BEA WebLogic Integration™

BPM プラグイン プログラミング ガイド

リリース 7.0
マニュアルの日付 : 2002 年 6 月

BEA

BEA

著作権

限定的権利条項

商標または登録商標

BPM プラグイン プログラミング ガイド

パート番号	日付	ソフトウェアのバージョン
なし	2002 年 6 月	7.0

目次

このマニュアルの内容

1. BPM プラグイン開発の概要

2. プラグイン開発の基礎

3. プラグイン セッション EJB の定義

4. プラグイン コンポーネントの定義

5. プラグイン通知の使い方

6. プラグイン イベントの処理

このマニュアルの内容

このマニュアルでは、WebLogic Integration の Business Process Management (BPM) の機能を拡張するプラグイン アプリケーションの開発方法について説明します。

このマニュアルの内容は以下のとおりです。

- 第 1 章「BPM プラグイン開発の概要」では、プラグイン アプリケーションの開発について説明します。特に、この章では、BPM Plug-in Manager とプラグイン API の両方についてその概要を説明します。また、BPM がデプロイされたプラグインを認識する方法について説明し、プラグイン アプリケーション開発プロセスの主なタスクの概要を示し、このマニュアルで示されるコードの例の抜粋元のプラグイン サンプルについて説明します。
- 第 2 章「プラグイン開発の基礎」では、プラグイン開発に必要な基本タスクについて説明します。この章では、パッケージおよびインタフェースのインポート、Plug-in Manager との接続および切断、Plug-in Manager バージョンへのアクセス、プラグイン値オブジェクトの使用について、その方法を説明します。
- 第 3 章「プラグイン セッション EJB の定義」では、プラグイン セッション EJB を定義する方法について説明します。
- 第 4 章「プラグイン コンポーネントの定義」では、プラグイン コンポーネントを定義する方法について説明します。
- 第 5 章「プラグイン通知の使い方」では、プラグイン通知を使用する方法について説明します。
- 第 6 章「プラグイン イベントの処理」では、プラグイン イベントを処理する方法について説明します。
- 第 7 章「プラグインの管理」では、プラグインの表示、ロードおよびコンフィグレーションの方法について説明します。
- 第 8 章「プラグイン オンライン ヘルプの定義」では、プラグイン オンライン ヘルプを定義する方法について説明します。

- 第 9 章「プラグインのデプロイメント」では、プラグインをデプロイする方法について説明します。
- 第 10 章「BPM プラグイン サンプル」では、BPM に用意されているプラグイン サンプルについて詳しく説明します。
- 付録 A「プラグイン コンポーネント定義のロードマップ」では、各タイプのプラグイン コンポーネントを定義するために必要な手順の要約を示します。
- 付録 B「プラグイン値オブジェクトのまとめ」では、BPM プラグイン値オブジェクトとそのメソッドについて説明します。
- 付録 C「BPM グラフィカル ユーザ インタフェース スタイル シート」では、Java Swing クラスに基づいたカスタム プラグインの設計に役立つ情報について説明します。

対象読者

このマニュアルは、カスタム プラグイン アプリケーションの作成に関心のあるアプリケーション開発者を対象としています。読者が、WebLogic Integration 製品、Java プログラミングおよび XML に精通していることを前提としています。

e-docs Web サイト

BEA 製品のドキュメントは、BEA Systems, Inc. の Web サイトで入手できます。BEA のホーム ページで [製品のドキュメント] をクリックするか、または次の URL にある製品ドキュメント ページを直接表示してください。

http://edocs.beasys.co.jp/e-docs/index.html

このマニュアルの印刷方法

Web ブラウザの [ファイル | 印刷] オプションを使用することにより、Web ブラウザからこのマニュアルを一度に 1 ファイルずつ印刷できます。

このマニュアルの PDF 版は、e-docs Web サイトにある WebLogic Integration ドキュメントのホームページで入手できます。PDF を Adobe Acrobat Reader で開くと、マニュアルの全体（または一部分）を書籍の形式で印刷できます。PDF を表示するには、WebLogic Integration ドキュメントのホーム ページを開き、[PDF 版] ボタンをクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader がない場合は、次の URL にある Adobe の Web サイトから無料で入手できます。

http://www.adobe.co.jp/

関連情報

WebLogic Integration BPM クライアント アプリケーションの Studio および Worklist を使用するプログラマに役立つ情報が、以下の WebLogic Integration マニュアルに記載されています。これらのアプリケーションは WebLogic Integration API の BPM コンポーネントを使用して構築されています。

- *BPM クライアント アプリケーション プログラミング ガイド*
- *WebLogic Integration Studio ユーザーズ ガイド*
- *WebLogic Integration Worklist ユーザーズ ガイド*
- *WebLogic Integration BPM ユーザーズ ガイド*
- BEA WebLogic Integration API Javadoc

Java アプリケーションの概要については、次の Sun Microsystems 社の Java Web サイトを参照してください。

http://java.sun.com/

XML および XML パーサの概要については、次の O' Reilly & Associates 社の XML.com Web サイトを参照してください。

http://www.xml.com/

サポート情報

WebLogic Integration のドキュメントに関するユーザからのフィードバックは弊社にとって非常に重要です。ご質問や意見などがあれば、電子メールで **docsupport-jp@bea.com** までお送りください。寄せられたご意見については、WebLogic Integration のドキュメントを作成および改訂する BEA の専門の担当者が直接目を通します。

電子メールのメッセージには、ご使用の WebLogic Integration のリリースをお書き添えください。

本バージョンの WebLogic Integration について不明な点がある場合、または WebLogic Integration のインストールおよび動作に問題がある場合は、BEA WebSUPPORT (**http://websupport.bea.com/custsupp**) を通じて BEA カスタマ サポートまでお問い合わせください。カスタマ サポートへの連絡方法については、製品パッケージに同梱されているカスタマ サポート カードにも記載されています。

カスタマ サポートでは以下の情報をお尋ねしますので、お問い合わせの際はあらかじめご用意ください。

- お名前、電子メール アドレス、電話番号、ファクス番号
- 会社の名前と住所
- お使いの機種とコード番号
- 製品の名前とバージョン
- 問題の状況と表示されるエラー メッセージの内容

表記規則

このマニュアルでは、全体を通して以下の表記規則が使用されています。

表記法	摘要
太字	用語集で定義されている用語を示す。
〔Ctrl〕 + 〔Tab〕	複数のキーを同時に押すことを示す。
斜体	強調または書籍のタイトルを示す。

等幅テキスト	コード サンプル、コマンドとそのオプション、データ構造体とそのメンバー、データ型、ディレクトリ、およびファイル名とその拡張子を示す。等幅テキストはキーボードから入力するテキストも示す。
	例:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

太字の等幅テキスト	コード内の重要な箇所を示す。 *例*: `void` **`commit`** `( )`
斜体の等幅テキスト	コード内の変数を示す。 *例*: `String` *`expr`*

表記法	摘要
すべて大文字のテキスト	デバイス名、環境変数、および論理演算子を示す。 *例*: LPT1 SIGNON OR
{ }	構文の中で複数の選択肢を示す。実際には、この括弧は入力しない。
[ ]	構文の中で任意指定の項目を示す。実際には、この括弧は入力しない。 *例*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
|	構文の中で相互に排他的な選択肢を区切る。実際には、この記号は入力しない。
...	コマンドラインで以下のいずれかを示す。 ■ 引数を複数回繰り返すことができる ■ 任意指定の引数が省略されている ■ パラメータや値などの情報を追加入力できる 実際には、この省略記号は入力しない。 *例*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	コード サンプルまたは構文で項目が省略されていることを示す。 実際には、この省略記号は入力しない。

1　BPM プラグイン開発の概要

注意： WebLogic Integration は、BPM 機能を拡張するために利用できるプラグイン フレームワークをサポートします。このプラグイン フレームワークは、BPM の本来の機能を拡張する場合に限り使用します。BPM を外部アプリケーションやシステムと統合する目的で、このプラグイン フレームワークを使用しないでください。その場合には、ビジネス オペレーションとして BPM から呼び出される EJB または Application Integration (AI) アダプタのいずれかを使用します。

この章では、Business Process Management (BPM) プラグインとその開発の概要について説明します。この章の内容は以下のとおりです。

- プラグインとは
- BPM によりプラグイン開発をサポートする方法
- デプロイされたプラグインを BPM が検出する方法
- BPM プラグイン開発タスク
- BPM プラグイン サンプル

プラグインとは

*プラグイン*は、実行時にロード可能な Java クラスで構成され、Business Process Management (BPM) の機能と WebLogic Integration の機能を拡張します。

プラグインを使用することにより、次の BPM ワークフロー コンポーネントの設計または実行時動作を変更できます。

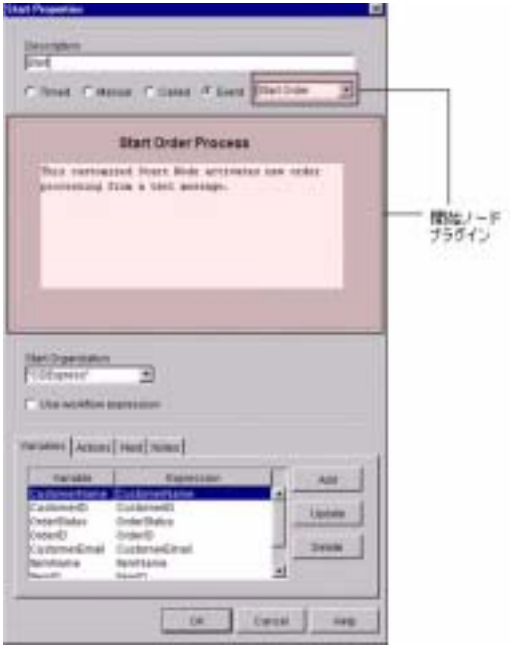
- 開始、イベント、および完了の各ノード
- タスク アクション
- ワークフロー テンプレートおよびテンプレート定義のプロパティ

- 関数
- メッセージ タイプ
- 変数タイプ

たとえば、WebLogic Integration Studio にある標準の 開始ノード トリガ メソッドを使用するよりも、電子メール メッセージなど、XML ではないイベントを送信することにより、ビジネス プロセスの実行をトリガしたほうがよい場合があります。これは、開始ノードの動作を拡張するプラグインを設計し、この新しい非 XML トリガに対するサポートを導入することにより実現できます。

次の図に、開始ノードの設計および実行時動作を変更するプラグインの例を示します。この図は、[開始のプロパティ] ダイアログ ボックスのプラグイン定義領域を示しています。

図 1-1 プラグイン の例 : 開始ノード



この例について説明します。

- [開始のプロパティ] ダイアログ ボックスでは、[イベント] プルダウン メニューの中に潜在的トリガとして [注文開始] イベントがリストされています。このイベントが選択されると、プラグインにより定義されたイベント情報がダイアログ ボックスの中心に表示されます。この動作は、開始ノードの *設計*がどのようにカスタマイズされたかを示しています。
- 実行時には、[注文開始] イベントによりワークフローの開始をトリガできます。この機能は、開始ノードの*実行時動作*がどのようにカスタマイズされたかを示しています。

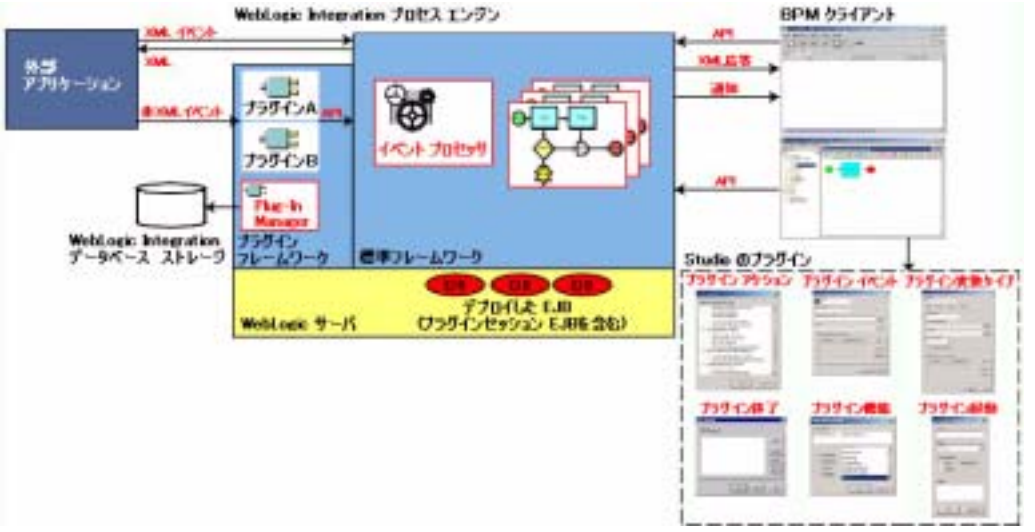
注意 : 前の図に示されている開始ノード プラグインの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

BPM によりプラグイン開発をサポートする方法

ビジネス プロセスの設計、実行、およびモニタのための標準フレームワーク以外に、WebLogic Integration は BPM 機能のための *プラグイン フレームワーク*をサポートしています。このフレームワークにより、既存のソフトウェアのカスタマイズするプラグインを作成し、他の製品や技術との強力でシームレスな統合を達成できます。

次の図に、BPM アーキテクチャ全体の中での BPM プラグイン フレームワークを示します。

図 1-2 BPM プラグイン フレームワーク



この図では次の点に注目してください。

- BEA WebLogic Server は、BPM およびプラグイン EJB のデプロイメントを管理します。

  WebLogic Server でプラグインをセッション EJB としてデプロイするだけで、BPM ユーザはプラグインを利用できるようになります。*WebLogic Integration プロセス エンジンや BPM クライアント上にはインストールする必要はありません。*

  プラグインは、`config.xml` ファイルを編集することにより、WLI アプリケーションの一部としてデプロイする必要があります。プラグインをデプロイする方法の詳細については、9-1 ページの「プラグインのデプロイメント」を参照してください。

- BPM Plug-in Manager は、WebLogic Integration データベースにプラグイン コンフィグレーション情報を格納します。

- 外部アプリケーションは、*XML イベント*または*非 XML イベント*を介してプロセス エンジンと会話します。非 XML イベントは、プラグイン フレームワークを介してサポートされます。

- Event Processor は、XML イベントと非 XML イベントの両方を管理します。プラグイン イベント処理の詳細については、6-1 ページの「プラグイン イベントの処理」を参照してください。
- プラグイン フレームワークは、次のメイン コンポーネントで構成されます。
  - プラグインの管理をサポートする Plug-in Manager
  - プラグインの設計と開発をサポートする Plug-in API

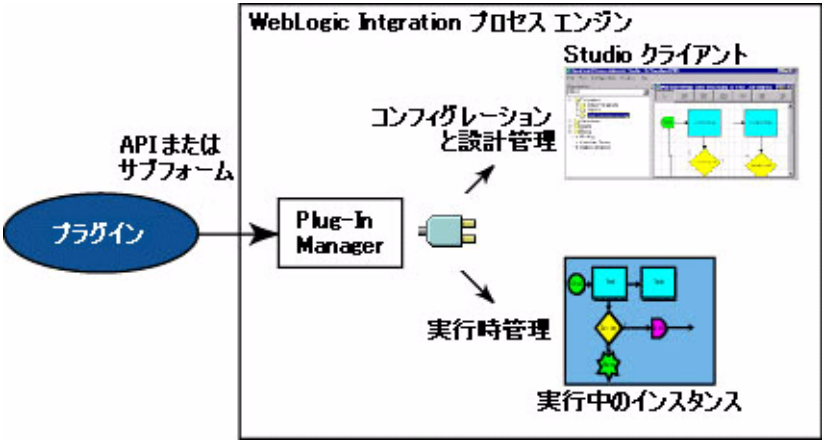
次の節で、この 2 つのコンポーネントについて説明します。

プロセス エンジンの役割の詳細については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「Business Process Management API による開発」の「WebLogic Integration プロセス エンジン」を参照してください。

Plug-In Manager

Plug-in Manager は、WebLogic Integration プロセス エンジンの専用パーツであり、次の図に示すように、プラグインのコンフィグレーション、設計および実行時管理をサポートします。

図 1-3 Plug-In Manager



Plug-in Manager は、ロードされたプラグイン、およびこれらのプラグインと WebLogic Integration プロセス エンジンや BPM クライアントとの対話を監視し、プラグイン関連のすべてのリクエストをルーティングします。

たとえば、Plug-in Manager で*コンフィグレーションと設計*を担当するサブセットには、次の機能があります。

- プラグイン固有のコンテンツの入ったワークフローのインポートおよびエクスポートをサポートします。
- 複数のプラグインの同時ロードを管理します。
- 一般的な式の文法を使用して、BPM 式評価子があらゆるフォーマットの着信プラグイン データを解釈できるようにします。
- テンプレートとテンプレート定義、およびプラグインとの間の依存関係を実現するとともに、前提となるプラグインが利用できないケースにも対処します。
- プラグイン定義の変数タイプを使用するビジネス オペレーションの定義時に、厳密なタイプ チェックを実施します。
- あらゆるタイプのメッセージ処理データ フォーマットおよびソースにワークフローが対応できるように、汎用メッセージ処理機能を提供します。

たとえば、Plug-in Manager で*実行時*を担当するサブセットには、次の機能があります。

- プラグインがプラグイン固有のワークフロー インスタンス データを格納および検索できるようにします。
- 必要なプラグインがロードされていない場合、ワークフローのインスタンス化を行いません。
- 初期化時のカスタム ライセンス チェックをサポートします。

プラグインを設計する場合、Plug-in Manager の管理機能にアクセスするには、次のセッション EJB を使用します。

- `com.bea.wlpi.server.plugin.PluginManager`
- `com.bea.wlpi.server.plugin.PluginManagerCfg`

これらの EJB は、次の節で説明する プラグイン API の一部です。

プラグイン API

BPM API は、コンフィグレーション、設計、およびプロセス エンジンとデプロイされたプラグインとの間の実行時対話をサポートします。

プラグイン API は、次の表で説明するように、2 つのセッション EJB、実行時管理クラスのセット、および 1 つのパッケージで構成されています。

表 1-1 BPM プラグイン API コンポーネント

コンポーネント	説明
com.bea.wlpi.server.plugin.PluginManager	ワークフローの実行時にプラグインの実行時管理を提供するステートレス セッション EJB ♦ デプロイされたプラグインに関するメタデータを提供する。 ♦ プラグインの設計コンポーネントおよび実行時コンポーネントにアクセスできるようにする。 ♦ プラグインとのイベント通知の送受信を処理する。
com.bea.wlpi.server.plugin.PluginManagerCfg	プラグインのコンフィグレーション管理および設計管理を提供するステートレス セッション EJB ♦ プラグインのコンフィグレーション情報を保守する。 ♦ プラグイン フレームワーク クラスタ全体でのステート遷移を管理する。 ♦ システム イベント通知のために登録されたプラグインのリストを保守する。
com.bea.wlpi.server.plugin.*	ワークフロー実行中にプラグインの実行時管理を行うクラス
com.bea.wlpi.common.plugin	値オブジェクト クラスなど、クライアントおよびプロセス エンジンの機能を提供するパッケージ。 このパッケージのすべてのメンバーは、クライアントとプロセス エンジン間でやりとりを行えるようにシリアライズされる。

注意： BPM API の詳細については、『*BPM クライアント アプリケーション プログラミング ガイド*』または『*BEA WebLogic Integration Javadoc*』を参照してください。

デプロイされたプラグインを BPM が検出する方法

WebLogic Server でプラグインをパッケージし、それをセッション EJB としてデプロイするだけで、BPM ユーザはプラグインを利用できるようになります。

プロセス エンジンや BPM クライアントでのインストレーションが必要ないにもかかわらず、BPM がデプロイされたプラグインとその実装詳細を検出できるのは、プラグインにより BPM に対して次の機能が提供されるからです。

- プラグインを検出する機能
- ライフサイクル タスクを管理し、プラグインに関する情報をキャッシュする機能
- プラグイン実装にアクセスし、設計クライアント内でプラグインの読み込み、表示、保存を行う機能
- 実行時にプラグインを実行する機能

プラグインの検出

プロセス エンジンは、起動時に、プラグインの `com.bea.wlpi.server.plugin.PluginHome` ホーム インタフェースに基づき、JNDI を介してプラグインを検出します。すべてのプラグイン Bean は、`PluginHome` をそのホーム インタフェースとして使用する必要があります。

`PluginHome` インタフェースの詳細については、3-4 ページの「プラグイン ホーム インタフェース」を参照してください。

ライフサイクル タスクの管理

プラグインが検出されると、Plug-in Manager は次のタスクを実行します。

- 次のメソッドを使用して、コンフィグレーション情報や依存関係など、プラグインに関する情報を取得します。
  - `getPluginInfo()` ―プラグインに関する基本情報を取得します。
  - `getPluginConfiguration()` ―プラグインのデフォルト コンフィグレーション情報を取得します。

    7-1 ページの「プラグインの管理」に説明されているように、コンフィグレーション情報は `setConfiguration()` メソッドを使用して後で設定できます。
  - `getDependencies()` ―プラグインの依存関係を取得します。

    Plug-in Manager は、プラグインをロードする前に、依存関係にあるものがすべてロードされていることを確認します。
- コンフィグレーションに基づき、`load()` メソッドまたは `unload()` メソッドを使用してプラグインをロードまたはアンロードします。

  プラグインは、ロードされて初めて利用できるようになります。プラグインがロードされると、Plug-in Manager は、`getPluginCapabilitiesInfo()` メソッドを呼び出して詳細なプラグイン情報を取得します。これにより、5-1 ページの「プラグイン通知の使い方」に説明されているように、プラグインを通知メッセージに登録することができます。この時点で、BPM クライアントからすべてのプラグイン クラスが見えるようになります。
- シャットダウン時には、`unloadPlugin()` メソッドを使用して、デプロイされたすべてのプラグインをアンロードし、消去します。

  プラグインは、そのライフサイクル中に一度だけ消去できます。

前述の各ライフサイクル メソッドは、プラグイン Bean により実装する必要のあるリモート インタフェース メソッドです。詳細については、3-6 ページの「プラグイン リモート インタフェース」を参照してください。

プラグイン実装へのアクセス

プラグインは、各コンポーネントに次のクラスを実装し、設計クライアントに対してその機能を定義する必要があります。

- 設計クライアントで表示されるプラグイン GUI コンポーネントを定義するためのプラグイン パネル クラス。詳細については、4-25 ページの「プラグイン GUI コンポーネントの表示」を参照してください。
- プラグイン データの読み込みと保存を行うためのプラグイン データ インタフェース。詳細については、4-9 ページの「プラグイン データの読み込みと保存」を参照してください。

BPM 設計クライアント（Studio など）へのプラグインのインストレーションは必要ないので、設計クライアントはプラグインが定義する具体的なクラスに関する知識を持っていません。設計クライアントでのリモート クラス ロードをサポートするために、引数を必要としないパブリック コンストラクタを実装するのはプラグインの責任です。

設計クライアントでのリモート クラス ロードは次のように行われます。

- 設計クライアントは、起動時に Plug-in Manager により取得され、キャッシュされた値オブジェクトを使用して、プラグイン コンポーネントの名前と ID に基づいてプラグイン Java クラス名を検索します。
- 検索されたクラスは、サーバからカスタム クラス ローダーによりロードされ、引数のないコンストラクタを使用して設計クライアントによりインスタンス化されます。

たとえば、1-2 ページの「プラグイン の例 : 開始ノード」の図では、ユーザが [注文開始] イベントを開始ノードのトリガとして選択すると、Plug-in Manager は、プラグイン パネル クラス `StartNodePanel` をロードします。このクラスは、引数のないコンストラクタを使用して、Studio クライアントによりインスタンス化されます。その後、Studio クライアントは、[開始のプロパティ] ダイアログボックスにプラグイン GUI コンポーネントを表示します。

実行時のプラグインの実行（コンテキストの引き渡し）

プラグイン実行特性を定義するには、プラグイン コンポーネントのための実行時インタフェースを実装する必要があります。実行時、プラグインは*コンテキスト引き渡し*と呼ばれるプロセスを使用して、プロセス エンジンやクライアントと通信します。Plug-in Manager は、プラグイン コンポーネント実行時インタフェースのインスタンスを取得し、そのインスタンスにコンテキストを渡します。

各*コンテキスト インタフェース*は、Plug-in Manager に対する制限付きアクセスを提供することにより、プラグインが独自のアプリケーション ロジックの実行と管理を行い、BPM 実行時環境にプラグイン インスタンス データを導入できるようにします。

実行時インタフェースの実装方法の詳細については、4-63 ページの「プラグインの実行」を参照してください。コンテキスト インタフェースの実装方法の詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照してください。

BPM プラグイン開発タスク

BPM プラグインを開発するには、まず、必要なプラグイン クラスおよびインタフェースを定義するセッション EJB を作成してから、このセッション EJB を WebLogic Server 上でパッケージ化およびデプロイする必要があります。

