・ファイルの修飾名です。 一連のタスクにおいてプロンプトが出されるので、その時にアプリケーションの値を変更できます。
3. 構成変更を保管します。

   wsadmin コマンド行プロンプトで、次のコマンドを入力します。

   ```
   $AdminConfig save
   ```

   変更を保管し、マスター構成リポジトリーへの更新を転送する必要があります。スクリプト・プロセスが終了し、変更を保存していなければ、変更は廃棄されます。

## プロセス・アプリケーションのデータ・ソースと設定参照の設定値の構成

特定のデータベース・インフラストラクチャーで SQL ステートメントを実行するプロセス・アプリケーションは、構成する必要があります。これらの SQL ステー

トメントは、情報サービス・アクティビティーから発生したり、プロセスのインストールまたはインスタンスの開始時に実行するステートメントであったりします。

## このタスクについて

このアプリケーションのインストール時には、次のタイプのデータ・ソースを指定できます。

- プロセス・インストール時に SQL ステートメントを実行するデータ・ソース
- プロセス・インスタンスの開始時に SQL ステートメントを実行するデータ・ソース
- SQL 断片アクティビティーを実行するデータ・ソース

SQL 断片アクティビティーを実行するために必要なデータ・ソースは、tDataSource タイプの BPEL 変数で定義されます。SQL 断片アクティビティーで必要とされるデータベース・スキーマ名とテーブル名は、tSetReference タイプの BPEL 変数で定義されます。これらの両方の変数の初期値は、構成することができます。

wsadmin ツールを使用してデータ・ソースを指定できます。

**手順**

1. wsadmin ツールを使用して、プロセス・アプリケーションを対話式にインストールします。
2. データ・ソースと設定参照を更新するタスクになるまで、タスクをステップスルーします。

   ご使用の環境でこれらの設定を構成します。次の例に、これらのタスクごとに変更可能な設定を示します。
3. 変更を保管します。

## 例: wsadmin ツールを使用したデータ・ソースと設定参照の更新

「**Updating data source**」タスクでは、プロセスのインストール時またはプロセスの開始時に使用される初期変数値およびステートメントのデータ・ソース値を変更できます。「**Updating set references**」タスクでは、データベース・スキーマとテーブル名に関連した設定を構成できます。

```
Task [24]: Updating data sources

//Change data source values for initial variable values at process start

Process name:  Test
// Name of the process template
Process start or installation time:  Process start
// Indicates whether the specified value is evaluated
//at process startup or process installation
Statement or variable:  Variable
// Indicates that a data source variable is to be changed
Data source name:  MyDataSource
// Name of the variable
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name to jdbc/newName
Task [25]: Updating set references

// Change set reference values that are used as initial values for BPEL variables
```

```
Process name:  Test
// Name of the process template
Variable:  SetRef
// The BPEL variable name
JNDI name:[jdbc/sample]:jdbc/newName
// Sets the JNDI name of the data source of the set reference to jdbc/newName
Schema name: [IISAMPLE]
// The name of the database schema
Schema prefix: []:
// The schema name prefix.
// This setting applies only if the schema name is generated.
Table name: [SETREFTAB]: NEWTABLE
// Sets the name of the database table to NEWTABLE
Table prefix: []:
// The table name prefix.
// This setting applies only if the prefix name is generated.
```

# 管理コンソールを使用した、ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのアンインストール

管理コンソールを使用すると、ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールできます。

## 始める前に

ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールするには、以下の前提条件を満たしている必要があります。

- アプリケーションがスタンドアロン・サーバーにインストールされている場合は、そのサーバーが稼働しており、Business Process Choreographer データベースへのアクセス権限を持っている必要があります。
- アプリケーションがクラスターにインストールされている場合は、デプロイメント・マネージャーおよび少なくとも 1 つのクラスター・メンバーが稼働している必要があります。クラスター・メンバーは、Business Process Choreographer データベースへのアクセス権限を所有している必要があります。
- アプリケーションが管理対象サーバーにインストールされている場合は、デプロイメント・マネージャーおよびこのサーバーが稼働している必要があります。このサーバーには Business Process Choreographer データベースへのアクセス権限が必要です。
- **-force** オプションを使用しない場合は、ビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートのどのような状態のインスタンスも存在しません。