次の手順の中で、BPM プラグイン開発タスクについてさらに詳しく説明します。

注意： 次に説明する手順を実行するだけでなく、2-1 ページの「プラグイン開発の基礎」に説明されているプラグイン開発の基本的タスクも確認しておいてください。

手順 1： 設計時および実行時のカスタマイズ要件を確認します。

プラグインを使用することにより、次のワークフロー コンポーネントの設計時および実行時の動作を変更できます。

- 開始、イベント、完了の各ノード
- ワークフロー テンプレートのプロパティおよびテンプレート定義
- タスク アクション
- 関数
- メッセージ タイプ
- 変数タイプ

プラグイン サンプルには、一般的なプラグイン シナリオを表わすプラグイン クラスのセットがあります。詳細については、1-14 ページの「BPM プラグイン サンプル」を参照してください。

手順 2 : プラグイン セッション EJB を定義します。

1. `javax.ejb.SessionBean` インタフェース メソッドを実装します。たとえば、次のようなメソッドがあります。

   ```
   ejbActivate()
   ejbPassivate()
   ejbRemove()
   setSessionContext(SessionContext ctx)
   ```

2. `com.bea.wlpi.server.plugin.PluginHome` ホーム インタフェースの `ejbCreate()` メソッドを実装します。

   ホーム インタフェースは、`com.bea.wlpi.server.plugin.PluginHome` インタフェースを介して BPM プラグイン フレームワークにより定義されています。BPM ホーム インタフェースの詳細については、3-4 ページの「プラグイン ホーム インタフェース」を参照してください。

   注意 : この時点で、3-5 ページの「ホーム インタフェース メソッド」の表に説明されているように、`ejbCreate()` メソッドの実装時に Studio インタフェース ビューのためのカスタム プラグイン アイコンを設定できます。

3. `com.bea.wlpi.server.plugin.Plugin` リモート インタフェース メソッドを実装します。

   リモート インタフェースは、`com.bea.wlpi.server.plugin.Plugin` インタフェースを介して BPM プラグイン フレームワークにより定義されています。BPM リモート インタフェースの詳細については、3-6 ページの「プラグイン リモート インタフェース」を参照してください。

注意： この時点で、3-10 ページの「リモート インタフェース プラグイン情報メソッド」の表に説明されているように、`getPluginCapabilitiesInfo()` メソッドの実装時に、プラグイン コンポーネント値オブジェクトを定義し、アクション ツリー（プラグイン アクションを定義する場合）をカスタマイズする必要があります。

この手順の詳細については、3-1 ページの「プラグイン セッション EJB の定義」を参照してください。

手順 3： プラグイン コンポーネントを定義します。

1. プラグイン データ インタフェースを実装し、プラグイン データの読み取りと保存に使用されるメソッドを定義します。
2. 設計クライアント内でプラグイン GUI コンポーネントを表示するためのプラグイン パネル クラスを定義します。
3. 実行時実行特性を定義するためのプラグイン実行時コンポーネント クラスを定義します。

この手順の詳細については、4-1 ページの「プラグイン コンポーネントの定義」を参照してください。

手順 4： 通知管理を設定します。

この手順の詳細については、5-1 ページの「プラグイン通知の使い方」を参照してください。

手順 5： 着信プラグイン イベントを処理するためのイベント ハンドラを実装および登録します。

この手順の詳細については、6-1 ページの「プラグイン イベントの処理」を参照してください。

手順 6： 必要に応じて、プラグイン コンフィグレーション要件をカスタマイズし、プラグインを管理します。

この手順の詳細については、7-1 ページの「プラグインの管理」を参照してください。

手順 7： プラグインのコンテキスト依存オンライン ヘルプを開発します。

この手順の詳細については、8-1 ページの「プラグイン オンライン ヘルプの定義」を参照してください。

手順 8 : すべてのプラグイン Java クラスを EJB JAR ファイルおよび WAR ファイルにパッケージ化し、プラグインをデプロイします。

この手順の詳細については、9-1 ページの「プラグインのデプロイメント」 を参照してください。

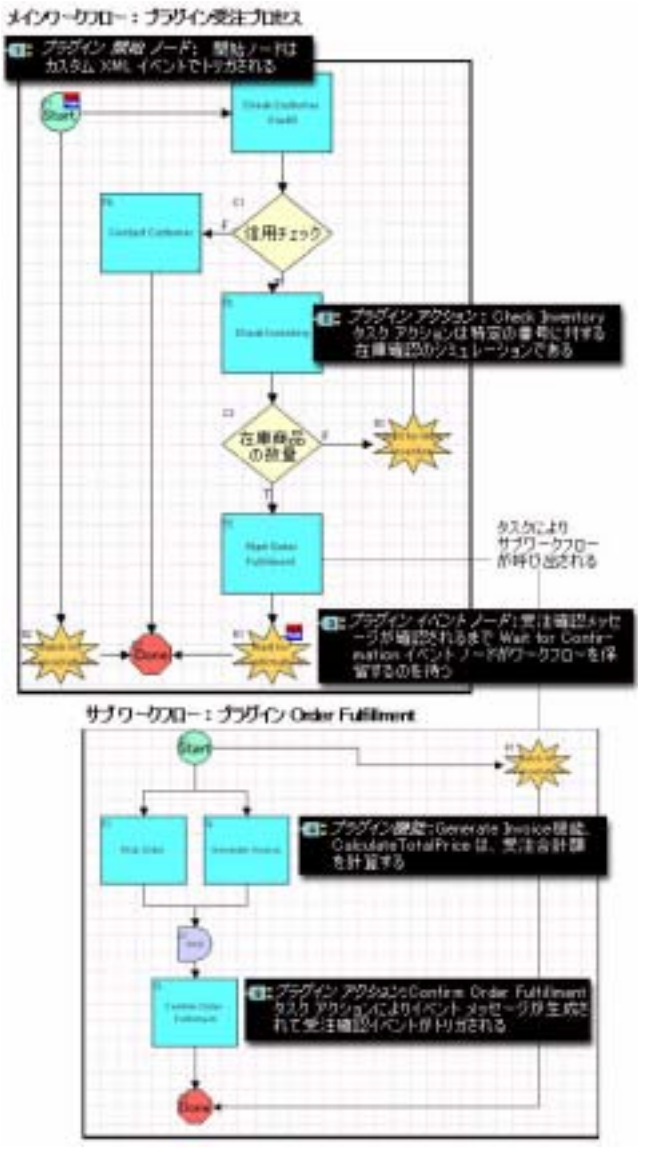
BPM プラグイン サンプル

このソフトウェアに用意されている BPM プラグイン サンプルは、一般的なプラグイン シナリオを示したプラグイン クラスのセットを提供します。このサンプルには、Plug-in Order Processing と Plug-in Order Fulfillment という 2 つのワークフロー テンプレートも入っており、いずれも *SAMPLES_HOME*`/integration/samples/bpm_api/plugin/sample_plug_in.jar` ファイルに格納されています。

注意 : プラグイン サンプルは、『*WebLogic Integration BPM ユーザーズ ガイド*』の「Business Process Management とサンプル ワークフローの紹介」 に詳しく説明されている一般的な Web ベースの発注シナリオにおおむね基づいています。

次の図に、プラグイン サンプル ワークフロー テンプレートと、追加されたプラグインを示します。

図 1-4 プラグイン サンプル ワークフロー テンプレート



前述のサンプルは、次を表しています。

- Plug-in Order Processing ワークフロー テンプレートには、サンプル プラグインからの 3 つのプラグイン コンポーネントがあります。プラグイン定義イベントによりトリガされる開始ノード、在庫チェックをシミュレートするタスク アクション、および受注確認メッセージを受信するまで処理をブロックするイベント ノードです。
- Plug-in Order Fulfillment ワークフロー テンプレートには、サンプル プラグインからの 2 つのプラグイン コンポーネントがあります。受注合計額を計算する関数と、[注文を確認] イベントをトリガするイベント メッセージを生成するタスク アクションです。

注意： 一般的な例の一部として提供されている Order Processing Trigger ワークフロー テンプレート（『*WebLogic Integration BPM ユーザーズ ガイド*』参照）がプラグインにより使用されるのではなく、Plug-in Order Processing ワークフロー テンプレートがプラグイン定義 XML イベントを介してトリガされます。

このプラグイン サンプルからの抜粋は、このドキュメントの各所で使用されています。BPM プラグイン サンプルとそのディレクトリ構造の詳細、およびサンプルのインポート方法と実行方法については、10-1 ページの「BPM プラグイン サンプル」を参照してください。一般的な Web ベースの受注シナリオの詳細については、『*WebLogic Integration BPM ユーザーズ ガイド*』を参照してください。

注意： 前述の図では、カスタマイズされたプラグイン プロパティが含まれていることを示すため、開始ノードおよびイベント ノードの右上すみに次のカスタム アイコンが表示されています。

このようなアイコンは、Studio インタフェース ビューが有効な場合に表示されます。インタフェース ビューを有効にする方法の詳細については、『*WebLogic Integration Studio ユーザーズ ガイド*』の「Studio インタフェースの使用法」を参照してください。

リモート プラグイン オブジェクト インタフェースを作成するときにカスタム プラグイン アイコンを指定する方法については、3-4 ページの「プラグイン ホーム インタフェース」 に説明されている `create()` メソッドを参照してください。

2　プラグイン開発の基礎

この章では、プラグイン開発に必要な基本的タスクについて説明します。この章の内容は以下のとおりです。

- パッケージおよびインタフェースのインポート
- Plug-in Manager への接続
- プラグイン フレームワーク バージョンの取得
- プラグイン値オブジェクトの使い方
- Plug-in Manager からの切断

また、『*BPM クライアント アプリケーション プログラミング ガイド*』の次の章も参考になります。

- 「プロセス エンジン情報へのアクセス」― WebLogic Integration プロセス エンジンに関する情報を入手する方法を説明
- 「JMS 接続の確立」― WebLogic Java Messaging Service (JMS) への接続を確立する方法を説明
- 「BPM トランザクション モデルの理解」― BPM アプリケーションでトランザクションが処理される方法を説明

パッケージおよびインタフェースのインポート

BPM パッケージおよびインタフェースをインポートし、必要であれば汎用 Java パッケージ もインポートします。

プラグイン管理のために使用される以下のものも含めて、インポートするパッケージやインタフェースの説明については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「パッケージおよびインタフェースのインポート」を参照してください。

- `com.bea.wlpi.server.plugin.PluginManager` インタフェース
- `com.bea.wlpi.server.plugin.PluginManagerCfg` インタフェース
- `com.bea.wlpi.common.plugin` パッケージ

Plug-in Manager への接続

BPM Plug-in Manager に接続するには、`PluginManager` セッション EJB と `PluginManagerCfg` セッション EJB のどちらか一方、または両方を使用します。

他の EJB と同じく、`PluginManager` EJB または `PluginManagerCfg` EJB にアクセスするには、ホーム インタフェースおよびリモート インタフェースを使用する必要があります。そのための手順は次のとおりです。

1. JNDI でセッション EJB のホーム インタフェースをルックアップします。
2. そのホーム インタフェースを使用して、リモート セッション オブジェクト（`EJBObject`）を作成します。

API セッション EJB（この場合は、`PluginManager` EJB と `PluginManagerCfg` EJB）へのアクセス方法については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「プロセス エンジンへの接続」を参照してください。

プラグイン サンプルから抜粋した後述のコード リストは、次の 2 ステップの手順により Plug-in Manager に接続する方法を示しています。

1. 初期コンテキストを作成し、JNDI コンテキスト `lookup()` メソッドを使用して、セッション EJB ホーム インタフェースにアクセスします。
2. そのホーム インタフェースを使用して、リモート セッション オブジェクトを作成します。

この抜粋は、*SAMPLES_HOME*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `SamplePluginBean.java` ファイルから取り出したものです。重要なコード行は、**太字**で示します。

注意： 初期コンテキストの詳細については、`javax.naming.InitialContext()` Javadoc を参照してください。

コード リスト 2-1 Plug-in Manager への接続

```
private final static String PLUGIN_MANAGER_CFG_HOME =
    "java:comp/env/ejb/PluginManagerCfg";
 .
 .
 .
private PluginManagerCfg getPluginManagerCfg() throws PluginException {

    PluginManagerCfg pm = null;
    InitialContext ic = null;

    try {
        ic = new InitialContext();

        PluginManagerCfgHome pmHome =
            (PluginManagerCfgHome)ic.lookup(PLUGIN_MANAGER_CFG_HOME);

        pm = pmHome.create();
    } catch (Exception e) {
        e.printStackTrace();

        throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                  "Unable to get PluginManagerCfg");
    } finally {
        try {
            ic.close();
        } catch (Exception e) {
        }
    }
```

```
    return pm;
}
```

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

プラグイン フレームワーク バージョンの取得

プラグイン フレームワーク バージョンを取得するには、次のいずれかのメソッドを使用します。

```
public com.bea.wlpi.common.VersionInfo
com.bea.wlpi.server.plugin.PluginManager.getFrameworkVersion(
) throws java.rmi.RemoteException
```

```
public com.bea.wlpi.common.VersionInfo
com.bea.wlpi.common.plugin.PluginInfo.getPluginFrameworkVersion(
) throws java.rmi.RemoteException
```

これらのメソッドは、Plug-in Manager バージョンを `com.bea.wlpi.common.VersionInfo` オブジェクトとして返します。バージョンに関する情報を取得するには、『*BPM クライアント アプリケーション プログラミング ガイド*』の「値オブジェクトのまとめ」に説明されている `VersionInfo` オブジェクト メソッドを使用します。

たとえば、次のコードは、`PluginManager` オブジェクト メソッドを使用して Plug-in Manager バージョンを取得し、その情報を `version` オブジェクトに保存します。この例では、`pm` は `PluginManager` EJB への `EJBObject` 参照を表します。

```
VersionInfo version = pm.getFrameworkVersion();
```

`getFrameworkVersion()` メソッドの詳細については、`com.bea.wlpi.server.plugin.PluginManager` Javadoc を参照してください。`getPluginFrameworkVersion()` メソッドの詳細については、`com.bea.wlpi.common.plugin.PluginInfo` Javadoc を参照してください。

プラグイン値オブジェクトの使い方

com.bea.wlpi.common.plugin パッケージは、定義時と実行時の両方でプラグイン オブジェクト データを取得するための *Info* クラス、すなわち *値オブジェクト*を提供します。プロセス エンジンと BPM クライアント アプリケーションは、プラグイン値オブジェクトを使用して、指定されたロケールに関するプラグイン オブジェクト データを要求し、プラグインで表示文字列などのリソースを適切にローカライズできるようにします。

値オブジェクトは、設計クライアントでのリモート クラスのロードに重要な役割を果たします。特に、BPM 設計クライアントは値オブジェクトを使用して、次のことを行います。

1. プラグイン コンポーネントの名前および ID に基づいて、プラグインに含まれる Java クラスの名前を検索します。
2. 戻り値に基づいて、設計クライアントでリモート クラス ロードを開始し、定義されたクラスのインスタンスを取得します。

リモート クラス ロードの詳細については、1-10 ページの「プラグイン実装へのアクセス」を参照してください。

次の表に、プラグイン オブジェクト データの作成およびアクセスに使用できる値オブジェクトを示します。

表 2-1 プラグイン値オブジェクト

使用する値オブジェクト	アクセスするプラグイン
com.bea.wlpi.common.plugin.ActionCategoryInfo	アクションまたはアクション カテゴリ情報
com.bea.wlpi.common.plugin.ActionInfo	アクション情報
com.bea.wlpi.common.plugin.CategoryInfo	アクション カテゴリ情報
com.bea.wlpi.common.plugin.ConfigurationData	コンフィグレーション データ
com.bea.wlpi.common.plugin.ConfigurationInfo	コンフィグレーション情報
com.bea.wlpi.common.plugin.DoneInfo	完了ノード情報

表 2-1 プラグイン値オブジェクト （続き）

使用する値オブジェクト	アクセスするプラグイン
com.bea.wlpi.common.plugin.EventHandlerInfo	イベント ハンドラ情報
com.bea.wlpi.common.plugin.EventInfo	イベント ノード情報
com.bea.wlpi.common.plugin.FieldInfo	メッセージ タイプ情報
com.bea.wlpi.common.plugin.FunctionInfo	エバリュエータ関数情報
com.bea.wlpi.common.plugin.HelpSetInfo	オンライン ヘルプ情報
com.bea.wlpi.common.plugin.InfoObject	すべてのプラグイン値オブジェクトのための抽象ベース クラス
com.bea.wlpi.common.plugin.PluginCapabilitiesInfo	機能情報
com.bea.wlpi.common.plugin.PluginDependency	依存関係情報
com.bea.wlpi.common.plugin.PluginInfo	基本プラグイン情報
com.bea.wlpi.common.plugin.StartInfo	開始ノード情報
com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo	テンプレート定義プロパティ情報
com.bea.wlpi.common.plugin.TemplateNodeInfo	テンプレート ノード（完了および開始）情報
com.bea.wlpi.common.plugin.TemplatePropertiesInfo	テンプレート プロパティ情報
com.bea.wlpi.common.plugin.VariableTypeInfo	変数タイプ情報

次の節では、値オブジェクトの作成および使用方法について説明します。

値オブジェクト コンストラクタ、および関連付けられた get メソッドと set メソッドの詳細については、B-1 ページの「プラグイン値オブジェクトのまとめ」を参照してください。値オブジェクトにより共有される一般的特性のリストと、値オブジェクトをソートする方法については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「値オブジェクトの使用法」を参照してください。

プラグイン値オブジェクトの定義

プラグイン値オブジェクトを定義するには、関連付けられているコンストラクタを使用します。2-5 ページの「プラグイン値オブジェクト」の表に記載された各プラグイン値オブジェクトには、オブジェクト データを作成するためのコンストラクタが 1 つまたは複数用意されています。値オブジェクトを作成するためのコンストラクタの詳細については、B-1 ページの「プラグイン値オブジェクトのまとめ」を参照してください。

`com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` オブジェクトを定義する際には、それぞれのプラグイン コンポーネントについてプラグイン値オブジェクトを渡す必要があります。詳細については、3-10 ページの「リモート インタフェース プラグイン情報メソッド」の表の `getPluginCapabilitiesInfo()` メソッドの説明を参照してください。

値オブジェクトを作成する場合、次を指定する必要があります。

- プラグイン ID。これは、それぞれのプラグインとオブジェクト タイプでユニークである必要があります。
- プラグイン機能の記述。
- グローバルにユニークな内部識別子。これは、ベンダ逆引き DNS 名の末尾に 1 つまたは複数のドット区切り文字列を付加した形式です（たとえば、`com.somedomain.someproduct.myplugin`）。
- プラグインと関連付けられた Java クラス名の入った配列。2-8 ページの「オブジェクト データの取得と設定」に説明されているように、設計クライアントはプラグインの名前と ID を使用してこの情報を検索できます。その後、設計クライアントは、リモート クラス ロードを開始し、実装クラスにアクセスできます。リモート クラス ロードの詳細については、1-10 ページの「プラグイン実装へのアクセス」を参照してください。
- インタフェース ビューが有効な場合にプラグインを表すために WebLogic Integration Studio により使用されるアイコン。カスタム アイコンは、3-4 ページの「プラグイン ホーム インタフェース」に説明されているように、ホーム インタフェース `ejbCreate()` メソッドの実装時に定義できます。

たとえば、次のコードは、`StartInfo` オブジェクトを作成し、結果のオブジェクトを `si` に割り当てます。

```
si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
          bundle.getString("startOrderName"),
          bundle.getString("startOrderDesc"), ICON_BYTE_ARRAY,
          SamplePluginConstants.START_CLASSES, orderFieldInfo);
```

START_CLASSES 配列は、プラグイン開始ノードのためのプラグイン データ、プラグイン パネル、実行時コンポーネント クラス名を定義します。

START_CLASSES 配列は、SamplePluginConstants.java クラス ファイル内で次のように定義されます。

```
final static String[] START_CLASSES = {
START_DATA,
START_PANEL,
START_NODE };
```

この例では、配列変数値は、SamplePluginConstants.java ファイルで次のように定義されます。

```
final static String START_NODE =
"com.bea.wlpi.tour.po.plugin.StartNode";
final static String START_DATA =
"com.bea.wlpi.tour.po.plugin.StartNodeData";
final static String START_PANEL =
"com.bea.wlpi.tour.po.plugin.StartNodePanel";
```

ICON_BYTE_ARRAY 変数は、グラフィカル イメージ（アイコン）のバイト配列表現を指定するもので、Studio インタフェース ビューが有効な場合に、プラグインを表すために Studio により使用されます。

値オブジェクト コンストラクタ、および関連付けられた get メソッドと set メソッドの詳細については、B-1 ページの「プラグイン値オブジェクトのまとめ」を参照してください。

オブジェクト データの取得と設定

2-5 ページの「プラグイン値オブジェクト」の表に記載した各プラグイン値オブジェクトは、オブジェクト データの取得および設定のためのさまざまなメソッドを提供します。これらのメソッドの詳細については、B-1 ページの「プラグイン値オブジェクトのまとめ」を参照してください。

たとえば、次のコードは、プラグイン開始ノードのための PluginTriggerPanel 実装クラスを取得し、それを startpanel 文字列に保存します。

```
java.lang.String startpanel = si.getClassName(KEY_PANEL);
```

この例では、si は、プラグイン開始ノードのための com.bea.wlpi.common.plugin.StartInfo 値オブジェクトに対する参照を表します。

プラグイン値オブジェクト メソッドの詳細については、B-1 ページの「プラグイン値オブジェクトのまとめ」を参照してください。

Plug-in Manager からの切断

BPM Plug-in Manager から切断するには、次の手順を実行します。

1. セッション EJB 参照を削除し、そのシステム スペースを他の EJB のために利用できるようにします。

   たとえば、プラグイン サンプルから抜粋した次のコードは、PluginManagerCfg EJB 参照を削除する方法を示しています。この例では、pm は PluginMangerCfg EJB への EJBObject 参照を表します。

   ```
   try {
      if (pm != null)
      pm.remove();
   } catch (Exception e) {}
   ```

   詳細については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「プロセス エンジンからの切断」の「セッション EJB 参照の削除」を参照してください。

2. JMS 接続など、その他のリソースを削除し（該当する場合）、コンテキストをクローズします。

   たとえば、次のコードは、JNDI コンテキスト リソースをクローズします。

   ```
   try {
       ic.close();
   { catch(Exception exp) {}
   ```

   詳細については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「プロセス エンジンからの切断」の「他のリソースの解放」を参照してください。

3　プラグイン セッション EJB の定義

この章では、プラグイン セッション EJB を定義する方法について説明します。この章の内容は以下のとおりです。

- 概要
- セッション EJB インタフェース
- プラグイン ホーム インタフェース
- プラグイン リモート インタフェース

概要

プラグイン セッション EJB を定義するには、次の表に説明する 3 つの事前定義されたインタフェースを実装する必要があります。

表 3-1　プラグイン セッション EJB で必要なインタフェース

名前	インタフェース	説明
セッション EJB	`javax.ejb.SessionBean`	すべてのセッション EJB により実装される必要のあるインタフェース。
プラグイン ホーム インタフェース	`com.bea.wlpi.server.plugin.PluginHome`	`javax.ejb.EJBHome` インタフェースの拡張。すべてのプラグインのためのホーム インタフェースを定義する。
プラグイン リモート インタフェース	`com.bea.wlpi.server.plugin.Plugin`	`javax.ejb.EJBObject` インタフェースの拡張。すべてのプラグインのためのリモート インタフェースを定義する。

次の節では、プラグイン セッション EJB を定義する際に実装する必要のある各インタフェースとメソッドについて説明します。プラグイン サンプルからの抜粋も示します。

セッション EJB インタフェース

定義により、セッション EJB は、`javax.ejb.SessionBean` とそのメソッドを実装する必要があります。

注意： `SessionBean` インタフェース メソッドの内容は、空であることも、単純にメッセージをログに返すこともありますが、必ず実装します。

次の表に、実装する必要のあるセッション EJB メソッドを示します。

表 3-2 セッション EJB メソッド

メソッド	説明
`public void ejbActivate() throws javax.ejb.EJBException, java.rmi.RemoteException`	インスタンスをアクティブ化する。
`public void ejbPassivate() throws javax.ejb.EJBException, java.rmi.RemoteException`	インスタンスをパッシブにする。
`public void ejbRemove() throws javax.ejb.EJBException, java.rmi.RemoteException`	インスタンスを削除する。
`public void setSessionContext(javax.ejb.SessionContext ctx) throws javax.ejb.EJBException, java.rmi.RemoteException`	関連付けられたセッション コンテキストを設定する。 メソッド パラメータの定義は次のとおり。 *ctx* ―セッション コンテキストを示す `javax.ejb.SessionContext` オブジェクト。

各メソッドの詳細については、`javax.ejb.SessionBean` Javadoc を参照してください。

プラグイン サンプルから抜粋した次のコード リストは、javax.ejb.SessionBean インタフェースとそのメソッドを実装する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SamplePluginBean.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 3-1 セッション EJB インタフェースの実装

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.VersionInfo;
import com.bea.wlpi.common.plugin.*;
import com.bea.wlpi.common.plugin.PluginData;
import com.bea.wlpi.server.plugin.InstanceNotification;
import com.bea.wlpi.server.plugin.Plugin;
import com.bea.wlpi.server.plugin.PluginManagerCfg;
import com.bea.wlpi.server.plugin.PluginManagerCfgHome;
import com.bea.wlpi.server.plugin.TaskNotification;
import com.bea.wlpi.server.plugin.TemplateDefinitionNotification;
import com.bea.wlpi.server.plugin.TemplateNotification;
import java.lang.reflect.Constructor;
import java.util.MissingResourceException;
import java.util.ResourceBundle;
import java.util.Locale;
import java.net.URL;
import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import java.io.*
/**
 * @homeInterface com.bea.wlpi.server.plugin.PluginHome
 * @remoteInterface com.bea.testplugin.SamplePlugin
 * @statemode Stateless
 */
public class SamplePluginBean implements SessionBean {
    private SessionContext ctx;
    private final static String PLUGIN_MANAGER_CFG_HOME =
        "java:comp/env/ejb/PluginManagerCfg";
    private static byte[] ICON_BYTE_ARRAY;

    // javax.ejb.SessionBean を実装
```

```
public void ejbActivate() {
}

// javax.ejb.SessionBean を実装

public void ejbRemove() {
}

// javax.ejb.SessionBean を実装

public void ejbPassivate() {
}

// javax.ejb.SessionBean を実装

public void setSessionContext(SessionContext ctx) {
    this.ctx = ctx;
}
.
.
.
```

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」 を参照してください。

プラグイン ホーム インタフェース

ホーム インタフェースは、ユーザのために BPM プラグイン フレームワークにより定義されます。`com.bea.wlpi.server.plugin.PluginHome` インタフェースは、`javax.ejb.EJBHome` インタフェースを拡張し、すべてのプラグインのホーム インタフェースを定義します。

次の表で、`PluginHome` ホーム インタフェースにより定義されるメソッドについて説明します。

表 3-3 ホーム インタフェース メソッド

メソッド	説明
`public com.bea.wlpi.server.plugin.Plugin create() throws java.rmi.RemoteException java.rmi.RemoteException`	リモート プラグイン オブジェクト インタフェースを作成する。 `imageStreamToByteArray()` メソッドを `com.bea.wlpi.common.plugin.InfoObject` に対して使用することにより、WebLogic Integration Studio インタフェース ビューのためのカスタム プラグイン アイコンを設定することもできる。例を示す。 `ICON_BYTE_ARRAY = InfoObject.imageStreamToByteArray( getClass().getResourceAsStream( "image") );` この例では、*`image`* は、カスタム プラグイン アイコンを指定する。 `imageStreamToByteArray()` メソッドの詳細については、B-30 ページの「InfoObject オブジェクト」を参照。 特定のプラグイン コンポーネントのための値オブジェクトを作成する際、インタフェース ビューが有効なときにプラグイン コンポーネントを表わすために Studio により使用されるアイコンを指定できる。詳細については、2-7 ページの「プラグイン値オブジェクトの定義」を参照。 このメソッドは、`com.bea.wlpi.server.plugin.Plugin` オブジェクトを返す。 Studio でインタフェース ビューを有効にする方法については、『*WebLogic Integration Studio ユーザーズ ガイド*』の「Studio インタフェースの使用法」を参照。

ホーム インタフェースの詳細については、`com.bea.wlpi.server.plugin.PluginHome` Javadoc を参照してください。

プラグイン サンプルから抜粋した次のコード リストは、ホーム インタフェースで宣言される 1 つの create() メソッドのための ejbCreate() メソッドを実装し、Studio インタフェース ビューのためのカスタム プラグイン アイコンを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SamplePluginBean.java ファイルから取り出したものです。

コード リスト 3-2 ホーム インタフェース メソッドの実装

```
// javax.ejb.SessionBean を実装

public void ejbCreate() throws CreateException {
    try {
        ICON_BYTE_ARRAY = InfoObject.imageStreamToByteArray(
                          getClass().getResourceAsStream("Sample.gif"));
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
```

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」 を参照してください。

プラグイン リモート インタフェース

リモート インタフェースは、BPM プラグイン フレームワークにより定義されます。com.bea.wlpi.server.plugin.Plugin インタフェースは、javax.ejb.EJBObject インタフェースを拡張し、すべてのプラグインのリモート インタフェースを定義します。

プラグイン セッション EJB を定義する場合、Plugin リモート インタフェースとそのメソッドを実装する必要があります。

Plugin リモート インタフェースは、次のカテゴリのメソッドを定義します。

- ライフサイクル管理

- 通知
- プラグイン情報
- オブジェクト作成

次の節では、実装する必要のある Plugin リモート インタフェースをカテゴリ別に説明します。

注意： リモート インタフェース メソッドの内容は、空のままにすること、あるいは単にログに対するメッセージを戻すことができますが、必ず実装します。

ライフサイクル管理メソッド

次の表に、実装する必要のあるリモート インタフェースにより定義されるライフサイクル管理メソッドを示します。BPM プラグイン ライフサイクルの詳細については、1-9 ページの「ライフサイクル タスクの管理」を参照してください。

表 3-4 リモート インタフェース ライフサイクル管理メソッド

メソッド	説明
`public void load( com.bea.wlpi.common.plugin.PluginObject *config*) throws java.rmi.RemoteException, com.bea.wlip.common.plugin.PluginException`	指定されたコンフィグレーションを使用してプラグインをロードし、指定されている場合には、そのプラグインを通知リスナとして登録する。プラグインを通知リスナとして登録する方法については、5-1 ページの「プラグイン通知の使い方」を参照。 メソッド パラメータの定義は次のとおり。 *config* ― `com.bea.wlpi.common.plugin.ConfigurationInfo` オブジェクト。これは、`com.bea.wlpi.common.plugin.PluginObject` オブジェクトにより提供され、プラグイン コンフィグレーション データを指定する。 **注意：** プラグインは、クラスタのプライベート ステートを、必要に応じてクラスタ全体に複製する役割を担当します。

表 3-4 リモート インタフェース ライフサイクル管理メソッド （続き）

メソッド	説明
`public void unload() throws java.rmi.RemoteException, com.bea.wlip.common.plugin.PluginException`	プラグインをアンロードし、必要であれば、通知リスナとしての登録を解除する。プラグインを通知リスナとして登録および登録解除する方法については、5-1 ページの「プラグイン通知の使い方」を参照。

リモート インタフェース ライフサイクルを管理するメソッドの詳細については、`com.bea.wlpi.server.plugin.Plugin` Javadoc を参照してください。

通知メソッド

次の表に、実装する必要のあるリモート インタフェースにより定義される通知メソッドを定義します。

注意： 通知を受信するには、プラグインが通知リスナとして登録されている必要があります。詳細については、5-1 ページの「プラグイン通知の使い方」を参照してください。

表 3-5 リモート インタフェース通知メソッド

メソッド	説明
`public void instanceChanged( com.bea.wlpi.common.plugin.InstanceNotification` *`notify`*`) throws java.rmi.RemoteException, com.bea.wlpi.common.plugin.PluginException`	ワークフロー インスタンスに対する変更をプラグインに通知する。 メソッド パラメータの定義は次のとおり。 *`notify`* ―ワークフロー インスタンス変更を示す `com.bea.wlpi.server.plugin.InstanceNotification` オブジェクト。

表 3-5 リモート インタフェース通知メソッド （続き）

メソッド	説明
`public void taskChanged( com.bea.wlpi.common.plugin.TaskNotification *notify*) throws java.rmi.RemoteException, com.bea.wlpi.common.plugin.PluginException`	タスク インスタンスに対する変更をプラグインに通知する。 メソッド パラメータの定義は次のとおり。 *notify* ―タスク変更を示す `com.bea.wlpi.server.plugin.TaskNotification` オブジェクト。
`public void templateChanged( com.bea.wlpi.common.plugin.TemplateNotification *notify*) throws java.rmi.RemoteException, com.bea.wlpi.common.plugin.PluginException`	テンプレートに対する変更をプラグインに通知する。 メソッド パラメータの定義は次のとおり。 *notify* ―テンプレート変更を示す `com.bea.wlpi.server.plugin.TemplateNotification` オブジェクト。
`public void templateDefinitionChanged( com.bea.wlpi.common.plugin.TemplateDefinitionNotification *notify*) throws java.rmi.RemoteException, com.bea.wliwlpip.common.plugin.PluginException`	テンプレート定義に対する変更をプラグインに通知する。 メソッド パラメータの定義は次のとおり。 *notify* ―テンプレート定義変更を示す `com.bea.wlpi.server.plugin.TemplateDefinitionNotification` オブジェクト。

リモート インタフェース通知メソッドの詳細については、`com.bea.wlpi.server.plugin.Plugin` Javadoc を参照してください。

プラグイン情報メソッド

次の表に、実装する必要のあるリモート インタフェースにより定義されるプラグイン情報メソッドを示します。

表 3-6 リモート インタフェース プラグイン情報メソッド

メソッド	説明
`public java.lang.Class classForName( java.lang.String *className*) throws java.rmi.RemoteException, java.lang.ClassNotFoundException, com.bea.wlpi.common.plugin.PluginException`	プラグイン提供のメタデータ オブジェクトの 1 つを呼び出すことにより、プラグイン定義されたクラスをインスタンス化する。 メソッド パラメータの定義は次のとおり。 *`className`* ― インスタンス化する Java クラスの完全修飾名を示す `java.lang.String` オブジェクト。 このメソッドは、指定された名前を持つクラスを返す。
`public com.bea.wlpi.common.plugin.PluginDependency[] getDependencies() throws java.rmi.RemoteException`	現在のプラグインが依存しているプラグインを取得する。 Plug-in Manager は、現在のプラグインのロードを試行する前に、すべての依存プラグインがロードされていることを確認する。 このメソッドは、`com.bea.wlpi.common.plugin.PluginDependency` オブジェクトのリストを返す。各プラグインの依存関係に関する情報にアクセスするには、B-31 ページの「PluginCapabilitiesInfo オブジェクト」 に説明されている `PluginDependency` オブジェクト メソッドを使用する。

表 3-6 リモート インタフェース プラグイン情報メソッド （続き）

メソッド	説明
`public java.lang.String getName() throws java.rmi.RemoteException`	現在のプラグインのグローバルにユニークな名前を逆引き DNS フォーマットで取得する。 **注意：** 逆引き DNS フォーマットは、グローバル ネームスペースの衝突を防止します。 このメソッドは、ユニークなプラグイン名を示す `java.lang.String` オブジェクトを返す。

表 3-6 リモート インタフェース プラグイン情報メソッド （続き）

メソッド	説明
`public com.bea.wlpi.common.plugin.PluginCapabilitiesInfo getPluginCapabilitiesInfo( java.util.Locale lc, com.bea.wlpi.common.plugin.CategoryInfo info) throws java.rmi.RemoteException`	プラグイン機能に関する詳細情報を取得する。 次の場合にこのメソッドを使用する。 ■ 2-5 ページの「プラグイン値オブジェクトの使い方」に定義されているように、プラグイン コンポーネント値オブジェクトを定義する。 ■ 4-72 ページの「アクション ツリーをカスタマイズする」に説明されているように、プラグイン アクションを定義するときにアクション ツリーをカスタマイズする。 ■ 6-12 ページの「プラグイン イベント ハンドラの定義」に説明されているように、プラグイン例外ハンドラを登録する。 メソッド パラメータの定義は次のとおり。 ■ *lc* －表示文字列をローカライズするロケールを示す `java.util.Locale` オブジェクト。 ■ *info* －事前定義されたアクションおよびプラグイン定義されたアクションの両方が入った既存のアクション カテゴリ ツリーを示す `com.bea.wlpi.common.plugin.CategoryInfo` オブジェクト。NULL 値は、アクション カテゴリ情報が必要ないことを示す。 このメソッドは、要求された場合、アクション ツリーに挿入すべき新しいカテゴリとアクションの入った `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` オブジェクトを返す。プラグイン機能に関する情報にアクセスするには、B-31 ページの「PluginCapabilitiesInfo オブジェクト」に説明されている `PluginCapabilitiesInfo` オブジェクト メソッドを使用する。

表 3-6 リモート インタフェース プラグイン情報メソッド （続き）

メソッド	説明
`public com.bea.wlpi.common.plugin.PlugInfo getPluginInfo(java.util.Locale lc) throws java.rmi.RemoteException`	プラグインに関する基本情報を取得する。 メソッド パラメータの定義は次のとおり。 *lc* ―表示文字列をローカライズするロケールを示す `java.util.Locale` オブジェクト。 このメソッドは、`com.bea.wlpi.common.plugin.PluginInfo` オブジェクトを返す。プラグインに関する情報にアクセスするには、B-35 ページの「PluginInfo オブジェクト」に説明されている `PluginInfo` オブジェクト メソッドを使用する。
`public com.bea.wlpi.common.VersionInfo getVersion() throws java.rmi.RemoteException`	プラグイン バージョン情報を取得する。 このメソッドは、`com.bea.wlpi.common.VersionInfo` オブジェクトを返す。プラグインに関する情報にアクセスするには、『*BPM クライアント アプリケーション プログラミング ガイド*』の「値オブジェクトのまとめ」の「VersionInfo オブジェクト」に説明されている `VersionInfo` オブジェクト メソッドを使用する。
`public void setConfiguration(com.bea.wlpi.common.plugin.PluginObject config) throws java.rmi.RemoteException`	プラグイン コンフィグレーション情報を設定する。 メソッド パラメータの定義は次のとおり。 *config* ―プラグイン コンフィグレーション情報を示す `com.bea.wlpi.common.plugin.PluginInfo` オブジェクト。 詳細については、7-1 ページの「プラグインの管理」を参照。

リモート インタフェース プラグイン情報メソッドの詳細については、`com.bea.wlpi.server.plugin.Plugin` Javadoc を参照してください。

オブジェクト作成メソッド

次の表に、実装する必要のあるリモート インタフェースにより定義されるオブジェクト作成メソッドを定義します。

表 3-7 リモート インタフェース オブジェクト作成メソッド

メソッド	説明

`public java.lang.Object getObject(`
`java.util.Locale` *`lc`*`,`
`java.lang.String` *`className`*`) throws`
`java.rmi.RemoteException,`
`java.lang.ClassNotFoundException,`
`com.bea.wlip.common.plugin.PluginException`
	プラグイン提供のメタデータ オブジェクトの 1 つを呼び出すことにより、プラグイン定義されたオブジェクトを取得する。
	メソッド パラメータの定義は次のとおり。
	■ *`lc`* ―表示文字列をローカライズするロケールを示す `java.util.Locale` オブジェクト。
	■ *`className`* ―インスタンス化する Java クラスの完全修飾名を示す `java.lang.String` オブジェクト。
	このメソッドは、名前付きクラスの `java.lang.Object` インスタンスを返す。

リモート インタフェース オブジェクト作成メソッドの詳細については、`com.bea.wlpi.server.plugin.Plugin` Javadoc を参照してください。

リモート インタフェースの実装例

プラグイン サンプルから抜粋した次のコード リストは、リモート インタフェースとそのメソッドを実装する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SamplePluginBean.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 3-3 リモート インタフェースの実装

```
    // プラグインの実装

    /**
     * プラグインをロードする。プラグインは、この時点で、
     * 各種システム イベントへの関心を登録する必要がある
     * @param pluginData プラグイン コンフィグレーション データ
     * @see PluginManager#addTemplateListener
     * @see PluginManager#addTemplateDefinitionListener
     * @see PluginManager#addInstanceListener
     * @see PluginManager#addTaskListener
     * @throws PluginException
     */
    public void load(PluginObject pluginData) throws PluginException {

        log("load called");
        // このブロックによる通知のサブスクライブを有効にする

        /*
        PluginManagerCfg pm = null;
        try {
            pm = getPluginManagerCfg();
            Plugin plugin = (Plugin)ctx.getEJBObject();
         pm.addInstanceListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
            pm.addTaskListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
            pm.addTemplateDefinitionListener(plugin,
PluginConstants.EVENT_NOTIFICATION_ALL);
          pm.addTemplateListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
        } catch (Exception e) {
            e.printStackTrace();
           throw new PluginException(SamplePluginConstants.PLUGIN_NAME, "Unable
to get PluginManager");
        } finally {
            try {
```

```
            if (pm != null)
                pm.remove();
        } catch (Exception e) {
        }
    }
    */
    log("loaded");
}

/**
 * プラグインをアンロードする。プラグイン フレームワークは、イベント通知を
 * サブスクライブしていた場合、プラグインを登録解除する
 */
public void unload() {
    log("unload called");
}

public PluginDependency[] getDependencies() {

    log("getDependencies called");

    return null;
}

public String getName() {
    return SamplePluginConstants.PLUGIN_NAME;
}

public VersionInfo getVersion() {
    return SamplePluginConstants.PLUGIN_VERSION;
}

/**
 * プラグインに関する詳細情報を返す
 * @param lc 表示文字列をローカライズするロケール
 * @return プラグインに関する詳細情報
 */
public PluginInfo getPluginInfo(Locale lc) {

    log("getPluginInfo called");

    return createPluginInfo(lc);
}

public void setConfiguration(PluginObject config) throws PluginException {
}

/**
```

```
     * プラグイン機能の詳しい説明を返す。新しいサブカテゴリとアクションを
     * 追加するには、プラグインは、渡された <code>info</code> パラメータを
     * 介して渡されるカテゴリ ID を使用して、親カテゴリ、および
    * 新しいカテゴリ ID として {@link ActionCategoryInfo#ID_PLUGIN} を識別する必要がある
     * PluginManager は、この呼び出しにより返される CategoryInfo 配列を
     * 現在の構造（<code>info</code> パラメータにより渡される）と結合し、
     * {@link ActionCategoryInfo#ID_PLUGIN} への参照を、新たに割り当てられた
     * ユニークなカテゴリ ID に置き換える。この事前定義されたカテゴリは、次の ID を持つ
    * {@link ActionCategoryInfo#ID_TASK}, {@link ActionCategoryInfo#ID_WORKFLOW},
     * {@link ActionCategoryInfo#ID_INTEGRATION}, {@link
ActionCategoryInfo#ID_MISCELLANEOUS},
     * {@link ActionCategoryInfo#ID_EXCEPTION}.


     * @param lc 表示文字列をローカライズするロケール
     * @param info 既存のアクション カテゴリ ツリー。事前定義された
     * カテゴリとアクション、およびプラグイン定義されたカテゴリとアクション
     * （以前にロードされたプラグインにより）の両方を含む
     * @return ツリーに挿入する新しいカテゴリとアクション
     */
    public PluginCapabilitiesInfo getPluginCapabilitiesInfo(Locale lc,
            CategoryInfo[] info) {

        PluginInfo pi;
        FieldInfo orderFieldInfo;
        FieldInfo confirmFieldInfo;
        FieldInfo[] fieldInfo;
        FunctionInfo fi;
        FunctionInfo[] functionInfo;
        StartInfo si;
        StartInfo[] startInfo;
        EventInfo ei;
        EventInfo[] eventInfo;
        SampleBundle bundle = new SampleBundle(lc);

        log("getPluginCapabilities called");

        pi = createPluginInfo(lc);
        orderFieldInfo =
            new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 3,
                          bundle.getString("orderFieldName"),
                          bundle.getString("orderFieldDesc"),
                          SamplePluginConstants.ORDER_FIELD_CLASSES, false);
        confirmFieldInfo =
            new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 4,
                          bundle.getString("confirmFieldName"),
                          bundle.getString("confirmFieldDesc"),
```

```
                        SamplePluginConstants.CONFIRM_FIELD_CLASSES, false);
        fieldInfo = new FieldInfo[]{ orderFieldInfo, confirmFieldInfo };
        ei = new EventInfo(SamplePluginConstants.PLUGIN_NAME, 6,
                           bundle.getString("confirmOrderName"),
                       bundle.getString("confirmOrderDesc"), ICON_BYTE_ARRAY,
                           SamplePluginConstants.EVENT_CLASSES,
                           confirmFieldInfo);
        eventInfo = new EventInfo[]{ ei };
        fi = new FunctionInfo(SamplePluginConstants.PLUGIN_NAME, 7,
                              bundle.getString("calcTotalName"),
                              bundle.getString("calcTotalDesc"),
                              bundle.getString("calcTotalHint"),
                              SamplePluginConstants.FUNCTION_CLASSES, 3, 3);
        functionInfo = new FunctionInfo[]{ fi };
        si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
                           bundle.getString("startOrderName"),


                          bundle.getString("startOrderDesc"), ICON_BYTE_ARRAY,
                          SamplePluginConstants.START_CLASSES, orderFieldInfo);
        startInfo = new StartInfo[]{ si };

        PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
                                         getCategoryInfo(bundle), eventInfo,
                                         fieldInfo, functionInfo, startInfo,
                                         null, null, null, null, null);

        return pci;
    }

    /**
     * プラグイン定義クラスを返す。呼び出し側は、プラグイン提供の
     * メタデータ オブジェクトの 1 つを呼び出すことによりクラス名を検索する
     * @param className インスタンス化する Java クラスの完全修飾名
     * @return 指定された名前を持つクラス
     * @throws ClassNotFoundException プラグインがクラスをロードできなかった場合
     * @see ActionInfo
     * @see EventInfo
     * @see FunctionInfo
     * @see FieldInfo
     * @see PluginInfo
     * @see StartInfo
     * @see DoneInfo
     * @see VariableTypeInfo
     * @see TemplatePropertiesInfo
     * @see TemplateDefinitionPropertiesInfo
     * @throws PluginException
     */
```

```
    public Class classForName(String className)
            throws ClassNotFoundException, PluginException {

        log("classForName called");

        return Class.forName(className);
    }

    /**
     * プラグイン定義されたオブジェクトを返す。呼び出し側は、プラグイン提供のメタデータ オブジェクトの
     * メタデータ オブジェクトの 1 つを呼び出すことによりクラス名を検索する
     * @param lc 表示文字列をローカライズするロケール
     * @param className インスタンス化する Java クラスの完全修飾名
     * @return
     * @see ActionInfo
     * @see EventInfo
     * @see FunctionInfo
     * @see FieldInfo
     * @see PluginInfo


     * @see StartInfo
     * @see DoneInfo
     * @see VariableTypeInfo
     * @see TemplateInfo
     * @see TemplateDefinitionInfo
     * @throws ClassNotFoundException
     * @throws PluginException
     */
    public Object getObject(Locale lc, String className)
            throws ClassNotFoundException, PluginException {

        log("getObject called with class name " + className);

        try {
            Class cls = Class.forName(className);
            Object obj;

            try {
                // Locale オブジェクトを取得する ctor があるかどうかを確認する
                Class[] paramType = new Class[]{ lc.getClass() };
                Constructor ctor = cls.getConstructor(paramType);
                Object[] param = new Object[]{ lc };

                obj = ctor.newInstance(param);

                return (obj);
```

```
            } catch (Exception e) {
            }

            // ctor がロケールを取得しない場合、引数なしの ctor を呼び出す
            return Class.forName(className).newInstance();
        } catch (InstantiationException ie) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Unable to instantiate class: " + ie);
        } catch (IllegalAccessException iae) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Unable to instantiate class: " + iae);
        }
    }

    /**
     * テンプレートでの変更をプラグインに通知する
     * @param e 変更を示すイベント オブジェクト
     */
    public void templateChanged(TemplateNotification e) {
        log("templateChanged called");
    }

    /**


     * テンプレート定義での変更をプラグインに通知する
     * @param e 変更を示すイベント オブジェクト
     */
    public void templateDefinitionChanged(TemplateDefinitionNotification e) {
        log("templateDefinitionChanged called");
    }

    /**
     * ワークフロー インスタンスでの変更をプラグインに通知する
     * @param e 変更を示すイベント オブジェクト
     */
    public void instanceChanged(InstanceNotification e) {
        log("instanceChanged called");
    }

    /**
     * タスク インスタンスでの変更をプラグインに通知する
     * @param e 変更を示すイベント オブジェクト
     */
    public void taskChanged(TaskNotification e) {
        log("taskChanged called");
    }
```