## このタスクについて

ビジネス・プロセスまたはヒューマン・タスクが含まれるエンタープライズ・アプリケーションをアンインストールするには、以下のアクションを実行します。

### 手順

1. データベース、クラスターごとに少なくとも 1 つのアプリケーション・サーバー、アプリケーションがデプロイされているスタンドアロン・サーバーが動作していることを確認してください。

ネットワーク・デプロイメント環境では、デプロイメント・マネージャー、管理対象のすべてのスタンドアロン・アプリケーション・サーバー、および少なくとも 1 つのアプリケーション・サーバーが、そのアプリケーションがインストールされているクラスターごとに稼働している必要があります。

2. アプリケーションにビジネス・プロセス・インスタンスまたはヒューマン・タスク・インスタンスがないことを確認します。

   必要であれば、管理者は、Business Process Choreographer Explorer を使用して、残存するプロセス・インスタンスまたはタスク・インスタンスを削除することができます。プロセスおよびタスク・テンプレートは、停止する必要はありません。なぜなら、アプリケーションをアンインストールするときに自動的に停止するからです。

3. アプリケーションを停止してアンインストールするには、以下のようにします。
   a. 管理コンソールのナビゲーション・ペインで、「**アプリケーション**」 → 「**エンタープライズ・アプリケーション**」をクリックします。
   b. アンインストールするアプリケーションを選択し、「**停止**」をクリックします。

      アプリケーションでプロセス・インスタンスまたはタスク・インスタンスがまだ存在する場合は、このステップは失敗します。
   c. アンインストールするアプリケーションを再度選択し、「**アンインストール**」をクリックします。
   d. 「**保管**」をクリックして、変更を保管します。

### タスクの結果

アプリケーションはアンインストールされます。

## 管理コマンドを使用した、ビジネス・プロセスおよびヒューマン・タスク・アプリケーションのアンインストール

管理コマンドには、ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールするための、管理コンソールの代替方法が用意されています。

### 始める前に

ビジネス・プロセスまたはヒューマン・タスクを含むアプリケーションをアンインストールするには、以下の前提条件を満たしている必要があります。

- アプリケーションがスタンドアロン・サーバーにインストールされている場合は、そのサーバーが稼働しており、Business Process Choreographer データベースへのアクセス権限を持っている必要があります。
- アプリケーションがクラスターにインストールされている場合は、デプロイメント・マネージャーおよび少なくとも 1 つのクラスター・メンバーが稼働している必要があります。クラスター・メンバーは、Business Process Choreographer データベースへのアクセス権限を所有している必要があります。

- アプリケーションが管理対象サーバーにインストールされている場合は、デプロイメント・マネージャーおよびこのサーバーが稼働している必要があります。このサーバーには Business Process Choreographer データベースへのアクセス権限が必要です。
- **-force** オプションを使用しない場合は、ビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートのどのような状態のインスタンスも存在しません。

また、管理セキュリティーが使用可能の場合は、ユーザー ID に管理者権限またはオペレーター権限があることを確認します。 **-force** オプションを使用するには、管理者権限が必要です。

管理クライアントが接続しているサーバー・プロセスが動作していることを確認します。管理クライアントが自動的にサーバー・プロセスに接続できるよう、`-conntype NONE` オプションをコマンド・オプションとして使用しないでください。

## このタスクについて

次の手順で、bpcTemplates.jacl スクリプトを使用して、ビジネス・プロセス・テンプレートまたはヒューマン・タスク・テンプレートを含むアプリケーションをアンインストールする方法を示します。

アプリケーションをアンインストールする前に、Business Process Choreographer Explorer を使用するなどして、アプリケーション内のテンプレートに関連付けられたプロセス・インスタンスまたはタスク・インスタンスを削除できます。また、bpcTemplates.jacl スクリプトで **-force** オプションを使用し、テンプレートに関連付けられたインスタンスの削除、テンプレートの停止、およびそれらのテンプレートのアンインストールを 1 つのステップで実行できます。