```
private PluginInfo createPluginInfo(Locale lc) {

    HelpSetInfo helpSet;
    PluginInfo pi;
    SampleBundle bundle = new SampleBundle(lc);
    String name = bundle.getString("pluginName");
    String desc = bundle.getString("pluginDesc");
    String helpName = bundle.getString("helpName");
    String helpDesc = bundle.getString("helpDesc");

    helpSet = new HelpSetInfo(SamplePluginConstants.PLUGIN_NAME, helpName,
                              helpDesc,
                              new String[]{ "htmlhelp/Sample", "index" },
                              HelpSetInfo.HELP_HTML);
    pi = new PluginInfo(SamplePluginConstants.PLUGIN_NAME, name, desc, lc,
                        SamplePluginConstants.VENDOR_NAME,
                        SamplePluginConstants.VENDOR_URL,
                        SamplePluginConstants.PLUGIN_VERSION,
                        SamplePluginConstants.PLUGIN_FRAMEWORK_VERSION,
                        null, null, helpSet);

    return pi;
}

// これらのオブジェクトは毎回作成し直す必要がある
// その理由は、結果の所有権を放棄するので、プラグイン フレームワークにより
// 各項目にシステム ID が割り当てられるからである
// 再割り当てにより、IllegalStateException が生成される
private CategoryInfo[] getCategoryInfo(SampleBundle bundle) {

    ActionInfo checkInventoryAction =
        new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 1,
                       bundle.getString("checkInventoryName"),
                   bundle.getString("checkInventoryDesc"), ICON_BYTE_ARRAY,
                       ActionCategoryInfo.ID_NEW,
                       ActionInfo.ACTION_STATE_ALL,
                       SamplePluginConstants.CHECKINV_CLASSES);
    ActionInfo sendConfirmAction =
        new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 2,
                       bundle.getString("sendConfirmName"),
                      bundle.getString("sendConfirmDesc"), ICON_BYTE_ARRAY,
                       ActionCategoryInfo.ID_NEW,
                       ActionInfo.ACTION_STATE_ALL,
                       SamplePluginConstants.SENDCONF_CLASSES);
    ActionCategoryInfo[] actions =
      new ActionCategoryInfo[]{ checkInventoryAction, sendConfirmAction };
    CategoryInfo[] catInfo =
        new CategoryInfo[]{ new
```

```
CategoryInfo(SamplePluginConstants.PLUGIN_NAME,
                                                    0, bundle.getString("catName"),
                                                       bundle.getString("catDesc"),
                                                       ActionCategoryInfo.ID_NEW,
                                                       actions) };

        return catInfo;
    }

    private void log(String msg) {
        System.out.println("SamplePlugin: " + msg);
    }
```

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

4　プラグイン コンポーネントの定義

この章では、プラグイン コンポーネントを定義する方法について説明します。この章の内容は以下のとおりです。

- 概要
- PluginObject インタフェース
- プラグイン データの読み込みと保存
- プラグイン GUI コンポーネントの表示
- プラグインの実行
- プラグイン実行時コンテキストの使い方
- プラグイン コンポーネント値オブジェクトの定義

概要

1-8 ページの「デプロイされたプラグインを BPM が検出する方法」に説明されているように、プラグインにより BPM で以下のことが行えます。

- プラグイン実装にアクセスし、設計クライアント内のプラグインの読み込み、表示、保存を行います。
- プラグインを実行します。

この機能はプラグイン コンポーネントにより提供されます。次の表で、指定された機能をサポートするために必要なプラグイン コンポーネントの要件を示します。

表 4-1 プラグイン コンポーネントの要件

BPM が行えるようにする動作	必要な作業
XML フォーマットでプラグイン データを読み込み（解析し）、保存する。	プラグイン データ インタフェースを実装する。 たとえば、プラグイン サンプルの *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventNodeData.java を参照。
設計クライアント内でプラグイン GUI コンポーネントを表示する。	プラグイン パネル クラスを定義する。 たとえば、プラグイン サンプルの *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventNodePanel.java を参照。
プラグインを実行する。	実行時コンポーネント クラスを定義する。 たとえば、プラグイン サンプルの *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventNode.java を参照。

プラグインが着信データを読み込む（解析する）ことができるようにするには、プラグイン データ インタフェースと実行時コンポーネント クラスの両方が、親インタフェースである `com.bea.wlpi.common.plugin.PluginObject` の `load()`（解析）メソッドを実装する必要があります。

最後に、コンポーネント データを記述するためのプラグイン コンポーネント値オブジェクトを定義する必要があります。

次の節では、`PluginObject` インタフェースの説明、前述の表にリストした機能をサポートするようにプラグイン コンポーネントを定義する方法、およびプラグイン コンポーネント値オブジェクトの定義について示します。

注意： プラグイン コンポーネントの各タイプを定義するために行う必要のある手順の概略については、A-1 ページの「プラグイン コンポーネント定義のロードマップ」を参照してください。

PluginObject インタフェース

`com.bea.wlpi.common.plugin.PluginObject` インタフェースは、プラグインがプラグイン データを読み込む（解析する）ことができるようにします。

このインタフェースは、次の要素により拡張しておく必要があります。

- 4-9 ページの「プラグイン データの読み込みと保存」に説明されているプラグイン データ インタフェース
- 4-63 ページの「プラグインの実行」に説明されている実行時コンポーネント クラス

`PluginObject` インタフェースは、次の表に示す 1 つのメソッド `load()` を定義します。

表 4-2 PluginObject インタフェース メソッド

メソッド	説明
`public void load(org.xml.sax.XMLReader parser)`	XML ドキュメントからデータをロードする準備を行うようにプラグインに通知する。 メソッド パラメータの定義は次のとおり。 *parser* ― 有効な SAX パーサを示す `org.xml.sax.XMLReader` オブジェクト。データ解析時に複数のコンテンツ ハンドラを使用するには（深いネスト要素を解析するときに便利）、プラグインはこの値を保存し、指定された *parser* オブジェクトで `setContentHandler()` メソッドを呼び出すことができる。

Plug-in Manager は、XML ドキュメントの中でプラグイン セクション（たとえば `<plugin-data>` 要素）が見つかると、`load()` メソッドを呼び出します。この状況は、たとえば、Plug-in Manager が WebLogic Integration Studio の中でテンプレート、テンプレート定義、またはプラグイン コンフィグレーション XML ドキュメントを開くときに生じます。

注意： BPM DTD の詳細については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」を参照してください。

また、`startElement()` メソッドや `endElement()` メソッドなど、必要なコンテンツ ハンドラ メソッドも実装する必要があります。Plug-in Manager は、プラグインをパーサ コンテンツ ハンドラとして設定し、`<plugin-data>` 要素に達したときに、コンテンツ ハンドラに対する最初と最後の呼び出しとして、`startElement()` メソッドと `endElement()` メソッドを使用します。コンテンツ ハンドラは、介入する SAX 通知を使用して、プラグイン固有のデータを格納します。コンテンツ ハンドラ メソッドの詳細については、`org.xml.sax` Javadoc を参照してください。

プラグイン サンプルには、プラグイン コンポーネントごとに、PluginObject インタフェースを拡張し、必要なメソッドを定義する独立したファイルが用意されています。このファイルを個別に定義する必要はありません。ただし、個別に定義した場合、サンプルの中でそのファイルを共有する複数のクラスに対して 1 つの定義が提供される点が便利です。

次の節では、プラグインの完了ノードと開始ノードのために PluginObject インタフェースを実装する方法をコード例で示します。

これらの例だけでなく、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリにある次のファイルも参照してください。

ファイル名	説明されている PluginObject 実装
EventObject.java	プラグイン イベント
CheckInventoryActionObject.java	プラグイン アクション
SendConfirmActionObject.java	プラグイン アクション

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

完了ノードの例

次のコード リストで、完了ノードのために PluginObject インタフェースを実装するクラスを定義する方法を示します。このサンプル コードへの入力は、分岐ダイアログ ボックスへのユーザ応答（yes または no）です。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 4-1 完了ノードのための PluginObject インタフェースの実装

```
package com.bea.wlpi.test.plugin;

import java.io.IOException;
```

```
import com.bea.wlpi.common.plugin.PluginObject;
import org.xml.sax.*;

public class DoneObject implements PluginObject
{

   protected String yesOrNo = null;
   protected static String YESORNO_TAG = "yesorno";
   protected transient String        lastValue;

   public DoneObject()
   {
   }

   public DoneObject(String yesOrNo)
   {
      this.yesOrNo = yesOrNo;
   }

   public void load(XMLReader parser)
   {
   }

   void setYesOrNo(String decision)
   {
      yesOrNo = decision;
   }
   String getYesOrNo()
   {
      return yesOrNo;
   }

   public void setDocumentLocator(Locator locator)
   {
   }

   public void startDocument()
   throws SAXException
   {
   }

   public void endDocument()
   throws SAXException
   {
   }

   public void startPrefixMapping(String prefix, String uri)
   throws SAXException
   {
```

```
   }

   public void endPrefixMapping(String prefix)
   throws SAXException
   {
   }

   public void startElement(String namespaceURI, String localName, String
qName, Attributes atts)
   throws SAXException
   {
      lastValue = null;
   }

   public void endElement(String namespaceURI, String localName, String name)
   throws SAXException
   {
      if(name.equals(YESORNO_TAG))
         yesOrNo = lastValue;
   }

   public void characters(char[] ch, int start, int length)
   throws SAXException
   {
      String value = new String(ch, start, length);

      if(lastValue == null)
         lastValue = value;
      else
         lastValue = lastValue + value;
   }



   public void ignorableWhitespace(char[] ch, int start, int length)
   throws SAXException
   {
   }

   public void processingInstruction(String target, String data)
   throws SAXException
   {
   }

   public void skippedEntity(String name)
   throws SAXException
   {
   }
}
```

関連する次のコード例も参照してください。

- 4-14 ページの「完了ノードのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。この例は、前述の例に示したクラスを拡張します。
- 4-35 ページの「完了ノードのための PluginPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-77 ページの「完了ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。

開始ノードの例

プラグイン サンプルから抜粋した次のコード リストは、開始ノードのために `PluginObject` インタフェースを実装するクラスを定義する方法を示しています。`load()`、`startelement()`、`endelement()` の各メソッドが定義されます。この抜粋は、*`SAMPLES_HOME`*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `StartObject.java` ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-2 開始ノードのための PluginObject インタフェースの実装

```
public class StartObject implements PluginObject {
.
.
.
    public void load(XMLReader parser) {
    }
.
.
.
    public void startElement(String namespaceURI, String localName, String
qName, Attributes atts)
            throws SAXException {
        lastValue = null;
```

```
}

public void endElement(String namespaceURI, String localName, String name)
        throws SAXException {
    if (name.equals(EVENTDESC_TAG))
        eventDesc = lastValue;
}
  .
  .
  .
```

関連する次のコード例も参照してください。

- 4-17 ページの「開始ノードのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。この例は、前述の例に示したクラスを拡張します。
- 4-51 ページの「開始ノードのための PluginTriggerPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-96 ページの「開始 ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。
- 4-99 ページの「プラグイン実行時コンテキストの使い方」では、エバリュエータ式から参照できるプラグイン フィールドを定義する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

プラグイン データの読み込みと保存

XML フォーマットでプラグイン データを読み込み（解析して）、保存するには、プラグイン データ インタフェースを実装する必要があります。

注意： KEY_DATA、KEY_PANEL、KEY_RENDERER の各値を介して BPM プラグイン値オブジェクトにより定義されるプラグイン データ インタフェース クラスのそれぞれについて、クライアントでリモート クラス ロードをサ

ポートするために、引数を必要としないパブリック コンストラクタを提供する必要があります。このパブリック コンストラクタを、このクラスにより参照されるプラグイン定義されたクラスに対して提供する必要はありません。BPM プラグイン値オブジェクトの詳細については、第 2 章「プラグイン開発の基礎」を参照してください。

これは、WebLogic Integration リリース 2.1 Service Pack 1 に対する要件です。WebLogic Integration の以前のリリースを使用して生成されたクラスに対して引数のないコンストラクタを提供しない場合、そのクラスはインスタンス化されますが、クライアントとサーバ プラットフォームの間に互換性がない場合、例外が発生することがあります。

プラグインが着信データを読み込む（解析する）ことができるようにするには、プラグイン データ インタフェース クラスが、その親インタフェースである `com.bea.wlpi.common.plugin.PluginObject` の `load()`（解析）メソッドを実装する必要があります。

プラグインがそのデータを XML フォーマットで保存できるようにするには、定義されるプラグイン コンポーネントのタイプに基づいて、次の表に示すプラグイン データ インタフェースの 1 つを実装する必要があります。たとえば、Studio でテンプレート、テンプレート定義、プラグイン コンフィグレーション XML ドキュメントを保存する場合、データを XML フォーマットで保存する必要があります。

注意： 関数、メッセージ タイプ、変数タイプのプラグイン コンポーネントについては、データの読み込みと保存のためにプラグイン データ インタフェースを実装する必要はありません。

表 4-3 プラグイン データ インタフェース

定義するプラグイン	実装の必要なインタフェース
プラグイン コンポーネント	プラグイン コンポーネントがそのデータを XML フォーマットで保存できるようにするための `com.bea.wlpi.common.plugin.PluginData`。 アクションを定義するときには、このインタフェースを拡張する `PluginActionData` インタフェースを実装する必要がある。

表 4-3 プラグイン データ インタフェース （続き）

定義するプラグイン	実装の必要なインタフェース
アクション	プラグイン アクションがそのデータを XML フォーマットで保存できるようにするための com.bea.wlpi.common.plugin.PluginActionData。このクラスは、Studio の [アクション プラグイン] ダイアログ ボックスで使用され、サブアクションのための汎用サポートを提供する。 **注意：** PluginActionData は、この表の前の段で定義されている PluginData インタフェースを拡張します。

注意： BPM DTD とプラグイン固有の出力の例については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」を参照してください。

次の節では、各プラグイン データ インタフェースについて、さらに詳しく説明します。

PluginData インタフェースの実装

プラグイン コンポーネントがそのデータを XML フォーマットで保存できるようにするには、com.bea.wlpi.common.plugin.PluginData インタフェースを実装する必要があります。

注意： アクションを定義するときには、4-22 ページの「PluginActionData インタフェースの実装」に説明されているように、PluginActionData インタフェースを実装する必要があります。

次の表で、PluginData インタフェースにより定義される、実装の必要なメソッドについて説明します。

注意： PluginData インタフェース メソッドの内容は、空の場合も、単純にログに対するメッセージを返す場合もありますが、必ず実装する必要があります。

表 4-4 PluginData インタフェース メソッド

メソッド	説明
`public java.lang.Object clone()`	プラグイン データを複製する。 このメソッドは、このオブジェクトのグラフの深い（再帰的）コピーを示す `java.lang.Object` インスタンスを返す。
`public java.lang.String getPrintableData()`	プラグイン データの印刷可能な記述を取得する。 このメソッドは、通常、テンプレート定義を印刷するときに使用する。 このメソッドは、印刷可能なデータを示す `java.lang.String` オブジェクトを返す。この値は、プラグイン データ コンストラクタに指定されたロケールを使用してローカライズする必要がある。

表 4-4 PluginData インタフェース メソッド （続き）

メソッド	説明
`public java.util.List getReferencedPublishables(java.util.Map publishables)`	参照された公開可能なオブジェクトを取得する。 設計時クライアントがワークフロー定義を依存関係とともにパッケージ化できるようにする。結果のパッケージはインポートおよび実行が可能。公開可能オブジェクトには、テンプレート、テンプレート定義、ビジネス カレンダー、ビジネス オペレーション、イベント キー、リポジトリ項目などがある。これらのオブジェクトに対する参照の入ったプラグインは、このメソッドを呼び出すときにオブジェクトを宣言する必要がある。エクスポート パッケージを作成するユーザは、参照されたオブジェクトのどれをパッケージに含めるかを指定できる。 メソッド パラメータの定義は次のとおり。 *publishables* ―公開可能なすべてのオブジェクトのマップを示す `java.util.Map` オブジェクト。`com.bea.wlpi.common.Publishable` インタフェースに定義された定数をキーとする。マップ内の値は、対応するキーと一致するタイプの値オブジェクトの入った一様な `java.util.List` オブジェクトである。設計クライアントは実際のオブジェクトに対する参照のリストを求めているので、プラグインは、これらのリスト内の該当するオブジェクトを、返されるリストに追加する必要がある。 このメソッドは、`com.bea.wlpi.common.Publishable` オブジェクトのリストを返す。 公開可能オブジェクトの詳細については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「ワークフロー オブジェクトの発行」を参照。

表 4-4 PluginData インタフェース メソッド （続き）

メソッド	説明
`public void save(com.bea.wlpi.common.XMLWriter writer, int indent) throws java.io.IOException`	XML ドキュメント内のデータを保存する。 Plug-in Manager は、XML ドキュメントでプラグイン セクション（たとえば、<plugin-data> 要素）を検出すると、このメソッドを呼び出す。この状況は、たとえば、テンプレート、テンプレート定義、プラグイン コンフィグレーション XML ドキュメントを Studio で保存するときに生じる。 メソッド パラメータの定義は次のとおり。 ■ *writer* ―プラグイン データをシリアル化するために使用する XMLWriter を示す com.bea.wlpi.common.XMLWriter オブジェクト。 ■ *indent* ―インデント レベルを示す整数値。正しくインデントされた XML ドキュメントを作成するには、この値を使用する必要がある。デフォルトのインデントはスペース 2 個である。

次の節では、PluginData インタフェースの実装方法をコード例で示します。

完了ノードの例

次のコード リストで、完了ノードのために PluginData インタフェースを実装するクラスを定義する方法を示します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 4-3 完了ノードのための PluginData インタフェースの実装

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
```

```
import java.util.List;
import java.util.Map;
import org.xml.sax.*;


public class DoneNodeData extends DoneObject implements PluginData
{
    public static int count = 0;
    private int c;

    public DoneNodeData()
    {
        c=count++;
    }

    public DoneNodeData(String yesOrNo)
    {
        super(yesOrNo);
        c=count++;
    }

    public void save(XMLWriter writer, int indent) throws IOException
    {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }
}
```

関連する次のコード例も参照してください。

- 4-5 ページの「完了ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明しています。このクラスは、前述の例で示したクラスにより拡張されます。
- 4-35 ページの「完了ノードのための PluginPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-77 ページの「完了ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。

イベント ノードの例

プラグイン サンプルから抜粋した次のコード リストは、イベント ノードのために PluginData インタフェースを実装するクラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventNodeData.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-4 イベント ノードのための PluginData インタフェースの実装

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.Map;
import org.xml.sax.*;

public class EventNodeData extends EventObject implements PluginData {
    private SampleBundle bundle;

    public EventNodeData() {
        this(Locale.getDefault());
    }

    public EventNodeData(Locale lc) {
        eventDesc = SamplePluginConstants.CONFIRM_EVENT;
        bundle = new SampleBundle(lc);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, EVENTDESC_TAG, eventDesc);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return bundle.getString("confirmOrderName");
    }
```

```
    public Object clone() {
        return new EventNodeData(bundle.getLocale());
    }
}
```

関連する次のコード例も参照してください。

- *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventObject.java では、XML フォーマットでプラグイン データを読み込む方法を説明しています。このクラスは、前述の例で示したクラスにより拡張されます。
- 4-55 ページの「イベント ノードのための PluginTriggerPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-82 ページの「イベント ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

開始ノードの例

プラグイン サンプルから抜粋した次のコード リストは、開始ノードのために PluginData インタフェースを実装するクラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの StartNodeData.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-5 開始ノードのための PluginData インタフェースの実装

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
```

```
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.Map;
import org.xml.sax.*;

public class StartNodeData extends StartObject implements PluginData {
    private SampleBundle bundle;

    public StartNodeData() {
        this(Locale.getDefault());
    }

    public StartNodeData(Locale lc) {
        eventDesc = SamplePluginConstants.START_ORDER_EVENT;
        bundle = new SampleBundle(lc);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, EVENTDESC_TAG, eventDesc);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return bundle.getString("startOrderLabel");
    }

    public Object clone() {
        return new StartNodeData(bundle.getLocale());
    }
}
```

関連する次のコード例も参照してください。

- 4-8 ページの「開始ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明しています。このクラスは、前述の例で示したクラスにより拡張されます。
- 4-51 ページの「開始ノードのための PluginTriggerPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。

- 4-96 ページの「開始 ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。
- 4-99 ページの「プラグイン実行時コンテキストの使い方」では、エバリュエータ式から参照できるプラグイン フィールドを定義する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

ワークフロー テンプレート プロパティの例

次のコード リストで、ワークフロー テンプレート プロパティのために PluginData インタフェースを実装するクラスを定義する方法を示します。このコードは、分岐ダイアログ ボックスへのユーザ応答（yes または no）を読み込み、保存します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 4-6 ワークフロー テンプレート プロパティのための PluginData インタフェースの実装

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;

public class TemplatePropertiesData extends DoneObject implements PluginData {

    public TemplatePropertiesData() {
    }

    public TemplatePropertiesData(String yesOrNo){
        super(yesOrNo);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }
```

```
    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return null;
    }
}
```

関連する次のコード例も参照してください。

- 4-5 ページの「完了ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明していします。このクラスは、前述の例で示したクラスにより拡張されます。
- 4-38 ページの「ワークフロー テンプレート プロパティのための PluginPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。

ワークフロー テンプレート定義プロパティの例

次のコード リストで、ワークフロー テンプレート定義プロパティのために PluginData インタフェースを実装するクラスを定義する方法を示します。このコードは、分岐ダイアログ ボックスへのユーザ応答（yes または no）を読み込み、保存します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 4-7 ワークフロー テンプレート定義プロパティのための PluginData インタフェースの実装

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import java.io.IOException;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;
```

```
public class TemplateDefinitionPropertiesData extends DoneObject implements
PluginData
{

    public TemplateDefinitionPropertiesData()
    {
    }

    public TemplateDefinitionPropertiesData(String yesOrNo)
    {
        super(yesOrNo);
    }

    public void save(XMLWriter writer, int indent) throws IOException
    {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return null;
    }

    public Object clone() {
        return new TemplateDefinitionPropertiesData(yesOrNo);
    }
}
```

関連する次のコード例も参照してください。

- 4-5 ページの「完了ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明しています。このクラスは、前述の例で示したクラスにより拡張されます。
- 4-41 ページの「ワークフロー テンプレート定義プロパティのための PluginPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。

PluginActionData インタフェースの実装

プラグイン アクションが XML フォーマットでデータを保存できるようにするには、com.bea.wlpi.common.plugin.PluginActionData を実装する必要があります。

注意： PluginActionData インタフェースは、PluginData インタフェースを拡張します。PluginData インタフェース メソッドの詳細については、4-12 ページの「PluginData インタフェース メソッド」の表を参照してください。

次の表で、PluginActionData インタフェースにより定義される、実装の必要なメソッドについて説明します。

注意： PluginActionData インタフェース メソッドの内容は、空の場合も、単純にログに対するメッセージを返す場合もありますが、必ず実装する必要があります。

表 4-5 PluginActionData インタフェース メソッド

メソッド	説明
public java.lang.String getLabel()	アクション リストに指定されたプラグイン アクションのフォーマット済みラベルを取得する。

プラグイン サンプルから抜粋した次のコード リストは、PluginActionData インタフェースを実装するクラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの CheckInventoryActionData.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

注意： PluginActionData インタフェースを実装するクラスの定義方法を示すその他の例については、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SendConfirmationActionData.java ファイルを参照してください。

コード リスト 4-8 PluginActionData インタフェースの実装

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.XMLWriter;
import com.bea.wlpi.common.plugin.PluginData;
import com.bea.wlpi.common.plugin.PluginActionData;
import java.io.IOException;
import java.util.ResourceBundle;
import java.util.Locale;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;

public class CheckInventoryActionData extends CheckInventoryActionObject
        implements PluginActionData {
    private SampleBundle bundle;

    public CheckInventoryActionData() {
        getBundle(Locale.getDefault());
    }

    public CheckInventoryActionData(Locale lc) {
        getBundle(lc);
    }

    public CheckInventoryActionData(Locale lc, String inputVariableName,
                                    String outputVariableName) {

        super(inputVariableName, outputVariableName);

        getBundle(lc);
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, INPUTVARIABLE_TAG, inputVariableName);
        writer.saveElement(indent, OUTPUTVARIABLE_TAG, outputVariableName);
    }

    private void getBundle(Locale lc) {
        bundle = new SampleBundle(lc);
    }

    public List getReferencedPublishables(Map publishables) {
        return null;
    }
```