**注意:**
**-force オプションはすべてのプロセス・インスタンスおよびタスク・インスタンスのデータを削除するため、注意して使用する必要があります。**

**手順**

1. Business Process Choreographer サンプル・ディレクトリーに移動します。

   Windows プラットフォームの場合は、以下を入力します。

   ```
   cd install_root¥ProcessChoreographer¥admin
   ```

   Linux、UNIX、および i5/OS プラットフォームの場合は、以下を入力します。

   ```
   cd install_root/ProcessChoreographer/admin
   ```

2. テンプレートを停止して、対応するアプリケーションをアンインストールします。

   Windows プラットフォームの場合は、以下を入力します。

   ```
   install_root¥bin¥wsadmin -f bpcTemplates.jacl
                               [-user user_name]
                               [-password user password]
                               -uninstall application_name
                               [-force]
   ```

Linux、UNIX、および i5/OS プラットフォームの場合は、以下を入力します。

```
install_root/bin/wsadmin -f bpcTemplates.jacl
                            [-user user_name]
                            [-password user password]
                            -uninstall application_name
                            [-force]
```

各部の意味は、次のとおりです。

*user_name*
管理セキュリティーが使用可能な場合は、認証用のユーザー ID を入力します。

*user_password*
管理セキュリティーが使用可能な場合は、認証用のユーザー・パスワードを入力します。

*application_name*
アンインストールするアプリケーションの名前を提供します。

## タスクの結果

アプリケーションはアンインストールされます。

# 第 16 章 アダプターおよびそのインストール

アダプターを使用すると、開発アプリケーションとエンタープライズ情報システム内のその他のコンポーネントとの通信ができるようになります。

アダプターのインストールに使用するプロセスについては、WebSphere Integration Developer インフォメーション・センターの『アダプターの構成および使用』に説明があります。

© Copyright IBM Corp. 2005, 2008

# 第 17 章 失敗したデプロイメントのトラブルシューティング

このトピックでは、アプリケーションのデプロイ時の問題の原因を判別するために行うステップについて説明します。また、参考になるいくつかのソリューションも示されています。

## 始める前に

このトピックは、以下の事項を前提としています。
- モジュールのデバッグの基本について理解している。
- モジュールのデプロイ中にロギングおよびトレースがアクティブになっている。

## このタスクについて

デプロイメントのトラブルシューティングのタスクは、エラーの通知を受け取った後に開始します。失敗したデプロイメントには、アクションをとる前に検査する必要のあるさまざまな症状があります。

**手順**

1. アプリケーションのインストールが失敗したかどうか判別します。

   SystemOut.log ファイルを調べて、失敗の原因を示すメッセージを探します。アプリケーションをインストールできない理由には、以下のようなものがあります。
   - 同一の Network Deployment セル内の複数のサーバーにアプリケーションをインストールしようとしている。
   - アプリケーションの名前が、アプリケーションをインストールする Network Deployment セル上の既存のモジュールの名前と同じである。
   - EAR ファイル内部の J2EE モジュールを異なるターゲット・サーバーにデプロイしようとしている。

   **重要:** インストールが失敗し、アプリケーションにサービスが含まれる場合、アプリケーションの再インストールを試みる前に、失敗の前に作成された SIBus 宛先または J2C アクティベーション・スペックを除去する必要があります。これらの成果物を除去する最も簡単な方法は、失敗後に「**保管**」>「**すべて廃棄 (Discard all)**」をクリックする方法です。不注意で変更を保存した場合、SIBus 宛先および J2C アクティベーション・スペックを手動で除去する必要があります (『管理』セクションの『SIBus 宛先の削除』および『J2C アクティベーション・スペックの削除』を参照)。
2. アプリケーションが正しくインストールされている場合は、アプリケーションが正常に開始したかどうかを確認します。