```
    public String getPrintableData() {
        return bundle.getString("checkInventoryDesc");
    }
    public Object clone() {

        return new CheckInventoryActionData(bundle.getLocale(),
                   new String(this.inputVariableName),
                   new String(this.outputVariableName));
    }

    public String getLabel() {
        return bundle.getString("checkInventoryDesc");
    }
}
```

関連する次のコード例も参照してください。

- *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの CheckInventoryActionObject.java では、XML フォーマットでプラグイン データを読み込む方法を説明しています。このクラスは、前述の例で示したクラスにより拡張されます。
- 4-43 ページの「PluginActionPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-70 ページの「アクションのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。
- 4-74 ページの「アクション ツリーのカスタマイズ」では、Studio 内の各種ダイアログ ボックスに表示されるアクション ツリーにリストされるアクションおよびアクション カテゴリをカスタマイズする方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

プラグイン GUI コンポーネントの表示

設計クライアントでプラグイン GUI コンポーネントを表示するには、すべてのプラグインがプラグイン パネル クラスを拡張するクラスを定義する必要があります。

たとえば、1-2 ページの「プラグイン の例 : 開始ノード」の図では、ユーザが Start Order イベントを開始ノードのトリガとして選択すると、Plug-in Manager により、プラグイン パネル クラス `StartNodePanel` がロードされます。このクラスは、引数のないコンストラクタを使用して、Studio クライアントによりインスタンス化されます。次に、Studio クライアントにより、[開始のプロパティ] ダイアログ ボックスにプラグイン GUI コンポーネントが表示されます（リモート クラス ロードの詳細については、1-10 ページの「プラグイン実装へのアクセス」を参照してください）。

注意 : `KEY_DATA`、`KEY_PANEL`、`KEY_RENDERER` の各値を介して BPM プラグイン値オブジェクトにより定義されるプラグイン GUI コンポーネント クラスのそれぞれについて、クライアントでのリモート クラス ロードをサポートするために、引数を必要としないパブリック コンストラクタを提供する必要があります。このパブリック コンストラクタを、このクラスにより参照されるプラグイン定義クラスに対して提供する必要はありません。BPM プラグイン値オブジェクトの詳細については、第 2 章「プラグイン開発の基礎」を参照してください。

これは、WebLogic Integration リリース 2.1 Service Pack 1 に対する要件です。WebLogic Integration の以前のリリースを使用して生成されたクラスに対して引数のないコンストラクタを提供しない場合、そのクラスはインスタンス化されますが、クライアントとサーバ プラットフォームの間に互換性がない場合、例外が発生することがあります。

次の表で、定義するプラグイン コンポーネントのタイプ別に、拡張の必要なプラグイン パネル クラスについて説明します。

注意 : 関数およびメッセージ タイプのプラグイン コンポーネントについては、GUI コンポーネントの表示のためにプラグイン パネル インタフェースを実装する必要はありません。

表 4-6 プラグイン パネル クラス

定義するプラグイン	拡張の必要なプラグイン パネル クラス	定義する情報
プラグイン コンポーネント	com.bea.wlpi.common.plugin.PluginPanel	設計クライアントで表示する GUI コンポーネント。 アクション、開始ノード、イベント ノード、または変数タイプが定義される場合、そのために定義されるプラグイン パネル クラス（この表の後段で定義）によりこのクラスは拡張される。
アクション	com.bea.wlpi.common.plugin.PluginActionPanel	プラグイン アクションのための GUI コンポーネント。 このクラスは、Studio の [アクション プラグイン] ダイアログ ボックスで使用され、サブアクションのための汎用サポートを提供する。 **注意：** `PluginActionPanel` は、この表の前段で定義されている `PluginPanel` クラスを拡張します。
開始ノードとイベント ノード	com.bea.wlpi.common.plugin.PluginTriggerPanel	設計クライアントで表示する開始ノードとイベント ノードの GUI コンポーネント。 このクラスは、Studio の [開始のプロパティ] ダイアログ ボックスと [イベントのプロパティ] ダイアログ ボックスで使用される。 **注意：** `PluginTriggerPanel` は、この表の前段で定義されている `PluginPanel` クラスを拡張します。

表 4-6 プラグイン パネル クラス （続き）

定義するプラグイン	拡張の必要なプラグイン パネル クラス	定義する情報
変数タイプ	com.bea.wlpi.common.plugin.PluginVariablePanel	ユーザがプラグイン変数タイプを編集する際に利用できるように設計クライアントに表示される GUI コンポーネント。 このクラスは、Studio の [変数を設定] ダイアログ ボックスにより使用される。 **注意：** PluginVariablePanel は、この表の前段で定義されている PluginPanel クラスを拡張します。
	com.bea.wlpi.common.plugin.PluginVariableRender	javax.swing.JTable のセルにプラグイン定義の変数タイプの値を表示する GUI コンポーネント。 このクラスは、Studio の [変数を設定] ダイアログ ボックスにより使用される。

次の節では、各プラグイン パネル クラスについて、さらに詳しく説明します。

PluginPanel クラスの定義

設計クライアントで表示されるプラグイン GUI コンポーネントを定義するには、com.bea.wlpi.common.plugin.PluginPanel クラスを拡張するクラスを定義する必要があります。

注意： アクション、開始ノード、イベント ノード、または変数タイプを定義する場合、4-26 ページの「プラグイン パネル クラス」の表で定義された対応するプラグイン パネル クラス（これは PluginPanel を拡張する）を拡張します。

次の表で、PluginPanel クラスにより定義されるクラス メソッドについて説明します。

注意： 最終として宣言されていないメソッドはオーバーライドできます。

表 4-7 PluginPanel クラス メソッド

メソッド	説明
`public void exceptionHandlerRenamed(java.lang.String oldName, java.lang.String newName)`	イベント ハンドラの名前を変更する。 このメソッドは、例外ハンドラへの直接参照を更新し、プラグイン パネルにより所有される `com.bea.wlpi.evaluator.Expression` オブジェクトに情報を伝える必要がある。 サブクラスがワークフロー イベント ハンドラを参照して、これらのハンドラに更新を伝える場合、サブクラスは参照が維持されるようにこのメソッドをオーバーライドする必要がある。 **注意：** デフォルトでアクションがサポートされるプラグイン ノードでは、Plug-in Manager はこの変更をアクション リスト全体に伝えます。 メソッド パラメータの定義は次のとおり。 ■ *oldName* ―古いイベント ハンドラ名を示す `java.lang.String` オブジェクト。 ■ *newName* ―新しいイベント ハンドラ名を示す `java.lang.String` オブジェクト。
`public final com.bea.wlpi.common.plugin.PluginPanelContext getContext()`	プラグイン パネルが表示される親コンポーネントを取得する。 このメソッドは、親コンポーネントを示す `com.bea.wlpi.common.plugin.PluginPanelContext` オブジェクトを返す。`PluginPanelContext` の実装方法の詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」 を参照。

表 4-7 PluginPanel クラス メソッド （続き）

メソッド	説明
`public final com.bea.wlpi.common.plugin.PluginData getData()`	プラグイン データを取得する。 このメソッドは、プラグイン データを示す `com.bea.wlpi.common.plugin.PluginData` オブジェクトを返す。`PluginData` オブジェクトの実装方法の詳細については、4-11 ページの「PluginData インタフェースの実装」 を参照。
`public java.lang.String getHelpIDString()`	プラグイン パネルのヘルプ トピック ID を取得する。 このメソッドは、ヘルプ トピック ID を示す `java.lang.String` オブジェクトを返す。
`public java.lang.String getString(java.lang.String key)`	ローカライズされた表示文字列を取得する。 リソース バンドル名は、`setResourceBundle()` メソッド （この表の後段で説明） に対する前の呼び出しにより設定されている必要がある。 メソッド パラメータの定義は次のとおり。 *key* ― リソース キーを示す `java.lang.String` オブジェクト。 このメソッドは、表示文字列を示す `java.lang.String` オブジェクトを返す。

表 4-7 PluginPanel クラス メソッド （続き）

メソッド	説明
`public java.lang.String getString(java.lang.String key, java.lang.Object[] args)`	ローカライズされた表示文字列を取得する。 リソース バンドル名は、`setResourceBundle()` メソッド（この表の後段で説明）に対する前の呼び出しにより設定されている必要がある。このメソッドは、オブジェクトの `ClassLoader` を使用して、`plugin-ejb.jar` ファイル内の指定されたリソース プロパティ ファイルから文字列リソースを取得する。 メソッド パラメータの定義は次のとおり。 ■ *key* ―リソース キーを示す `java.lang.String` オブジェクト。 ■ *args* ―メッセージ テキストに挿入する引数を示す `java.lang.Object` オブジェクト。 このメソッドは、表示文字列を示す `java.lang.String` オブジェクトを返す。
`public abstract void load()`	プラグイン パネルに対して、プラグイン データを使用してユーザ インタフェースを初期化するように指示する。 このメソッドは、`getData()` を呼び出してプラグイン データにアクセスし、対応するプラグイン クラスに結果を送信し、適切な get メソッドを呼び出して、表示値を取り出す。 Plug-in Manager は、このメソッドがモードの表示サイクルあたり 1 回だけ呼び出されるように管理する。 **注意：** プラグインは、このメソッドを呼び出してはなりません。

表 4-7 PluginPanel クラス メソッド （続き）

メソッド	説明
`public boolean referencesExceptionHandler(java.lang.String handler)`	プラグインが、指定されたイベント ハンドラを参照しているかどうかをチェックする。 このメソッドは、プラグイン パネル クラスが指定された例外ハンドラに対して保持している直接参照を名前でチェックする必要がある。 サブクラスがワークフロー イベント ハンドラを参照する場合、参照されたイベント ハンドラを誤って削除することを防止するため、そのサブクラスはこのメソッドをオーバーライドする必要がある。 **注意：** デフォルトでアクションがサポートされるプラグイン ノードでは、Plug-in Manager はこの変更をアクション リスト全体に伝えます。 メソッド パラメータの定義は次のとおり。 *handler* ―イベント ハンドラ名を示す `java.lang.String` オブジェクト。 このメソッドは、プラグイン パネルが指定されたイベント ハンドラを参照する場合はブール値の `true` を、参照しない場合は `false` を返す。

表 4-7 PluginPanel クラス メソッド （続き）

メソッド	説明
`public boolean referencesVariable(java.lang.String variable)`	プラグイン パネルが、指定された変数を参照しているかどうかをチェックする。 このメソッドは、プラグイン パネル クラスが指定された変数に対して保持している直接参照を名前でチェックする必要がある。 サブクラスが、名前により直接、または式により間接的に、ワークフロー変数を参照する場合、参照された変数を誤って削除することを防止するため、そのサブクラスはこのメソッドをオーバーライドする必要がある。 **注意：** デフォルトでアクションがサポートされるプラグイン ノードでは、Plug-in Manager はこの変更をアクション リスト全体に伝えます。 メソッド パラメータの定義は次のとおり。 *`variable`* ―変数を示す `java.lang.String` オブジェクト。 このメソッドは、プラグイン パネルが指定された変数を参照する場合はブール値の `true` を、参照しない場合は `false` を返す。

表 4-7 PluginPanel クラス メソッド （続き）

メソッド	説明
`public final void setContext(com.bea.wlpi.common.plugin.PluginPanelContext context, com.bea.wlpi.common.plugin.PluginData data)`	プラグイン パネルの操作コンテキストを設定する。 Plug-in Manager は、設計クライアント ダイアログ ボックスにプラグイン パネルを追加する前に、このメソッドを呼び出す。このメソッドは、対応するメンバー変数にオーナ パラメータとデータ パラメータを格納する。 **注意：** プラグインは、このメソッドを呼び出してはなりません。 メソッド パラメータの定義は次のとおり。 ■ *context* ―プラグイン パネルを表示する設計クライアント ダイアログ ボックス コンテンツを示す `com.bea.wlpi.common.plugin.PluginPanelContext` オブジェクト。`PluginPanelContext` オブジェクトの実装方法の詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照。 ■ *data* ―プラグイン データを示す `com.bea.wlpi.common.plugin.PluginData` オブジェクト。`PluginData` オブジェクトの実装方法の詳細については、4-11 ページの「PluginData インタフェースの実装」を参照。
`public void setResourceBundle(java.lang.String bundleName)`	文字列とメッセージをローカライズするときに使用するリソース バンドルを設定する。 メソッド パラメータの定義は次のとおり。 *bundleName* ―リソース バンドルの名前を示す `java.lang.String` オブジェクト。

表 4-7 PluginPanel クラス メソッド （続き）

メソッド	説明
`public abstract boolean validateAndSave()`	GUI コントロール値を検証し、その値を保存するように、プラグイン パネルに指示する。 このメソッドは、`getData()` を呼び出してプラグイン データにアクセスし、対応するプラグイン クラスに結果を送信し、適切な set メソッドを呼び出して、表示値を保存する。 このメソッドは、パネルが検証され、保存された場合はブール値の `true` を、そうでない場合は `false` を返す。
`public void variableRenamed(java.lang.String` *`oldName`*`, java.lang.String` *`newName`*`)`	変数の名前を変更する。 このメソッドは、変数への直接参照を更新し、プラグイン パネルにより所有される `com.bea.wlpi.evaluator.Expression` オブジェクトに情報を伝える必要がある。これは、`variableRenamed()` メソッドを呼び出してから、`toString()` メソッドを呼び出して、更新された式テキストを取得することによって行える。 サブクラスが、名前により直接、または式により間接的に、ワークフロー変数を参照する場合、参照が維持されるように、そのサブクラスはこのメソッドをオーバーライドする必要がある。 **注意：** デフォルトでアクションがサポートされるプラグイン ノードでは、Plug-in Manager はこの変更をアクション リスト全体に伝えます。 メソッド パラメータの定義は次のとおり。 ■ *`oldName`* ―古い変数名を示す `java.lang.String` オブジェクト。 ■ *`newName`* ―新しい変数名を示す `java.lang.String` オブジェクト。

次の節では、`PluginPanel` クラスの定義方法をコード例で示します。

完了ノードの例

次のコード リストで、完了ノードのために PluginPanel クラスを定義する方法を示します。このコードは、[完了のプロパティ] ダイアログ ボックス内に分岐ダイアログ ボックス（yes または no）を表示します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

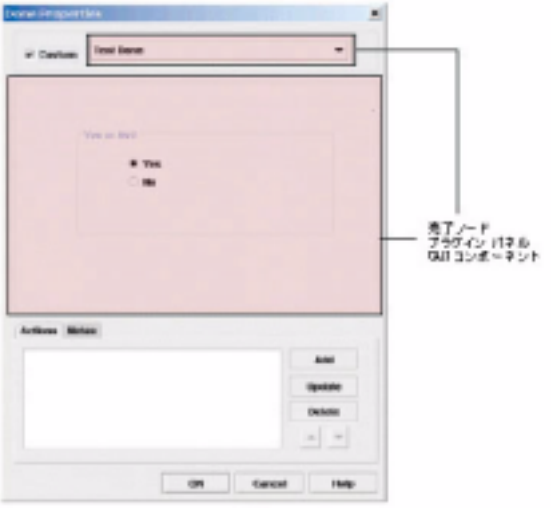
コード リスト 4-9 完了ノードのための PluginPanel クラスの定義

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class DoneNodePanel extends PluginPanel
{

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

   public DoneNodePanel()
   {
      super(Locale.getDefault(), "jackolantern");
      setLayout(null);
      setBounds(12,12,420,300);
      setPreferredSize(new Dimension(420,300));
      ButtonPanel.setBorder(titledBorder);
      ButtonPanel.setLayout(null);
      add(ButtonPanel);
      ButtonPanel.setBounds(72,60,300,144);
      YesButton.setText("Yes");
      YesButton.setSelected(true);
```

```
      YesNoButtonGroup.add(YesButton);
      ButtonPanel.add(YesButton);
      YesButton.setBounds(60,36,46,23);
      NoButton.setText("No");
      YesNoButtonGroup.add(NoButton);
      ButtonPanel.add(NoButton);
      NoButton.setBounds(60,60,46,23);
      titledBorder.setTitle("Yes or No?");
   }

public void load() {

    DoneNodeData myData = (DoneNodeData)getData();
    if(myData != null) {
        if(myData.getYesOrNo() != null &&
myData.getYesOrNo().equals(TestPluginConstants.DONE_NO)) {
            NoButton.setSelected(true);
        } else {
            YesButton.setSelected(true);
        }
    }
}

   public boolean validateAndSave()
   {
    DoneNodeData myData = (DoneNodeData)getData();
    if(myData != null) {
        if(YesButton.isSelected()) {
            myData.setYesOrNo(TestPluginConstants.DONE_YES);
        } else {
            myData.setYesOrNo(TestPluginConstants.DONE_NO);
        }
    }

      return true;
   }
            }
```

次の図に、結果の `PluginPanel` GUI コンポーネントを示します。

図 4-1 完了ノードのための PluginPanel GUI コンポーネント



関連する次のコード例も参照してください。

- 4-5 ページの「完了ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明しています。
- 4-14 ページの「完了ノードのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。このクラスは、`PluginObject` クラスを拡張します。
- 4-77 ページの「完了ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。

ワークフロー テンプレート プロパティの例

次のコード リストで、ワークフロー テンプレート プロパティのために `PluginPanel` クラスを定義する方法を示します。このコードは、[ワークフロー テンプレート プロパティ] ダイアログ ボックス内に分岐ダイアログ ボックス（yes または no）を表示します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

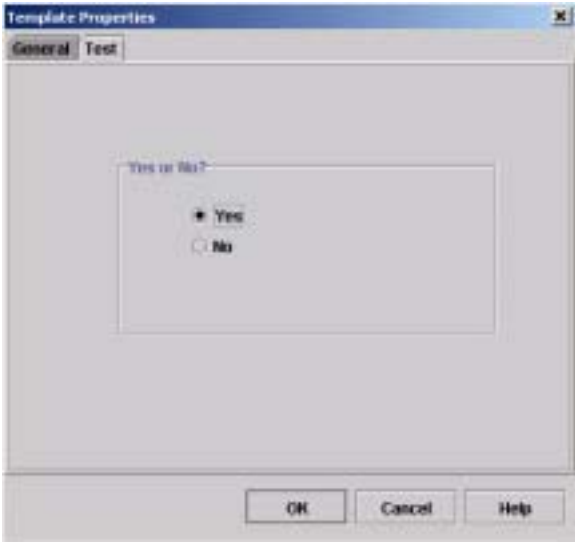
コード リスト 4-10 ワークフロー テンプレート プロパティのための PluginPanel クラスの定義

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class TemplatePropertiesPanel extends PluginPanel
{

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

   public TemplatePropertiesPanel()
   {
      super(Locale.getDefault(), "stpatty");
      setLayout(null);
      setBounds(12,12,420,300);
      ButtonPanel.setBorder(titledBorder);
      ButtonPanel.setLayout(null);
      add(ButtonPanel);
      ButtonPanel.setBounds(72,60,300,144);
      YesButton.setText("Yes");
      YesButton.setSelected(true);
      YesNoButtonGroup.add(YesButton);
      ButtonPanel.add(YesButton);
      YesButton.setBounds(60,36,46,23);
      NoButton.setText("No");
      YesNoButtonGroup.add(NoButton);
      ButtonPanel.add(NoButton);
      NoButton.setBounds(60,60,46,23);
      titledBorder.setTitle("Yes or No?");
   }

public void load() {
```

```
    TemplatePropertiesData myData = (TemplatePropertiesData)getData();
    if(myData != null) {
        if(myData.getYesOrNo() != null &&
myData.getYesOrNo().equals(TestPluginConstants.DONE_NO)) {
            NoButton.setSelected(true);
        } else {
            YesButton.setSelected(true);
        }
    }
}

   public boolean validateAndSave()
   {
    TemplatePropertiesData myData = (TemplatePropertiesData)getData();
    if(myData != null) {
        if(YesButton.isSelected()) {
            myData.setYesOrNo(TestPluginConstants.DONE_YES);
        } else {
            myData.setYesOrNo(TestPluginConstants.DONE_NO);
        }
    }

      return true;
   }
```

次の図に、結果の PluginPanel GUI コンポーネントを示します。

図 4-2 ワークフロー テンプレート プロパティのための PluginPanel GUI コンポーネント



関連する次のコード例も参照してください。

- 4-5 ページの「完了ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む `PluginObject` クラスを定義する方法を説明しています。
- 4-19 ページの「ワークフロー テンプレート プロパティのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。このクラスは、`PluginObject` クラスを拡張します。

ワークフロー テンプレート定義プロパティの例

次のコード リストで、ワークフロー テンプレート定義プロパティのために `PluginPanel` クラスを定義する方法を示します。このコードは、[ワークフロー テンプレート定義プロパティ] ダイアログ ボックス内に分岐ダイアログ ボックス（yes または no）を表示します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 4-11 ワークフロー テンプレート定義プロパティのための PluginPanel クラスの定義

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class TemplateDefinitionPropertiesPanel extends PluginPanel
{

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

   public TemplateDefinitionPropertiesPanel()
   {
      super(Locale.getDefault(), "valentine");
      setLayout(null);
      setBounds(12,12,420,300);
      ButtonPanel.setBorder(titledBorder);
      ButtonPanel.setLayout(null);
      add(ButtonPanel);
      ButtonPanel.setBounds(72,60,300,144);
      YesButton.setText("Yes");
      YesButton.setSelected(true);
      YesNoButtonGroup.add(YesButton);
      ButtonPanel.add(YesButton);
      YesButton.setBounds(60,36,46,23);
      NoButton.setText("No");
      YesNoButtonGroup.add(NoButton);
      ButtonPanel.add(NoButton);
      NoButton.setBounds(60,60,46,23);
      titledBorder.setTitle("Yes or No?");
   }

public void load() {
```

```
    TemplateDefinitionPropertiesData myData =
(TemplateDefinitionPropertiesData)getData();
    if(myData != null) {
        if(myData.getYesOrNo() != null &&
myData.getYesOrNo().equals(TestPluginConstants.DONE_NO)) {
            NoButton.setSelected(true);
        } else {
            YesButton.setSelected(true);
        }
    }
}

   public boolean validateAndSave()
   {
    TemplateDefinitionPropertiesData myData =
(TemplateDefinitionPropertiesData)getData();
    if(myData != null) {
        if(YesButton.isSelected()) {
            myData.setYesOrNo(TestPluginConstants.DONE_YES);
        } else {
            myData.setYesOrNo(TestPluginConstants.DONE_NO);
        }
    }

      return true;
}
```

次の図に、結果の `PluginPanel` GUI コンポーネントを示します。

図 4-3 ワークフロー テンプレート定義プロパティのための PluginPanel GUI コンポーネント

関連する次のコード例も参照してください。

- 4-5 ページの「完了ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む `PluginObject` クラスを定義する方法を説明しています。
- 4-20 ページの「ワークフロー テンプレート定義プロパティのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。 このクラスは、`PluginObject` クラスを拡張します。

PluginActionPanel クラスの定義

プラグイン アクションを定義するときに設計クライアントに表示される GUI コンポーネントを定義するには、`com.bea.wlpi.common.plugin.PluginActionPanel` クラスを拡張するクラス

を定義する必要があります。Studio では、PluginActionPanel クラスは [アクション プラグイン] ダイアログ ボックスにより使用されます。このダイアログ ボックスは、サブアクションに対する汎用サポートを提供します。

PluginActionPanel クラスで定義される追加メソッドはありません。

注意： PluginActionPanel クラスは、PluginPanel クラスを拡張します。PluginPanel クラス メソッドの詳細については、4-28 ページの「PluginPanel クラス メソッド」の表を参照してください。

プラグイン サンプルから抜粋した次のコード リストは、PluginActionPanel クラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの CheckInventoryActionPanel.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

注意： PluginActionPanel クラスの定義方法を示すその他の例については、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SendConfirmationActionPanel.java ファイルを参照してください。

コード リスト 4-12 PluginActionPanel クラスの定義

```
package com.bea.wlpi.tour.po.plugin;

import java.awt.*;
import javax.swing.*;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.VariableInfo;
import com.bea.wlpi.common.plugin.PluginActionPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;

public class CheckInventoryActionPanel extends PluginActionPanel {
    private JLabel inputLabel = new JLabel();
    private JLabel outputLabel = new JLabel();
    private JComboBox inputComboBox = new JComboBox();
    private JComboBox outputComboBox = new JComboBox();
    private List variables = null;

    public CheckInventoryActionPanel() {
        this(Locale.getDefault());
    }
```

```
public CheckInventoryActionPanel(Locale lc) {

    super(lc, "checkinventory");

    setLayout(null);
    setBounds(12, 12, 420, 210);
    setPreferredSize(new Dimension(420, 210));
    add(inputLabel);
    inputLabel.setBounds(12, 48, 96, 24);
    add(outputLabel);
    outputLabel.setBounds(12, 108, 166, 24);
    add(inputComboBox);
    inputComboBox.setBounds(190, 48, 212, 24);
    inputComboBox.setEditable(true);
    add(outputComboBox);
    outputComboBox.setBounds(190, 108, 212, 24);
    outputComboBox.setEditable(true);
}

public void load() {

    setResourceBundle("com.bea.wlpi.tour.po.plugin.SamplePlugin");
    inputLabel.setText(getString("inputLabel"));
    outputLabel.setText(getString("outputLabel"));

    CheckInventoryActionData myData = (CheckInventoryActionData)getData();

    variables = getContext().getVariableList(VariableInfo.TYPE_INT);

    // load はこのパネルの表示前に毎回呼び出される。
    // 現在定義されている変数を入れる前に、
    // コンボ ボックスから必ず項目を削除すること。
    inputComboBox.removeAllItems();

    String inputVar = myData.getInputVariableName();
    int n = variables == null ? 0 : variables.size();

    for (int i = 0; i < n; i++) {
        VariableInfo varInfo = (VariableInfo)variables.get(i);

        inputComboBox.addItem(varInfo.getName());

        if (inputVar != null && inputVar.equals(varInfo.getName())) {
            inputComboBox.setSelectedIndex(i);
        }
    }
```

```
        if (inputVar == null && n > 0)
            inputComboBox.setSelectedIndex(0);

        outputComboBox.removeAllItems();

        String outputVar = myData.getOutputVariableName();

        for (int i = 0; i < n; i++) {


            VariableInfo varInfo = (VariableInfo)variables.get(i);

            outputComboBox.addItem(varInfo.getName());

            if (outputVar != null && outputVar.equals(varInfo.getName())) {
                outputComboBox.setSelectedIndex(i);
            }
        }

        if (outputVar == null && n > 0)
            outputComboBox.setSelectedIndex(0);
    }

    public boolean validateAndSave() {

        CheckInventoryActionData myData = (CheckInventoryActionData)getData();
        String input = (String)inputComboBox.getEditor().getItem();

        try {
            VariableInfo varInfo = getContext().checkVariable(input,
                                        new String[]{ VariableInfo.TYPE_INT });

            if (varInfo == null)
                return false;

            if (!(varInfo.getType().equals(VariableInfo.TYPE_INT))) {

JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                              getString("Message_100"),
                                              getString("variableErrorTitle"),
                                              JOptionPane.ERROR_MESSAGE);

                return false;
            }

            input = varInfo.getName();
        } catch (Exception e) {
          JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                          e.getLocalizedMessage(),
```

```
                                        getString("variableErrorTitle"),
                                        JOptionPane.ERROR_MESSAGE);

            return false;
        }

        String output = (String)outputComboBox.getEditor().getItem();

        try {
            VariableInfo varInfo = getContext().checkVariable(output,


                                    new String[]{ VariableInfo.TYPE_INT });

            if (varInfo == null)
                return false;

            if (!(varInfo.getType().equals(VariableInfo.TYPE_INT))) {

JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                            getString("Message_100"),
                                            getString("variableErrorTitle"),
                                            JOptionPane.ERROR_MESSAGE);

                return false;
            }

            output = varInfo.getName();
        } catch (Exception e) {
          JOptionPane.showMessageDialog(SwingUtilities.windowForComponent(this),
                                        e.getLocalizedMessage(),
                                        getString("variableErrorTitle"),
                                        JOptionPane.ERROR_MESSAGE);

            return false;
        }

        if (input == null || output == null) {
            JOptionPane.showMessageDialog(null, getString("Message_101"),
                                        getString("invalidDataTitle"),
                                        JOptionPane.ERROR_MESSAGE);

            return false;
        }

        myData.setInputVariableName(input);
        myData.setOutputVariableName(output);

        return true;
```

```
    }
}
```

次の図に、結果の PluginActionPanel GUI コンポーネントを示します。

図 4-4 PluginActionPanel GUI コンポーネント

関連する次のコード例も参照してください。

- *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの CheckInventoryActionObject.java では、XML フォーマットでプラグイン データを読み込む方法を説明しています。
- 4-23 ページの「PluginActionData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。このクラスは、CheckInventoryActionObject クラスを拡張します。
- 4-65 ページの「アクションのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。
- 4-74 ページの「アクション ツリーのカスタマイズ」では、Studio 内の各種ダイアログ ボックスに表示されるアクション ツリーにリストされるアクションおよびアクション カテゴリをカスタマイズする方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

PluginTriggerPanel クラスの定義

プラグイン開始ノードまたはイベント ノードを定義するときに設計クライアントに表示される GUI コンポーネントを定義するには、`com.bea.wlpi.common.plugin.PluginTriggerPanel` クラスを拡張するクラスを定義する必要があります。Studio では、開始ノードとイベント ノードの `PluginTriggerPanel` クラスは、それぞれ [開始のプロパティ] ダイアログ ボックスと [イベントのプロパティ] ダイアログ ボックスで使用されます。

次の表で、`PluginTriggerPanel` クラスにより定義されるクラス メソッドについて説明します。

注意： `PluginTriggerPanel` クラスは、`PluginPanel` クラスを拡張します。`PluginPanel` クラス メソッドの詳細については、4-28 ページの「PluginPanel クラス メソッド」の表を参照してください。

表 4-8 PluginTriggerPanel クラス メソッド

メソッド	説明
`public java.lang.String getEventDescriptor()`	プラグイン イベント記述子の特性を示す文字列を取得する。 イベント記述子は、プラグイン ノードが監視するデータを定義する。 式エバリュエータは、`com.bea.wlpi.common.plugin.FieldInfo` オブジェクトにより指定される、関係する `com.bea.wlpi.common.plugin.PluginField` 実装クラスのインスタンスにイベント記述子を渡す。`FieldInfo` オブジェクトは、親である `com.bea.wlpi.common.plugin.StartInfo` オブジェクトまたは `com.bea.wlpi.common.plugin.EventInfo` オブジェクトにより提供される。 このメソッドは、イベント記述子を示す `java.lang.String` オブジェクトを返す。デフォルト実装では、このメソッドは null を返す。 プラグイン固有の外部イベントにアクセスするためのプラグイン フィールドを定義する方法については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照。

表 4-8 PluginTriggerPanel クラス メソッド （続き）

メソッド	説明
`public java.lang.String[] getFields()`	イベントと関連付けられたフィールド名のリストを取得する（既知の場合）。 このリストが null でない場合、関連付けられた [開始のプロパティ] ダイアログ ボックスまたは [イベントのプロパティ] ダイアログ ボックスは、Expression Builder にこのリストを渡す。 このメソッドは、フィールド名を示す `java.lang.String` オブジェクトの配列を返す。デフォルト実装では、このメソッドは null を返す。 プラグイン固有の外部イベントにアクセスするためのプラグイン フィールドを定義する方法については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照。

次の節では、`PluginTriggerPanel` クラスの定義方法をコード例で示します。

開始ノードの例

プラグイン サンプルから抜粋した次のコード リストは、開始ノードのために `PluginTriggerPanel` クラスを定義する方法を示しています。この抜粋は、*`SAMPLES_HOME`*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `StartNodePanel.java` ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-13 開始ノードのための PluginTriggerPanel クラスの定義

```
package com.bea.wlpi.tour.po.plugin;
```

```
import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginTriggerPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.common.VariableInfo;

public class StartNodePanel extends PluginTriggerPanel {
    private JLabel StartOrderLabel = new JLabel();
    private JTextArea StartOrderText = new JTextArea();

    public StartNodePanel() {
        this(Locale.getDefault());
    }

    public StartNodePanel(Locale lc) {

        super(lc, "startorder");

        setLayout(null);
        setBounds(12, 12, 420, 240);
        setPreferredSize(new Dimension(420, 240));
        add(StartOrderLabel);
        StartOrderLabel.setFont(new Font("Dialog", Font.BOLD, 16));
        StartOrderLabel.setBounds(120, 12, 156, 24);
        StartOrderText.setLineWrap(true);
        StartOrderText.setWrapStyleWord(true);
        StartOrderText.setEditable(false);
        add(StartOrderText);
        StartOrderText.setBounds(30, 48, 348, 144);
    }

    public void load() {

        setResourceBundle("com.bea.wlpi.tour.po.plugin.SamplePlugin");
        StartOrderLabel.setText(getString("startOrderLabel"));
        StartOrderText.setText(getString("startOrderText"));
    }

    public boolean validateAndSave() {
        return true;
    }

    public String[] getFields() {
        return SamplePluginConstants.ORDER_FIELDS;
```

```
    }

    public String getEventDescriptor() {
        return SamplePluginConstants.START_ORDER_EVENT;
    }
}
```

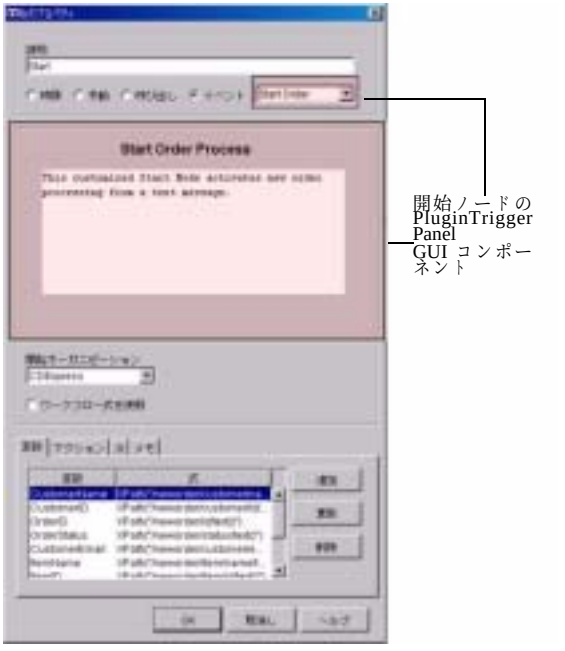
START_ORDER_EVENT フィールド要素値と ORDER_FIELDS フィールド要素値が、SamplePluginConstants.java クラス ファイル内に含まれます。これらは、プラグイン開始ノードのイベント記述子とフィールド要素値を次のように定義します。

```
final static String START_ORDER_EVENT = "startOrder";
final static String[] ORDER_FIELDS = {
"CustomerName", "CustomerID", "OrderStatus", "OrderID",
"CustomerEmail", "ItemName", "ItemID", "ItemQuantity",
"CustomerState"
};
```

プラグイン固有の外部イベントにアクセスするためのプラグイン フィールドを定義する方法については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照してください。

次の図に、結果の PluginTriggerPanel GUI コンポーネントを示します。

図 4-5 開始ノードのための PluginTriggerPanel GUI コンポーネント



関連する次のコード例も参照してください。

- 4-8 ページの「開始ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明しています。
- 4-17 ページの「開始ノードのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。
- 4-96 ページの「開始 ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。
- 4-99 ページの「プラグイン実行時コンテキストの使い方」では、エバリュエータ式から参照できるプラグイン フィールドを定義する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

イベント ノードの例

プラグイン サンプルから抜粋した次のコード リストは、イベント ノードのために PluginTriggerPanel クラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventNodePanel.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-14 イベント ノードのための PluginTriggerPanel クラスの定義

```
package com.bea.wlpi.tour.po.plugin;

import java.awt.*;
import javax.swing.*;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginTriggerPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.common.VariableInfo;

public class EventNodePanel extends PluginTriggerPanel {
    private JLabel confirmOrderLabel = new JLabel();
    private JTextArea confirmOrderText = new JTextArea();

    /**
     * 新しい EventNodePanel を作成する
     */
    public EventNodePanel() {
        this(Locale.getDefault());
    }

    public EventNodePanel(Locale lc) {

        super(lc, "confirmevent");

        setLayout(null);
        setBounds(12, 12, 420, 240);
        setPreferredSize(new Dimension(420, 240));
        add(confirmOrderLabel);
        confirmOrderLabel.setFont(new Font("Dialog", Font.BOLD, 16));
        confirmOrderLabel.setBounds(144, 12, 120, 24);
        confirmOrderText.setRequestFocusEnabled(false);
        confirmOrderText.setLineWrap(true);
        confirmOrderText.setWrapStyleWord(true);
        confirmOrderText.setEditable(false);
        add(confirmOrderText);
```

```
        confirmOrderText.setBounds(30, 48, 348, 144);
    }

    public void load() {

        setResourceBundle("com.bea.wlpi.tour.po.plugin.SamplePlugin");
        confirmOrderLabel.setText(getString("confirmOrderLabel"));
        confirmOrderText.setText(getString("confirmOrderText"));
    }

    public boolean validateAndSave() {

        // このパネル上には、ユーザ入力を受け取る UI コントロールがない。
        // したがって、このメソッドで行うことは何もない。
        return true;
    }

    public String[] getFields() {
        return SamplePluginConstants.CONFIRM_FIELDS;
    }

    public String getEventDescriptor() {
        return SamplePluginConstants.CONFIRM_EVENT;
    }
}
```

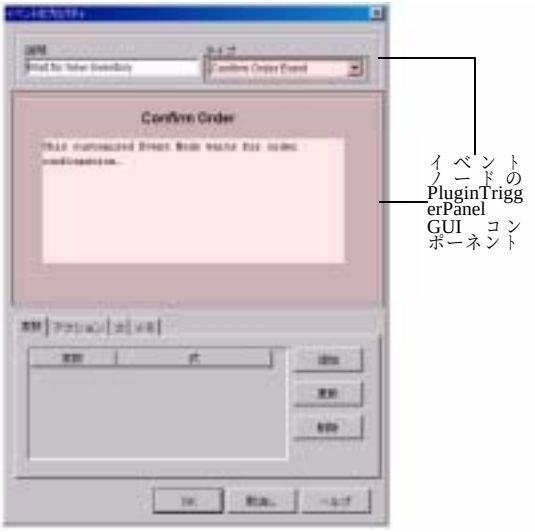
CONFIRM_EVENT と CONFIRM_FIELD が、SamplePluginConstants.java クラス内に含まれます。これらは、プラグイン イベント ノードのイベント記述子とフィールド要素値を次のように定義します。

```
final static String CONFIRM_EVENT = "confirmOrder";
final static String[] CONFIRM_FIELDS = { "Status", "TotalPrice" };
```

プラグイン固有の外部イベントにアクセスするためのプラグイン フィールドを定義する方法については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照してください。

次の図に、結果の PluginTriggerPanel GUI コンポーネントを示します。

図 4-6 イベント ノードのための PluginTriggerPanel GUI コンポーネント



関連する次のコード例も参照してください。

- *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventObject.java では、プラグイン データを読み込むために PluginObject インタフェースを実装する方法を説明しています。
- 4-16 ページの「イベント ノードのための PluginData インタフェースの実装」では、プラグイン データを読み込み、保存する方法を説明しています。このクラスは、EventObject クラスを拡張します。
- 4-82 ページの「イベント ノードのための実行時コンポーネント クラスの定義」では、プラグインに関する実行情報を定義する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

PluginVariablePanel クラスの定義

ユーザがプラグイン変数タイプを編集できるようにするためのプラグイン変数を定義するときに設計クライアントに表示される GUI コンポーネントを定義するには、`com.bea.wlpi.common.plugin.PluginVariablePanel` クラスを拡張するクラスを定義する必要があります。Studio では、`PluginVariablePanel` クラスは [変数を設定] ダイアログ ボックスにより使用されます。

次の表では、`PluginVariablePanel` クラスにより定義されるクラス メソッドについて説明します。

注意： `PluginVariablePanel` クラスは、`PluginPanel` クラスを拡張します。`PluginPanel` クラス メソッドの詳細については、4-28 ページの「PluginPanel クラス メソッド」の表を参照してください。

表 4-9 PluginVariablePanel クラス メソッド

メソッド	説明
`public final java.lang.Object getVariableValue()`	プラグイン変数の値を取得する。このメソッドは、変数値を示す `java.lang.Object` オブジェクトを返す。

表 4-9 PluginVariablePanel クラス メソッド （続き）

メソッド	説明
`public final void setContext(java.lang.Object variableValue)`	プラグイン パネルの操作コンテキストを設定する。 Plug-in Manager は、[変数を設定] ダイアログ ボックスにプラグイン変数パネルを追加する前に、このメソッドを呼び出す。 **注意：** プラグインは、このメソッドを呼び出してはなりません。 メソッド パラメータの定義は次のとおり。 *variableValue* ―変数値を示す `java.lang.Object` オブジェクト。この表の前段で説明した `getVariableValue()` メソッドを使用して、変数値を取得できる。

次のコード リストで、`PluginVariablePanel` クラスの定義方法を示します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

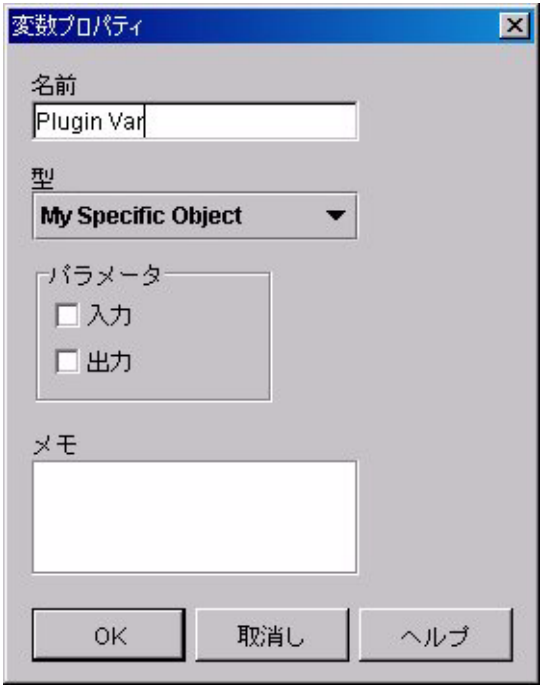
コード リスト 4-15 PluginVariablePanel クラスの定義

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;
import com.bea.wlpi.common.plugin.PluginVariablePanel;

public class VariablePanel extends PluginVariablePanel {
    JTextField highField, lowField;
```

```
    public VariablePanel() {
        super(Locale.getDefault(), "augustus");
        setLayout(null);
        setBounds(12,12,420,60);
        highField = new JTextField();
        highField.setLocation(20, 10);
        highField.setSize(300, 20);
        add(highField);
        lowField = new JTextField();
        lowField.setLocation(20, 40);
        lowField.setSize(300, 20);
        add(lowField);
    }

    public void load() {
        if (variableValue != null) {
            highField.setText(((MySpecificObject)variableValue).getHigh());
            lowField.setText(((MySpecificObject)variableValue).getLow());
        } else {
            highField.setText("");
            lowField.setText("");
        }
    }

    public boolean validateAndSave() {
        try {
            variableValue = new MySpecificObject(lowField.getText(),
highField.getText());
        } catch (Exception e) {
            return false;
        }
        return true;
    }
}
```

次の図に、結果の `PluginVariablePanel` GUI コンポーネントを示します。

図 4-7 PluginVariablePanel GUI コンポーネント



関連するリスト例として、4-62 ページの「PluginVariableRenderer クラスの定義」を参照してください。`javax.swing.JTable` のセルにプラグイン定義の変数タイプの値を表示する方法について説明しています。

PluginVariableRenderer クラスの定義

`javax.swing.JTable` のセルにプラグイン定義の変数タイプの値を表示するには、`com.bea.wlpi.common.plugin.PluginVariableRenderer` インタフェースを実装します。

注意： このインタフェースを実装するクラスは、`java.awt.Component` のサブクラスとする必要があります。

次の表で、実装の必要な PluginVariableRenderer インタフェース メソッドについて説明します。

表 4-10 PluginVariableRenderer インタフェース メソッド

メソッド	説明
public void setValue(java.lang.Object *value*)	表示する変数を設定する。 メソッド パラメータの定義は次のとおり。 *value* ―表示する変数値を示す java.lang.Object オブジェクト。この値は、null、または対応する com.bea.wlpi.common.plugin.VariableTypeInfo オブジェクトで宣言されたクラスのインスタンスのいずれかとすることができる。

次のコード リストで、javax.swing.JTable のセルにプラグイン定義の変数タイプの値を表示する方法を示します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 4-16 PluginVariableRenderer クラスの定義

```
package com.bea.wlpi.test.plugin;

import java.io.Serializable;
import javax.swing.JLabel;
import com.bea.wlpi.common.plugin.PluginVariableRenderer;

public class VariableRenderer extends JLabel implements PluginVariableRenderer,
Serializable {
    public VariableRenderer() {
    }

    public void setValue(Object value) {
        if (value == null)
            setText("null");
        else
            setText(value.toString());
```

```
    }
}
```

4-59 ページの「PluginVariablePanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。

プラグインの実行

プラグインを実行するには、プラグインのための実行時コンポーネントを定義する必要があります。

次の表で、作成するプラグイン コンポーネントのタイプ別に、実装の必要なプラグイン コンポーネント インタフェースについて説明します。プラグインが着信データを読み込む（解析する）ことができるようにするには、実行時コンポーネント クラスが、その親インタフェースである `com.bea.wlpi.common.plugin.PluginObject` の `load()`（解析）メソッドを実装する必要があります。

注意： プラグイン コンポーネントのうち、変数タイプ、ワークフロー テンプレート プロパティ、ワークフロー テンプレート定義プロパティについては、実行情報を定義する必要はありません。

表 4-11 プラグイン実行時コンポーネント インタフェース

定義するプラグイン	実装の必要なインタフェース	定義する情報
アクション	`com.bea.wlpi.server.plugin.PluginAction`	プラグイン アクション実行情報。 **注意：** プラグイン アクションをサポートするには、Studio 内の各種ダイアログ ボックスに表示されるアクション ツリーにリストされるアクションおよびアクション カテゴリもカスタマイズする必要があります。

表 4-11 プラグイン実行時コンポーネント インタフェース （続き）

定義するプラグイン	実装の必要なインタフェース	定義する情報
完了ノード	com.bea.wlpi.server.plugin.PluginDone	プラグイン 完了ノード実行情報。 **注意：** PluginDone インタフェースは、com.bea.wlpi.server.plugin.PluginTemplateNode インタフェースを拡張します。詳細については、4-97 ページの「PluginTemplateNode インタフェース」を参照してください。
イベント ノード	com.bea.wlpi.server.plugin.PluginEvent	プラグイン イベント ノード実行情報。
関数	com.bea.wlpi.common.plugin.PluginFunction	新しいエバリュエータ関数情報。
メッセージ タイプ	com.bea.wlpi.server.plugin.PluginField	プラグイン固有のメッセージ タイプ。
開始ノード	com.bea.wlpi.server.plugin.PluginStart2	プラグイン開始ノード実行情報。 **注意：** PluginStart2 インタフェースは、com.bea.wlpi.server.plugin.PluginTemplateNode インタフェースを拡張します。詳細については、4-97 ページの「PluginTemplateNode インタフェース」を参照してください。

注意： 実行時に、Plug-in Manager により渡されるコンテキスト インタフェースを使用して、関連付けられたプラグインのための実行時コンテキストとサービスにアクセスできます。コンテキスト インタフェースの詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照してください。

次の節では、それぞれのプラグイン実行時コンポーネント クラスを定義する方法について説明します。

アクションのための実行時コンポーネント クラスの定義

プラグイン アクションのための実行時コンポーネント クラスを定義する手順は、次のとおりです。

- com.bea.wlpi.server.plugin.PluginAction インタフェースを実装し、プラグイン アクション実行情報を定義します。
- Studio 内の各種ダイアログ ボックスに表示されるアクション ツリーにリストされるアクションおよびアクション カテゴリをカスタマイズします。

プラグイン アクションのための実行情報の定義

プラグイン アクションのための実行情報を定義するには、次の表で説明するように、com.bea.wlpi.server.plugin.PluginAction インタフェースとそのメソッドを実装する必要があります。

表 4-12 PluginAction インタフェース メソッド

メソッド	説明
`public int execute(com.bea.wlpi.server.plugin.ActionContext actionContext, com.bea.wlpi.server.common.ExecutionContext execContext) throws com.bea.wlpi.common.WorkflowException`	プラグイン アクションとそのビジネス ロジックを実行する。 メソッド パラメータの定義は次のとおり。 ■ *actionContext* ―アクション コンテキストを示す com.bea.wlpi.server.plugin.ActionContext オブジェクト。アクション コンテキストは、サブアクションの実行やサブワークフローのインスタンス化のようなアクション レベルのサービスを提供する。 ■ *execContext* ―実行コンテキストを示す com.bea.wlpi.server.common.ExecutionContext オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。 アクションと実行コンテキストの詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照。 このメソッドは、処理を続行するかどうかを指定するリターン コードを示す次のいずれかの com.bea.wlpi.server.common.ExecutionContext 整数値を返す。 ■ EXIT_CONTINUE ―エラー ハンドラを終了し、後続の操作の処理を許可する。 ■ EXIT_RETRY ―エラー ハンドラを終了し、失敗した操作の再試行を要求する。 ■ EXIT_ROLLBACK ―エラー ハンドラを終了し、ユーザ トランザクションのロールバックを要求する。 ■ STOP ―プラグインの実行を停止する。

表 4-12 PluginAction インタフェース メソッド （続き）

メソッド	説明
`public void fixup(com.bea.wlpi.evaluator.ExpressionParser parser) throws com.bea.wlpi.common.WorkflowException`	プラグイン アクションが必要な式をコンパイルできるようにする。 Plug-in Manager は、テンプレート定義を解析して、結果をメモリに格納した後、ワークフローを開始する前に、このメソッドを呼び出す。 メソッド パラメータの定義は次のとおり。 *parser* ―式パーサを示す `com.bea.wlpi.evaluator.ExpressionParser` オブジェクト。

表 4-12 PluginAction インタフェース メソッド （続き）

メソッド	説明
`public void response(com.bea.wlpi.server.plugin.ActionContext actionContext, com.bea.wlpi.server.common.ExecutionContext execContext, java.lang.Object data) throws com.bea.wlpi.common.WorkflowException`	このアクションに送られる非同期応答を処理する。 通常、このメソッドは、前の `execute()` メソッド（この表の前段で説明）に対する呼び出しでアクションにより生成された外部要求への応答として返される。 プラグイン アクションは、このメソッドを使用して、サブアクションの非同期実行などを開始できる。Plug-in Manager は、`com.bea.wlpi.server.worklist.Worklist.response()` メソッドまたは他のオーバーロード メソッドに対する呼び出しを受け取ったときに、このメソッドを呼び出す。 メソッド パラメータの定義は次のとおり。 ■ *`actionContext`* ―アクション コンテキストを示す `com.bea.wlpi.server.plugin.ActionContext` オブジェクト。アクション コンテキストは、サブアクションの実行やサブワークフローのインスタンス化のようなアクション レベルのサービスを提供する。 ■ *`execContext`* ―実行コンテキストを示す `com.bea.wlpi.server.common.ExecutionContext` オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。 ■ *`data`* ― 必要な情報を抽出するためにプラグインにより既知のタイプにキャストされるデータ オブジェクトを示す `java.lang.Object` オブジェクト。 アクションと実行コンテキストの詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照。

表 4-12 PluginAction インタフェース メソッド （続き）

メソッド	説明
`public void startedWorkflowDone(com.bea.wlpi.server.plugin.ActionContext` *`actionContext`*`, com.bea.wlpi.server.common.ExecutionContext` *`execContext`*`, com.bea.wlpi.common.VariableInfo[]` *`output`*`) throws com.bea.wlpi.common.WorkflowException`	以前に開始されていたサブワークフローが完了したことをプラグイン アクションに通知する。 メソッド パラメータの定義は次のとおり。 ■ *`actionContext`* ―アクション コンテキストを示す `com.bea.wlpi.server.plugin.ActionContext` オブジェクト。アクション コンテキストは、サブアクションの実行やサブワークフローのインスタンス化のようなアクション レベルのサービスを提供する。 ■ *`execContext`* ―実行コンテキストを示す `com.bea.wlpi.common.common.ExecutionContext` オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。 ■ *`output`* ―呼び出されたワークフローの出力変数を示す `com.bea.wlpi.common.VariableInfo` 配列。 アクションと実行コンテキストの詳細については、4-99 ページの 「プラグイン実行時コンテキストの使い方」 を参照。

プラグイン サンプルから抜粋した次のコード リストは、アクションのための実行時コンポーネント クラスを定義する方法を示しています。この抜粋は、*`SAMPLES_HOME`*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `CheckInventoryAction.java` ファイルから取り出したものです。重要なコード行は、**太字**で示します。

注意： プラグイン アクション実行時コンポーネント クラスの定義方法を示すその他の例については、*`SAMPLES_HOME`*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `SendConfirmationAction.java` ファイルを参照してください。

コード リスト 4-17 アクションのための実行時コンポーネント クラスの定義