   アプリケーションが正常に開始していない場合は、サーバーがアプリケーションのリソースを初期化しようとしたときに障害が起きています。

   a. SystemOut.log ファイルを調べて、対処法を指示するメッセージを探します。

© Copyright IBM Corp. 2005, 2008

b. アプリケーションで必要なリソースが使用可能か、また、それらのリソースが正常に開始されたかどうかを確認します。

開始されないリソースがあると、アプリケーションは実行されません。これは、情報が失われるのを防ぐためです。リソースが開始しない理由には次のものがあります。
- 指定されたバインディングが正しくない。
- リソースが正しく構成されていない。
- リソースがリソース・アーカイブ (RAR) ファイルに含まれていない。
- Web リソースが Web サービス・アーカイブ (WAR) ファイルに含まれていない。

c. コンポーネントが欠落していないかどうか判別します。

コンポーネント欠落の原因は、エンタープライズ・アーカイブ (EAR) ファイルが正しく作成されなかったことにあります。モジュールが必要とするすべてのコンポーネントが、Java アーカイブ (JAR) ファイルをビルドするテスト・システムの正しいフォルダーにあることを確認してください。『サーバーへのデプロイの準備』で追加情報について説明します。

3. アプリケーションで情報が処理されているかどうかを調べます。

実行中のアプリケーションでも、情報の処理に失敗することがあります。この理由は、ステップ 2b で示した理由と同様です。

a. アプリケーションが、別のアプリケーションに含まれるサービスを使用するかどうかを判別します。 その別のアプリケーションがインストール済みで、正常に開始されていることを確認します。

b. 失敗したアプリケーションが使用する別のアプリケーションに含まれる、各種デバイス用のインポート・バインディングおよびエクスポート・バインディングが正しく構成されていることを確認します。 管理コンソールを使用して、バインディングを調べ、訂正してください。

4. 問題を解決してから、アプリケーションを再始動します。

## J2C アクティベーション・スペックの削除

サービスを含むアプリケーションをインストールすると、システムによって J2C アプリケーションの仕様が作成されます。アプリケーションを再インストールする前に、この仕様を削除する必要がある場合があります。

### 始める前に

アプリケーションのインストールに失敗したために仕様を削除する場合、Java Naming and Directory Interface (JNDI) 名の中のモジュールとインストールできなかったモジュールの名前とが一致するようにしてください。JNDI 名の 2 番目の部分が、宛先をインプリメントしたモジュールの名前に相当します。例えば、sca/SimpleBOCrsmA/ActivationSpec の場合、**SimpleBOCrsmA** がモジュール名です。

**このタスクで必要なセキュリティー・ロール:** セキュリティーとロール・ベースの許可が有効になっている場合、このタスクを実行するには、管理者またはコンフィギュレーターとしてログインする必要があります。

### このタスクについて

サービスを含むアプリケーションをインストールした後で間違って構成を保管したが、J2C アクティベーション・スペックが不要な場合は、その仕様を削除します。

**手順**

1. 削除するアクティベーション・スペックを見つけます。

   仕様は「リソース・アダプター」パネルに表示されます。「**リソース**」>「**リソース・アダプター**」をクリックして、このパネルにナビゲートします。

   a. 「**Platform Messaging Component SPI Resource Adapter**」を見つけます。

      このアダプターを見つけるには、スタンドアロン・サーバーの「**ノード**」スコープ、またはデプロイメント環境の「**サーバー**」スコープで作業する必要があります。

2. Platform Messaging Component SPI Resource Adapter に関連した J2C アクティベーション・スペックを表示します。

   リソース・アダプター名をクリックすると、次のパネルが表示され、関連した仕様が表示されます。

3. 削除するモジュール名に一致した「**JNDI 名**」の仕様をすべて削除します。

   a. 該当する仕様の横にあるチェック・ボックスをクリックします。

   b. 「**削除**」をクリックします。

### タスクの結果

システムは、選択された仕様を表示から削除します。

### 次のタスク

変更を保管します。

## SIBus 宛先の削除

SIBus 宛先とは、アプリケーションでサービスを使用可能にするための関連付けのことです。宛先の削除が必要になる場合があります。

### 始める前に

アプリケーションのインストールに失敗したために宛先を削除する場合、宛先名の中のモジュールとインストールできなかったモジュールの名前とが一致するようにしてください。宛先の 2 番目の部分が、宛先をインプリメントしたモジュールの名前に相当します。例えば、sca/SimpleBOCrsmA/component/test/sca/cros/simple/cust/Customer の場合、**SimpleBOCrsmA** がモジュール名です。