```
package com.bea.wlpi.tour.po.plugin;

import java.io.IOException;
import com.bea.wlpi.server.plugin.PluginAction;
import com.bea.wlpi.common.WorkflowException;
import com.bea.wlpi.common.plugin.PluginException;
import com.bea.wlpi.common.Messages;
import com.bea.wlpi.common.VariableInfo;
import com.bea.wlpi.evaluator.Expression;
import com.bea.wlpi.evaluator.EvaluatorException;
import com.bea.wlpi.server.common.ExecutionContext;
import com.bea.wlpi.evaluator.ExpressionParser;
import com.bea.wlpi.server.plugin.ActionContext;
import org.xml.sax.*;

public class CheckInventoryAction extends CheckInventoryActionObject
        implements PluginAction {
    private Expression inputValueExpression;
    static int[] quantities = {
        250, 120, 5, 75, 0, 300, 550, 25, 16, 630, 3
    };

    public CheckInventoryAction() {
    }

    public void fixup(ExpressionParser parser) {

        System.out.println("SamplePlugin: CheckInventoryAction.fixup called");

        try {
            inputValueExpression =
                inputVariableName != null
                ? new Expression("$" + inputVariableName, parser) : null;
        } catch (EvaluatorException ee) {
            System.out.println("EvaluationException ocurred in
CheckInventoryAction");
        }
    }

    public int execute(ActionContext actionContext, ExecutionContext context)
            throws WorkflowException {

        System.out.println("SamplePlugin: CheckInventoryAction.execute called");

        Object valueObject = inputValueExpression != null
```

```
                            ? inputValueExpression.evaluate(context) : null;

        if (valueObject == null)
            throw new PluginException("Sample Plugin", "itemNo is null");

        if (!(valueObject instanceof Long))
            throw new PluginException("Sample Plugin", "itemNo not an integer");

        int itemNo = ((Long)valueObject).intValue();
        int quantity = quantities[itemNo % quantities.length] + itemNo;

        System.out.println("CheckInventoryAction: Output = " + quantity);
        context.setVariableValue(outputVariableName, new Long(quantity));

        return ExecutionContext.CONTINUE;
    }

    public void response(ActionContext actionContext, ExecutionContext
execContext, Object data)
            throws WorkflowException {
    }

    public void startedWorkflowDone(ActionContext actionContext,
                                    ExecutionContext context,
                                    VariableInfo[] output) {
    }
```

関連する次のコード例も参照してください。

- *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの CheckInventoryActionObject.java では、XML フォーマットでプラグイン データを読み込む方法を説明しています。
- 4-23 ページの「PluginActionData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。このクラスは、CheckInventoryActionObject クラスを拡張します。
- 4-43 ページの「PluginActionPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-74 ページの「アクション ツリーのカスタマイズ」では、Studio 内の各種ダイアログ ボックスに表示されるアクション ツリーにリストされるアク

ションおよびアクション カテゴリをカスタマイズする方法を説明しています。

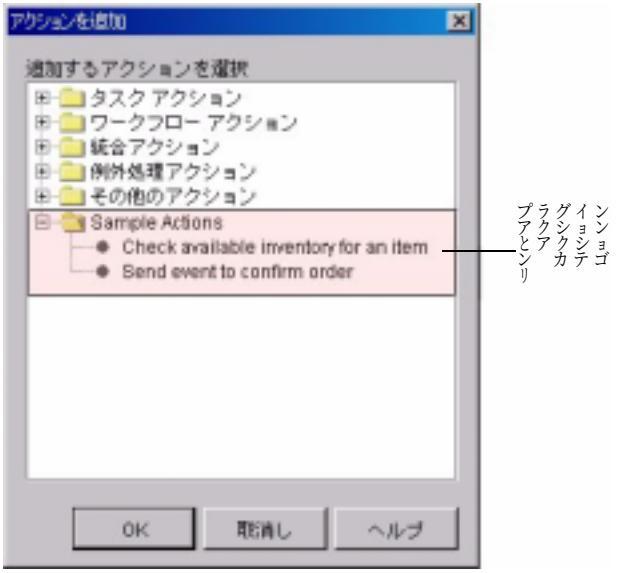
プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

アクション ツリーをカスタマイズする

プラグイン アクションをサポートするには、Studio 内の各種ダイアログ ボックスに表示されるアクション ツリーにリストされるアクションおよびアクション カテゴリをカスタマイズする必要があります。

たとえば、次の図は、アクション ツリーがカスタマイズされたバージョンの [アクションを追加] ダイアログ ボックスです。

図 4-8 カスタマイズされたアクション ツリー



前の図に示したように、BPM アクション ツリーに、新しいアクション カテゴリである Sample Actions が追加されています。これは次のプラグイン アクションを提供します。

- Checks available inventory for an item アクション

- Sends Confirm Order Event アクション

アクション ツリーをカスタマイズする手順は、次のとおりです。

1. com.bea.wlpi.common.plugin.CategoryInfo オブジェクトを定義します。このオブジェクトが、カスタムのプラグイン アクションとアクション カテゴリを定義します。

   CategoryInfo オブジェクトの作成方法については、2-7 ページの「プラグイン値オブジェクトの定義」を参照してください。

2. com.bea.wlpi.server.plugin.Plugin リモート インタフェースの getPluginCapabilitiesInfo() メソッドを実装して、com.bea.wlpi.common.plugin.PluginCapabilitiesInfo オブジェクトを定義します。

   手順 1 で定義した CategoryInfo オブジェクトを使用して、カスタム アクション ツリー特性を定義します。

Plug-in Manager は、getPluginCapabilitiesInfo() メソッドを呼び出すときに、既存のアクション カテゴリ ツリーを com.bea.wlpi.common.plugin.CategoryInfo オブジェクトとして渡し、プラグインがツリーを調べて、カスタムのアクションやアクション カテゴリを追加する位置を決定できるようにする必要があります。Plug-in Manager は、この有効なツリー構造を取得すると、取得したツリー構造を既存のツリー構造と結合し、新しいカテゴリのそれぞれに新しい systemID を割り当てます。

Plug-in Manager は、getPluginCapabilitiesInfo() メソッドを複数回呼び出すことができますが、毎回新たに初期化されたアクション ツリーを返す必要があります。既存の CategoryInfo オブジェクトを再利用した場合、Plug-in Manager は、setSystemID() メソッドの 2 回目の呼び出し時に IllegalStateException を生成します。

既存のアクション カテゴリのどのレベルでも新しいアクションまたはサブカテゴリを追加できますが、既存のカテゴリからアクションやサブアクションを削除することはできません。

プラグイン サンプルから抜粋した次のコード リストは、次のメソッドを定義する方法を示しています。

- getCategoryInfo() メソッド ―このメソッドは、カスタムのプラグイン アクションとカテゴリを定義する CategoryInfo オブジェクトを定義します。

- getPluginCapabilitiesInfo() メソッド ―このメソッドにより、アクション ツリーをカスタマイズできます。

この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SamplePluginBean.java ファイルから取り出したものです。この例は、Check Inventory アクションと Send Confirmation アクションの 2 つを定義します。重要なコード行は、**太字**で示します。

コード リスト 4-18 アクション ツリーのカスタマイズ

```
private CategoryInfo[] getCategoryInfo(SampleBundle bundle) {

    ActionInfo checkInventoryAction =
            new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 1,
                    bundle.getString("checkInventoryName"),
                    bundle.getString("checkInventoryDesc"), ICON_BYTE_ARRAY,
                    ActionCategoryInfo.ID_NEW,
                    ActionInfo.ACTION_STATE_ALL,
                    SamplePluginConstants.CHECKINV_CLASSES);

    ActionInfo sendConfirmAction =
            new ActionInfo(SamplePluginConstants.PLUGIN_NAME, 2,
                    bundle.getString("sendConfirmName"),
                    bundle.getString("sendConfirmDesc"), ICON_BYTE_ARRAY,
                    ActionCategoryInfo.ID_NEW,
                    ActionInfo.ACTION_STATE_ALL,
                    SamplePluginConstants.SENDCONF_CLASSES);

    ActionCategoryInfo[] actions =
            new ActionCategoryInfo[]{ checkInventoryAction, sendConfirmAction};

    CategoryInfo[] catInfo =
          new CategoryInfo[]{ new CategoryInfo(SamplePluginConstants.PLUGIN_NAME,
                    0, bundle.getString("catName"),
                    bundle.getString("catDesc"),
                    ActionCategoryInfo.ID_NEW,
                    actions)};
    return catInfo;
}

public PluginCapabilitiesInfo getPluginCapabilitiesInfo(Locale lc,
        CategoryInfo[] info) {

    PluginInfo pi;
```

```
FieldInfo orderFieldInfo;
FieldInfo confirmFieldInfo;
FieldInfo[] fieldInfo;
FunctionInfo fi;
FunctionInfo[] functionInfo;
StartInfo si;
StartInfo[] startInfo;
EventInfo ei;
EventInfo[] eventInfo;
SampleBundle bundle = new SampleBundle(lc);

log("getPluginCapabilities called");

pi = createPluginInfo(lc);

orderFieldInfo =
        new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 3,
                bundle.getString("orderFieldName"),
                bundle.getString("orderFieldDesc"),
                SamplePluginConstants.ORDER_FIELD_CLASSES, false);

confirmFieldInfo =
        new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 4,
                bundle.getString("confirmFieldName"),
                bundle.getString("confirmFieldDesc"),
                SamplePluginConstants.CONFIRM_FIELD_CLASSES, false);

fieldInfo = new FieldInfo[]{ orderFieldInfo, confirmFieldInfo};

ei = new EventInfo(SamplePluginConstants.PLUGIN_NAME, 6,
                   bundle.getString("confirmOrderName"),
                   bundle.getString("confirmOrderDesc"), ICON_BYTE_ARRAY,
                   SamplePluginConstants.EVENT_CLASSES,
                   confirmFieldInfo);

eventInfo = new EventInfo[]{ ei};

fi = new FunctionInfo(SamplePluginConstants.PLUGIN_NAME, 7,
        bundle.getString("calcTotalName"),
        bundle.getString("calcTotalDesc"),
        bundle.getString("calcTotalHint"),
        SamplePluginConstants.FUNCTION_CLASSES, 3, 3);

functionInfo = new FunctionInfo[]{ fi};

si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
        bundle.getString("startOrderName"),
        bundle.getString("startOrderDesc"), ICON_BYTE_ARRAY,
        SamplePluginConstants.START_CLASSES, orderFieldInfo);
```

```
    startInfo = new StartInfo[]{ si};

    PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
            getCategoryInfo(bundle), eventInfo, fieldInfo, functionInfo,
startInfo,
            null, null, null, null, null);

    return pci;
}
```

関連する次のコード例も参照してください。

- *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの CheckInventoryActionObject.java では、XML フォーマットでプラグイン データを読み込む方法を説明しています。
- 4-23 ページの「PluginActionData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。このクラスは、CheckInventoryActionObject クラスを拡張します。
- 4-43 ページの「PluginActionPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-70 ページの「アクションのための実行時コンポーネント クラスの定義」では、プラグイン実行情報を定義する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

完了ノードのための実行時コンポーネント クラスの定義

完了ノードのための実行時コンポーネント クラスを定義するには、com.bea.wlpi.server.plugin.PluginDone インタフェースを実装します。

注意： PluginDone インタフェースは、com.bea.wlpi.server.plugin.PluginTemplateNode を拡張します。PluginTemplateNode インタフェースとそのメソッドの詳細については、4-97 ページの「PluginTemplateNode インタフェース」を参照してください。

PluginDone インタフェースにより追加されるメソッドはありません。

次のコード リストで、完了ノードのための実行時コンポーネント クラスを定義する方法を示します。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 4-19 完了ノードのための実行時コンポーネント クラスの定義

```
package com.bea.wlpi.test.plugin;

import com.bea.wlpi.common.Messages;
import com.bea.wlpi.common.WorkflowException;
import com.bea.wlpi.evaluator.ExpressionParser;
import com.bea.wlpi.server.common.ExecutionContext;
import com.bea.wlpi.server.plugin.PluginDone;
import java.io.IOException;
import java.util.Map;
import org.xml.sax.*;

public class DoneNode extends DoneObject implements PluginDone {
    public DoneNode() {
    }

    public int activate(ExecutionContext context)
        throws WorkflowException {

        System.out.println("TestPlugin: DoneNode activated");

        // プラグイン インスタンス データを初期化する
        Map instanceData =
(Map)context.getPluginInstanceData(TestPluginConstants.PLUGIN_NAME);
        if (instanceData != null) {
            Object started =
instanceData.get(TestPluginConstants.INST_DATA_STARTED);
            System.out.println("instance data = " + started);
        }

        int stopMode;
```

```
        if (yesOrNo.equals(TestPluginConstants.DONE_YES)) {
            System.out.println("TestPlugin: DoneNode = YES");
            stopMode = ExecutionContext.CONTINUE;
        } else {
            System.out.println("TestPlugin: DoneNode = NO");
            stopMode = ExecutionContext.STOP;
        }

        return stopMode;
    }

    public void fixup(ExpressionParser parser) {
    }
```

関連する次のコード例も参照してください。

- 4-5 ページの「完了ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明しています。
- 4-14 ページの「完了ノードのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。このクラスは、`PluginObject` クラスを拡張します。
- 4-35 ページの「完了ノードのための PluginPanel クラスの定義」では、プラグイン GUI コンポーネントを定義する方法を説明しています。

イベント ノードのための実行時コンポーネント クラスの定義

イベント ノードのための実行時コンポーネント クラスを定義するには、`com.bea.wlpi.server.plugin.PluginEvent` インタフェースを実装します。

次の表で、実行時コンポーネント クラスの一部として実装の必要な `PluginEvent` インタフェース メソッドについて説明します。

表 4-13 PluginEvent インタフェース メソッド

メソッド	説明
`public int activate(com.bea.wlpi.server.common.EventContext *eventContext*, com.bea.wlpi.server.common.ExecutionContext *execContext*) throws com.bea.wlpi.common.WorkflowException`	イベントをアクティブ化する。 Event Processor は、マッチング イベントが前のノードからの着信遷移によりアクティブ化されるときに、このメソッドを呼び出す。 次に、プラグインは、イベント ウォッチを記録して、Event Processor が着信イベントをこの特定のノードとワークフロー インスタンスに対してマッチングできるようにする。プラグインは、`com.bea.wlpi.server.plugin.EventContext` の `activateEvent()` メソッドを呼び出すことにより、デフォルトのイベント ウォッチ登録、送信されるメッセージの処理、イベント マッチング機能を使用できる。プラグインでは、必要な情報を記録すること、および定義された基準に基づいて実行時マッチングを実行するイベント ハンドラを提供する必要がある。プラグイン イベントの処理方法の詳細については、6-1 ページの「プラグイン イベントの処理」を参照。 **注意：** プラグインが BPM JMS イベント リスナに依存していない場合、イベント ハンドラを用意する必要はありません。 メソッド パラメータの定義は次のとおり。 ■ *`eventContext`* — イベント コンテキストを示す `com.bea.wlpi.server.plugin.EventContext` オブジェクト。イベント コンテキストは、イベント ウォッチ登録のような実行時イベント関係のサービスへのアクセスを提供する。 ■ *`execContext`* — 実行コンテキストを示す `com.bea.wlpi.server.common.ExecutionContext` オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。 イベントと実行コンテキストの詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照。 このメソッドは、イベントがアクティブ化された後に処理を続行するかどうかを指定するリターン コードを示す次のいずれかの `com.bea.wlpi.server.common.ExecutionContext` 整数値を返す。 ■ `CONTINUE` —処理を続行する。 ■ `STOP` —処理を停止する。

表 4-13 PluginEvent インタフェース メソッド （続き）

メソッド	説明
`public void fixup(com.bea.wlpi.evaluator.ExpressionParser parser) throws com.bea.wlpi.common.WorkflowException`	プラグイン ノードが必要な式をコンパイルできるようにする。 Plug-in Manager は、テンプレート定義を解析して、結果をメモリに格納した後、ワークフローを開始する前に、このメソッドを呼び出す。 メソッド パラメータの定義は次のとおり。 *parser* ―式パーサを示す `com.bea.wlpi.evaluator.ExpressionParser` オブジェクト。

表 4-13 PluginEvent インタフェース メソッド （続き）

メソッド	説明
`public int trigger(com.bea.wlpi.server.common.EventContext` *`eventContext`*`, com.bea.wlpi.server.common.ExecutionContext` *`execContext`*`) throws com.bea.wlpi.common.WorkflowException`	イベントをトリガする。 Event Processor は、このイベント ノードの基準とマッチする着信イベントを検出したときに、このメソッドを呼び出す。デフォルトのイベント ウォッチ登録とマッチング サービスを使用するプラグインは、`com.bea.wlpi.server.plugin.EventContext` の `removeEventWatch()` メソッドを呼び出して、イベントを非リスン状態にする必要がある。デフォルトのイベント ウォッチ登録とマッチング サービスを使用しないプラグインは、前の `activate()` の呼び出し時にこのノードに対してプラグイン提供のイベント ウォッチ レコードが確立されている場合、それらをすべて無効にする必要がある。 プラグインは、BPM JMS イベント リスナを介してイベントが到着する場合、イベント ハンドラを提供する必要がある。そうでない場合は、独自のイベント リスナ サービスを実装する必要がある。イベント ハンドラの定義方法の詳細については、6-1 ページの「プラグイン イベントの処理」を参照。 メソッド パラメータの定義は次のとおり。 ■ *`eventContext`* ―イベント コンテキストを示す `com.bea.wlpi.server.plugin.EventContext` オブジェクト。イベント コンテキストは、イベント ウォッチ登録のような実行時イベント関係のサービスへのアクセスを提供する。 ■ *`execContext`* ―実行コンテキストを示す `com.bea.wlpi.server.common.ExecutionContext` オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。 イベントと実行コンテキストの詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照。 このメソッドは、イベントがトリガされた後に処理を続行するかどうかを指定するリターン コードを示す次のいずれかの `com.bea.wlpi.server.common.ExecutionContext` 整数値を返す。 ■ `CONTINUE` ―処理を続行する。 ■ `STOP` ―処理を停止する。

プラグイン サンプルから抜粋した次のコード リストは、イベント ノードのための実行時コンポーネント クラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの StartNode.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-20 イベント ノードのための実行時コンポーネント クラスの定義

```
package com.bea.wlpi.tour.po.plugin;

import java.io.IOException;
import com.bea.wlpi.server.plugin.PluginEvent;
import com.bea.wlpi.common.WorkflowException;
import com.bea.wlpi.common.Messages;
import com.bea.wlpi.evaluator.Expression;
import com.bea.wlpi.evaluator.EvaluatorException;
import com.bea.wlpi.server.common.ExecutionContext;
import com.bea.wlpi.server.plugin.EventContext;
import com.bea.wlpi.server.workflow.Workflow;
import com.bea.wlpi.server.workflow.Variable;
import com.bea.wlpi.evaluator.ExpressionParser;
import com.bea.wlpi.server.workflow.TemplateNode;
import org.xml.sax.*;

public class EventNode extends EventObject implements PluginEvent {

    public EventNode() {
    }

    public int activate(EventContext eventContext, ExecutionContext execContext)
            throws WorkflowException {

        System.out.println("SamplePlugin: EventNode activated");
        eventContext.activateEvent(execContext,
                                   SamplePluginConstants.CONTENTTYPE,
                                   eventDesc, null, null);

        return ExecutionContext.CONTINUE;
    }

    public int trigger(EventContext context, ExecutionContext execContext)
            throws WorkflowException {