**このタスクで必要なセキュリティー・ロール:** セキュリティーとロール・ベースの許可が有効になっている場合、このタスクを実行するには、管理者またはコンフィギュレーターとしてログインする必要があります。

## このタスクについて

サービスを含むアプリケーションのインストール後に不注意で構成を保管した場合、または SIBus 宛先を必要としなくなった場合、その宛先を削除します。

**注:** このタスクは、SCA システム・バスからのみ宛先を削除します。サービスを含むアプリケーションを再インストールする前に、アプリケーション・バスからもその項目を削除する必要があります (このインフォメーション・センターの『管理』セクションの『J2C アクティベーション・スペックの削除』を参照してください)。

**手順**

1. 管理コンソールにログインします。
2. SCA システム・バスの宛先を表示します。

   「**サービス統合**」>「**バス**」をクリックして、パネルにナビゲートします。
3. SCA システム・バスの宛先を選択します。

   画面上で、「**SCA.SYSTEM.*cellname*.Bus**」をクリックします。ここで、*cellname* は、削除しようとしている宛先を持つモジュールが含まれているセルの名前です。
4. 削除するモジュールと一致するモジュール名を含む宛先を削除します。
   a. 該当する宛先の横にあるチェック・ボックスをクリックします。
   b. 「**削除**」をクリックします。

## タスクの結果

パネルには残りの宛先のみが表示されます。

## 次のタスク

これらの宛先を作成したモジュールに関連した J2C アクティベーション・スペックを削除します。

# 第 3 部 付録

© Copyright IBM Corp. 2005, 2008

# 特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

*〒106-8711*
*東京都港区六本木 3-2-12*
*日本アイ・ビー・エム株式会社*
*法務・知的財産*
*知的財産権ライセンス渉外*

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

© Copyright IBM Corp. 2005, 2008

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
1001 Hillsdale Blvd., Suite 400
Foster City, CA 94404
U.S.A.

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生した創作物にも、次のように、著作権表示を入れていただく必要があります。「(c) (お客様の会社名) (西暦年)」このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。 (C) Copyright IBM Corp. _年を入れる_. All rights reserved.

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

**プログラミング・インターフェース情報**

プログラミング・インターフェース情報がある場合、それらはこのプログラムを使用してアプリケーション・ソフトウェアを作成する際に役立つよう提供されています。

一般使用プログラミング・インターフェースにより、お客様はこのプログラム・ツール・サービスを含むアプリケーション・ソフトウェアを書くことができます。

ただし、この情報には、診断、修正、および調整情報が含まれている場合があります。診断、修正、調整情報は、お客様のアプリケーション・ソフトウェアのデバッグ支援のために提供されています。

**警告:** 診断、修正、調整情報は、変更される場合がありますので、プログラミング・インターフェースとしては使用しないでください。

**商標**

IBM、IBM ロゴ、および ibm.com は、International Business Machines Corporation の米国およびその他の国における商標または登録商標です。これらおよび他の IBM 商標に、この情報の最初に現れる個所で商標表示 (® または ™) が付されている場合、これらの表示は、この情報が公開された時点で、米国において、IBM が所有する登録商標またはコモン・ロー上の商標であることを示しています。このような商標は、その他の国においても登録商標またはコモン・ロー上の商標である可能性があります。現時点での IBM の商標リストについては、www.ibm.com/legal/copytrade.shtml の「Copyright and trademark information」をご覧ください。

Java は、Sun Microsystems, Inc. の米国およびその他の国における商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

この製品には、Eclipse Project (http://www.eclipse.org) により開発されたソフトウェアが含まれています。

Printed in Japan

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12