        System.out.println("SamplePlugin: EventNode triggered");
```

```
        context.removeEventWatch(execContext);

        return ExecutionContext.CONTINUE;
    }

    public void fixup(ExpressionParser parser) {
    }
}
```

関連する次のコード例も参照してください。

- *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの EventObject.java では、プラグイン データを読み込むために PluginObject インタフェースを実装する方法を説明しています。
- 4-16 ページの「イベント ノードのための PluginData インタフェースの実装」では、プラグイン データを読み込み、保存する方法を説明しています。このクラスは、EventObject クラスを拡張します。
- 4-55 ページの「イベント ノードのための PluginTriggerPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

関数のための実行時コンポーネント クラスの定義

関数のための実行時コンポーネント クラスを定義するには、com.bea.wlpi.common.plugin.PluginFunction インタフェースを実装します。次の表で、実装の必要な PluginFunction インタフェース メソッドについて説明します。

表 4-14 PluginFunction インタフェース メソッド

メソッド	説明
`public java.lang.Object evaluate(com.bea.wlpi.evaluator.EvaluationContext context, java.lang.Object[] args ) throws com.bea.wlpi.common.plugin.PluginException`	関数を評価する。 式エバリュエータは、この関数呼び出しを評価する必要がある場合に、このメソッドを呼び出す。プラグイン関数は、コンテキスト パラメータにより提供されるコンテキスト情報から戻り値を計算できる。このパラメータは、イベント データとワークフロー インスタンス状態へのアクセスを提供する（該当する場合）。 メソッド パラメータの定義は次のとおり。 ■ *context* ―評価コンテキストを示す `com.bea.wlpi.evaluator.EvaluationContext` オブジェクト。 ■ *args* ―パラメータ値を示す `java.lang.Object` 値の配列。値は、ソース式で関数引数を表す式の評価を通じて、あらかじめ計算される。 このメソッドは、関数評価結果を示す `java.lang.String` オブジェクトを返す。

プラグイン サンプルから抜粋した次のコード リストは、関数のための実行時コンポーネント クラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `CalculateTotalPriceFunction.java` ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-21 関数のための実行時コンポーネント クラスの定義

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.Messages;
import com.bea.wlpi.common.plugin.PluginFunction;
import com.bea.wlpi.common.plugin.PluginException;
import com.bea.wlpi.evaluator.*;
import com.bea.wlpi.tour.po.BadStateException;
import java.lang.NumberFormatException;

/**
 * このサンプル関数は、受注合計額を計算する。
 * 金額は、品目 ID、数量、および出荷先を使用して計算される。
 * 出荷先は、売上税率を調べるために使用される。
 */
public class CalculateTotalPriceFunction implements PluginFunction {
    static StateTax[] stateTax = {
        new StateTax("AB", 0.07), new StateTax("AK", 0.06),
        new StateTax("AL", 0.06), new StateTax("AR", 0.03),
        new StateTax("AZ", 0.05), new StateTax("BC", 0.05),
        new StateTax("CA", 0.04), new StateTax("CO", 0.08),
        new StateTax("CT", 0.03), new StateTax("DC", 0.05),
        new StateTax("DE", 0.05), new StateTax("FL", 0.00),
        new StateTax("GA", 0.06), new StateTax("HI", 0.07),
        new StateTax("IA", 0.07), new StateTax("ID", 0.08),
        new StateTax("IL", 0.06), new StateTax("IN", 0.03),
        new StateTax("KS", 0.05), new StateTax("KY", 0.07),
        new StateTax("LA", 0.06), new StateTax("MA", 0.05),
        new StateTax("MB", 0.05), new StateTax("MD", 0.04),
        new StateTax("ME", 0.04), new StateTax("MI", 0.03),
        new StateTax("MN", 0.05), new StateTax("MO", 0.06),
        new StateTax("MS", 0.07), new StateTax("MT", 0.07),
        new StateTax("NB", 0.08), new StateTax("NC", 0.07),
        new StateTax("ND", 0.08), new StateTax("NE", 0.03),
        new StateTax("NF", 0.06), new StateTax("NH", 0.09),
        new StateTax("NJ", 0.03), new StateTax("NM", 0.06),
        new StateTax("NV", 0.03), new StateTax("NY", 0.06),
        new StateTax("NS", 0.08), new StateTax("NT", 0.07),
        new StateTax("OH", 0.07), new StateTax("OK", 0.02),
        new StateTax("ON", 0.08), new StateTax("OR", 0.08),
        new StateTax("PA", 0.07), new StateTax("PE", 0.07),
        new StateTax("PQ", 0.05), new StateTax("RI", 0.05),
        new StateTax("SC", 0.05), new StateTax("SD", 0.04),
        new StateTax("SK", 0.04), new StateTax("TN", 0.06),
        new StateTax("TX", 0.06), new StateTax("UT", 0.07),
        new StateTax("VA", 0.07), new StateTax("VT", 0.08),
```

```
        new StateTax("WA", 0.07), new StateTax("WI", 0.07),
        new StateTax("WV", 0.08), new StateTax("WY", 0.05),
        new StateTax("YT", 0.07)
    };
    static double[] prices = {
        29.95, 524.79, 33.21, 9.99, 12.28, 152.50, 43.55, 32.90, 328.55, 72.50,
        87.50
    };

    public CalculateTotalPriceFunction() throws EvaluatorException {
        System.out.println("CalculateTotalPriceFunction: constructor called");
    }

    public Object evaluate(EvaluationContext context, Object[] args)
            throws PluginException {

        int itemNo;
        int quantity;
        String state;

        System.out.println("CalculateTotalPriceFunction: evaluate called");

        try {
            itemNo = ((Long)args[0]).intValue();
        } catch (Exception e) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Invalid ItemID argument");
        }

        try {
            quantity = ((Long)args[1]).intValue();
        } catch (Exception e2) {
            e2.printStackTrace();

            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Invalid Quantity argument");
        }

        if (!(args[2] instanceof String)) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "Invalid State argument");
        }

        state = (String)args[2];

        int i;

        // stateTax 配列で出荷先を検索する
        for (i = 0; i < stateTax.length; ++i) {
```

```
            if (stateTax[i].equals(state))


                break;
        }

        if (i == stateTax.length)
            throw new PluginException(new BadStateException("Invalid state
abbreviation: "
                    + state));

        double total = (prices[itemNo % prices.length] + itemNo / prices.length)
                    * quantity * (1 + stateTax[i].getTax());

        return new Double(total);
    }
}

class StateTax {
    String abbrev;
    double tax;

    public boolean equals(String abbrev) {
        return this.abbrev.equalsIgnoreCase(abbrev);
    }

    public double getTax() {
        return tax;
    }

    public StateTax(String abbrev, double tax) {
        this.abbrev = abbrev;
        this.tax = tax;
    }
}
```

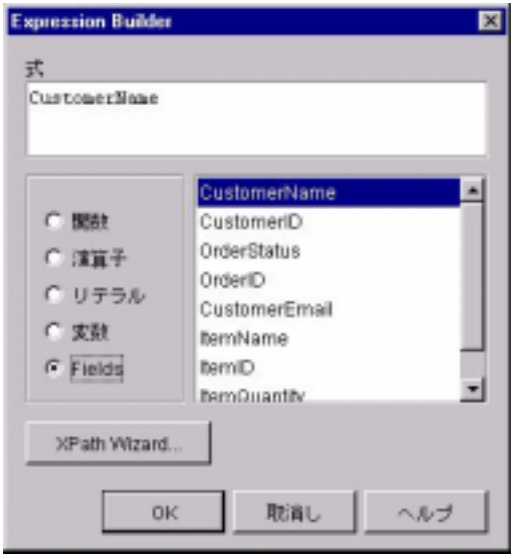
プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

メッセージ タイプのための実行時コンポーネント クラスの定義

メッセージ タイプのための実行時コンポーネントを定義するには、`com.bea.wlpi.common.plugin.PluginField` インタフェースを実装して、プラグイン フィールドを定義する必要があります。プラグイン フィールドにより、開始ノードまたはイベント ノードで受信した外部イベントと関連付けられているカスタム プラグイン データを解析できます。その後、エバリュエータ式からこのデータを参照することができます。プラグインは、関連付けられたイベント記述子にアクセスすることにより、外部イベントのコンテンツ タイプを判別します。

プラグイン フレームワークは、プラグイン フィールド データを使用して、[Expression Builder] ダイアログ ボックスを表示します。たとえば、次の図に示す [Expression Builder] ダイアログ ボックスでは、プラグインの [Fields] カテゴリが選択され、その結果である有効なフィールド要素のリストが表示されています。

図 4-9 [Expression Builder] ダイアログ ボックスに表示されたプラグイン フィールド



プラグイン フィールドを定義するには、com.bea.wlpi.common.plugin.PluginField インタフェースを実装します。次の表で、実装の必要な PluginField インタフェース メソッドについて説明します。

表 4-15 PluginField インタフェース メソッド

メソッド	説明
public java.lang.Object evaluate(com.bea.wlpi.evaluator.EvaluationContext *context*) throws com.bea.wlpi.common.plugin.PluginException	フィールドを評価する。 式エバリュエータは、このオブジェクトにより参照される値を、評価コンテキスト パラメータに含まれるイベントから取得する必要がある場合に、このメソッドを呼び出す。 その値を計算する際、プラグイン フィールドはフィールド修飾子を考慮する必要がある。 メソッド パラメータの定義は次のとおり。 *context* ―評価コンテキストを示す com.bea.wlpi.evaluator.EvaluationContext オブジェクト。 このメソッドは、フィールド評価結果を示す java.lang.Object オブジェクトを返す。

表 4-15 PluginField インタフェース メソッド （続き）

メソッド	説明
`public void init(java.lang.String name, java.lang.String eventDescriptor) throws com.bea.wlpi.common.plugin.PluginException`	フィールド名とイベント記述子の値を初期化する。 Plug-in Manager は、イベント記述子のコンテキストで値が要求されているフィールドの名前を使用する。 1 つのフィールド タイプは、1 つまたは複数のメッセージ タイプをサポートできる。複数のメッセージ タイプがサポートされている場合、フィールド タイプは、イベント記述子を使用してタイプを区別する。 メソッド パラメータの定義は次のとおり。 ■ *name* ―フィールド名を示す `java.lang.String` オブジェクト。 ■ *eventDescriptor* ―フィールド イベント記述子を示す `java.lang.String` オブジェクト。

表 4-15 PluginField インタフェース メソッド （続き）

メソッド	説明
`public void setQualifier(com.bea.wlpi.server.plugin.PluginField *qualifier*) throws com.bea.wlpi.common.plugin.PluginException`	フィールド修飾子を設定する。 Plug-in Manager は、修飾されたフィールド参照を式パーサが検出したときに、このメソッドを呼び出す。 このメソッドは、プラグイン フィールドが、外部データ ディクショナリ情報にアクセスし、コンパイルされた式の一部として保存されている列詳細を取得できるようにする。 メソッド パラメータの定義は次のとおり。 *qualifier* ―このオブジェクトと同じクラスのフィールド修飾子を示す `com.bea.wlpi.common.plugin.PluginField` オブジェクト。

プラグイン サンプルから抜粋した次のコード リストは、メッセージ タイプのための実行時コンポーネント クラスを定義する方法を示しています。このプラグイン フィールドは、顧客の注文と関連付けられたデータの入っている 1 つのメッセージ タイプを処理します。データは、`String` フォーマットであり、各データ要素はセミコロンで区切られています。プラグイン フィールドには、セミコロンで区切られた値の入っている文字列バッファとして外部イベント データを提供することになっています。`execute()` メソッドでは、このクラスはイベント データについて単純な検証をいくつか実行し、要求されたフィールド名として `String` オブジェクトを返します。この抜粋は、*SAMPLES_HOME*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `OrderField.java` ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-22 メッセージ タイプのための実行時コンポーネント クラスの定義

```
package com.bea.wlpi.tour.po.plugin;

import com.bea.wlpi.common.plugin.PluginException;
```

```
import com.bea.wlpi.common.plugin.PluginField;
import com.bea.wlpi.evaluator.EvaluationContext;
import com.bea.wlpi.server.eventprocessor.EventData;
import java.util.StringTokenizer;

/*
 * このサンプル フィールド タイプは、セミコロンで区切られたフィールド値の入った
 * 文字列バッファを予想している。フィールドの順序は固定である。.
 */

public final class OrderField implements PluginField {
    private String docType;
    private String name;

    public void init(String name, String eventDescriptor)
            throws PluginException {
        this.name = name;
        docType = eventDescriptor;
    }

    public void setQualifier(PluginField qualifier) throws PluginException {

        System.out.println("OrderField.setQualifier(" + qualifier + ')');

        throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                  "Qualifiers are not supported");
    }

    public Object evaluate(EvaluationContext context) throws PluginException {

        // イベント データを取得し、それが String オブジェクトであることを確認する
        EventData eventData = context.getEventData();

        if (eventData == null)
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event data is null.");

        docType = eventData.getEventDescriptor();

        if (!docType.equals(SamplePluginConstants.START_ORDER_EVENT))
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event descriptor is invalid.");

        Object object = eventData.getContent();

        if (!(object instanceof String))
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event data is invalid.");
```

```
        // フィールド名が有効であるかチェックする
        int i;

        for (i = 0; i < SamplePluginConstants.ORDER_FIELDS.length; i++) {
            if (SamplePluginConstants.ORDER_FIELDS[i].equals(name))


                break;
        }

        // 有効なフィールド名のリストにフィールド名があるか？
        if (i == SamplePluginConstants.ORDER_FIELDS.length)
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The field name " + name
                                      + " is invalid.");

        String data = (String)object;
        StringTokenizer st = new StringTokenizer(data, ";");
        String token = null;

        while (st.hasMoreTokens() && i >= 0) {
            token = st.nextToken();

            i--;
        }

        // 必要なフィールドの検索が終了する前に、その中のデータがなくなったか？
        if (i >= 0) {
            throw new PluginException(SamplePluginConstants.PLUGIN_NAME,
                                      "The event data is invalid.");
        }

        String value = token;

        System.out.println("OrderField: name = " + name + ", value = " + value);

        // テキスト値を返す
        return value;
    }
}
```

The `ORDER_FIELDS` value is defined within the `SamplePluginConstants.java` class file as follows:

```
final static String[] ORDER_FIELDS = {
     "CustomerName", "CustomerID", "OrderStatus", "OrderID",
     "CustomerEmail", "ItemName", "ItemID", "ItemQuantity",
     "CustomerState"
};
```

4-88 ページの「[Expression Builder] ダイアログ ボックスに表示されたプラグイン フィールド」の図に、[Expression Builder] ダイアログ ボックス内に表示されるフィールド要素を示してあります。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

開始ノードのための実行時コンポーネント クラスの定義

開始ノードのための実行時コンポーネント クラスを定義するには、`com.bea.wlpi.server.plugin.PluginStart2` インタフェースを実装します次の表で、実装の必要な `PluginStart2` インタフェース メソッドについて説明します。

注意： `PluginStart2` インタフェースは、`com.bea.wlpi.server.plugin.PluginTemplateNode` インタフェースからのメソッドを継承します。詳細については、4-97 ページの「PluginTemplateNode インタフェース」を参照してください。

表 4-16 PluginStart2 インタフェース メソッド

メソッド	説明
`public void setTrigger(com.bea.wlpi.server.plugin.EventContext *context*, java.lang.String *orgExpr*, boolean *orgIsExpression*) throws com.bea.wlpi.common.WorkflowException`	この開始ノードのためのイベント ウォッチを設定する。 テンプレート定義は、テンプレート定義がアクティブ化され、保存されるときに、このメソッドを呼び出す。 プラグインは、このメソッドを使用してイベント ウォッチを記録する。Event Processor は、記録されたイベント ウォッチを使用して、着信イベントをこの特定のノードとテンプレート定義に対してマッチングできる。プラグインは、`com.bea.wlpi.server.plugin.EventContext.postStartWatch()` メソッドを呼び出すことにより、デフォルトのイベント ウォッチ登録、送信されるメッセージの処理、イベント マッチング機能を使用できる。プラグインは、実行時マッチングを実行するためにイベント ハンドラを提供する必要がある。 **注意：** プラグインが BPM JMS イベント リスナに依存していない場合、イベント ハンドラを用意する必要はありません。 メソッド パラメータの定義は次のとおり。 ■ *eventContext* ―開始ノード イベント コンテキストを示す `com.bea.wlpi.server.plugin.EventContext` オブジェクト。イベント コンテキストは、イベント ウォッチ登録のような実行時イベント関係のサービスへのアクセスを提供する。 ■ *orgExpr* ― ワークフローをインスタンス化するオーガニゼーションの ID を生成するために使用される式を示す `java.lang.String` オブジェクト。 ■ *orgIsExpression* ―指定されたオーガニゼーションが式であるかどうかを示すブール値（式である場合は `true`、式でない場合は `false`）。 プラグイン イベントを処理するイベント ハンドラの定義方法の詳細については、6-1 ページの「プラグイン イベントの処理」を参照。

プラグイン サンプルから抜粋した次のコード リストは、開始ノードのための実行時コンポーネント クラスを定義する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの StartNode.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-23 開始 ノードのための実行時コンポーネント クラスの定義

```
package com.bea.wlpi.tour.po.plugin;

import java.io.IOException;
import com.bea.wlpi.server.plugin.PluginStart2;
.
.
.
public class StartNode extends StartObject implements PluginStart2 {

    public StartNode() {
    }
.
.
.
    public void setTrigger(EventContext context, String orgExpr,
                           boolean orgIsExpr)
            throws WorkflowException {

        System.out.println("SamplePlugin: StartNode - setTrigger called");
        context.postStartWatch(SamplePluginConstants.CONTENTTYPE, eventDesc,
                               null, null);
    }

    public void fixup(ExpressionParser parser) {
    }
}
```

関連する次のコード例も参照してください。

- 4-8 ページの「開始ノードのための PluginObject インタフェースの実装」では、XML フォーマットでプラグイン データを読み込む方法を説明しています。

- 4-17 ページの「開始ノードのための PluginData インタフェースの実装」では、XML フォーマットでプラグイン データを読み込み、保存する方法を説明しています。このクラスは、`StartObject` クラスを拡張します。
- 4-51 ページの「開始ノードのための PluginTriggerPanel クラスの定義」では、設計クライアントでプラグイン GUI コンポーネントを表示する方法を説明しています。
- 4-99 ページの「プラグイン実行時コンテキストの使い方」では、エバリュエータ式から参照できるプラグイン フィールドを定義する方法を説明しています。

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

PluginTemplateNode インタフェース

`com.bea.wlpi.server.plugin.PluginTemplateNode` インタフェースは、完了ノードと開始ノードをアクティブ化するメソッドと、これらの式をコンパイルするためのメソッドを提供します。

`PluginTemplateNode` インタフェースは、次のインタフェースにより拡張されます。

- `com.bea.wlpi.server.plugin.PluginDone`
- `com.bea.wlpi.server.plugin.PluginStart2`

次の表で、完了ノードまたは開始ノードを定義するときに実行時コンポーネント クラスの一部として実装の必要な `PluginTemplateNode` インタフェース メソッドについて説明します。

表 4-17 PluginTemplateNode インタフェース メソッド

メソッド	説明
`public void activate(com.bea.wlpi.server.common.ExecutionContext context) throws com.bea.wlpi.common.WorkflowException`	ノードをアクティブ化する。 プラグイン フレームワークは、マッチング ノードが前のノードからの着信遷移によりアクティブ化されるときに、このメソッドを呼び出す。 メソッド パラメータの定義は次のとおり。 *execContext* ―実行コンテキストを示す `com.bea.wlpi.server.common.ExecutionContext` オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。 実行コンテキストの詳細については、4-99 ページの「プラグイン実行時コンテキストの使い方」を参照。
`public void fixup(com.bea.wlpi.evaluator.ExpressionParser parser) throws com.bea.wlpi.common.WorkflowException`	プラグイン ノードが必要な式をコンパイルできるようにする。 Plug-in Manager は、テンプレート定義を解析して、結果をメモリに格納した後、ワークフローを開始する前に、このメソッドを呼び出す。プラグインの開始ノードと完了ノードは、負荷のかかる初期化手順をすべてこの時点で実行する必要がある。 メソッド パラメータの定義は次のとおり。 *parser* ―式パーサを示す `com.bea.wlpi.evaluator.ExpressionParser` オブジェクト。

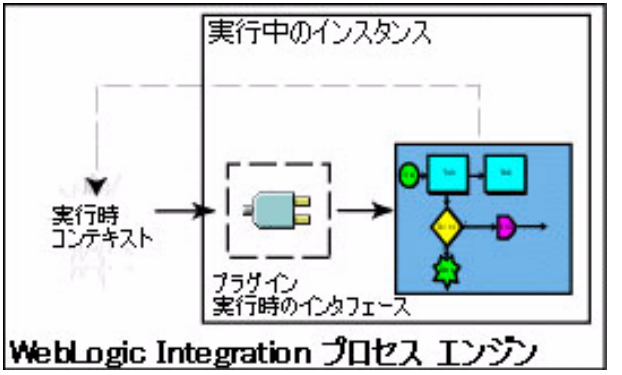
`PluginTemplateNode` インタフェースの詳細については、`com.bea.wlpi.server.plugin.PluginTemplateNode` Javadoc を参照してください。

プラグイン実行時コンテキストの使い方

プラグインに関する実行時実行情報を定義するには、4-63 ページの「プラグインの実行」に説明されているように、プラグイン コンポーネントのための実行時インタフェースを実装する必要があります。実行時には、プラグインは*コンテキスト引き渡し*と呼ばれるプロセスを使用して、プロセス エンジンと通信します。このプロセスで、Plug-in Manager はプラグイン コンポーネント実行時インタフェースのインスタンスを取得し、そのインスタンスにコンテキストを渡します。

次の図に、コンテキスト引き渡しの仕組みを示します。

図 4-10　コンテキスト引き渡し



各コンテキスト インタフェースは、Plug-in Manager の機能に対して限定的アクセスを提供することにより、プラグインが独自のアプリケーション ロジックの実行と管理を行い、BPM 実行時環境にプラグイン インスタンス データを導入できるようにします。

次の表で、プラグイン実行時コンテキスト インタフェースについて説明します。

表 4-18　プラグイン実行時コンテキスト インタフェース

コンテキスト	提供される要素
`com.bea.wlpi.server.plugin.ActionContext`	実行時コンテキスト、およびアクションと関連付けられたサービス。

表 4-18 プラグイン実行時コンテキスト インタフェース （続き）

コンテキスト	提供される要素
com.bea.wlpi.evaluator.EvaluationContext	式の要素のための実行時評価パラメータ。
com.bea.wlpi.server.plugin.EventContext	実行時コンテキスト、およびイベントと関連付けられたサービス。
com.bea.wlpi.server.common.ExecutionContext	実行中のワークフロー インスタンスの実行コンテキスト。
com.bea.wlpi.common.plugin.PluginPanelContext	BPM 設計クライアントのためのクライアント側コンテキスト。

次の節では、コンテキスト インタフェースについて更に詳しく説明します。

アクション コンテキスト

com.bea.wlpi.server.plugin.ActionContext インタフェースは、プラグイン アクションのための実行時コンテキストとサービスを提供します。このコンテキストは、com.bea.wlpi.server.plugin.PluginAction インタフェースの execute() を介して渡されます。

次の表で、アクション コンテキストに関する情報にアクセスするために使用できる ActionContext インタフェース メソッドについて説明します。

表 4-19 ActionContext インタフェース メソッド

メソッド	説明
`public int executeSubActionList(int index, com.bea.wlpi.server.common.ExecutionContext context) throws com.bea.wlpi.common.WorkflowException`	リスト上の各サブアクションを実行する。 メソッド パラメータの定義は次のとおり。 ■ *index* ― 実行するサブアクション リストのインデックスを指定する整数値。この値は、com.bea.wlpi.common.plugin.ActionInfo オブジェクトに対応する classNames 配列のインデックスと同じである。 ■ *execContext* ―呼び出し側により渡される実行コンテキストを示す com.bea.wlpi.server.common.ExecutionContext オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。 このメソッドは、処理を続行するかどうかを指定するリターン コードを示す次のいずれかの com.bea.wlpi.server.common.ExecutionContext 整数値を返す。 ■ CONTINUE ―処理を続行する。 ■ STOP ―処理を停止する。この場合、ただちに呼び出し側に戻って、この戻り値を渡す必要がある。 実行コンテキストの詳細については、4-111 ページの「実行コンテキスト」 を参照。

表 4-19 ActionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String getActionId() throws com.bea.wlpi.common.WorkflowException`	このアクションをユニークに定義する ID を取得する。 この ID は、プラグイン アクションに対して非同期的に実行されるコールバックをサポートするために必要となる。この ID は、`com.bea.wlpi.server.worklist.Worklist` インタフェースの `response()` メソッドに渡す必要がある。 このメソッドは、アクション ID を示す `java.lang.String` オブジェクトを返す。

表 4-19 ActionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String instantiateWorkflow(com.bea.wlpi.server.common.ExecutionContext context, java.lang.String orgID, java.lang.String templateID, com.bea.wlpi.common.VariableInfo[] initialValues, java.util.Map pluginData) throws com.bea.wlpi.common.WorkflowException`	新しいワークフロー インスタンスを作成する。 メソッド パラメータの定義は次のとおり。 ■ *context* ―呼び出し側により渡される実行コンテキストを示す com.bea.wlpi.server.common.ExecutionContext オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。実行コンテキストの詳細については、4-111 ページの「実行コンテキスト」を参照。 ■ *orgID* ― ワークフローをインスタンス化する必要のあるオーガニゼーションの ID を示す java.lang.String オブジェクト。 ■ *templateID* ― インスタンス化するワークフロー テンプレートの ID を示す java.lang.String オブジェクト。 ■ *intialValues* ―インスタンス化するワークフロー テンプレートの ID を示す com.bea.wlpi.common.VariableInfo オブジェクトの配列。 ■ *pluginData* ―プラグイン名に対して入力されたすべてのプラグイン インスタンス データのマップを示す java.util.Map オブジェクト。 ■ *start* ―新しいワークフロー インスタンスをただちに開始するか（true）、サスペンド状態のままにするか（false）を示すブール フラグ。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である ClientReq.dtd に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによりアクセスできる。
`public java.lang.String instantiateWorkflow(com.bea.wlpi.server.common.ExecutionContext context, java.lang.String orgID, java.lang.String templateID, com.bea.wlpi.common.VariableInfo[] initialValues, java.util.Map pluginData, boolean start) throws com.bea.wlpi.common.WorkflowException`	

表 4-19 ActionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String startWorkflow(com.bea.wlpi.server.common.ExecutionContext context, java.lang.String istanceID, com.bea.wlpi.server.eventprocessor.EventData eventData) throws com.bea.wlpi.common.WorkflowException`	前にインスタンス化されたワークフローを開始する。ワークフローは、サスペンド状態でインスタンス化されている必要がある。 メソッド パラメータの定義は次のとおり。 ■ *context* ―呼び出し側により渡される実行コンテキストを示す com.bea.wlpi.server.common.ExecutionContext オブジェクト。実行コンテキストは、テンプレート ID、テンプレート定義 ID、ワークフロー インスタンス ID、イベント データ、ワークフロー実行に関係する各種サービスなどの実行時コンテキストへのアクセスを提供する。実行コンテキストの詳細については、4-111 ページの「実行コンテキスト」 を参照。 ■ *instanceID* ― 開始するワークフロー インスタンスの ID を示す java.lang.String オブジェクト。 ■ *eventData* ―呼び出された開始ノードを表す com.bea.wlpi.server.eventprocessor.EventData オブジェクト。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である ClientReq.dtd に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによりアクセスできる。

評価コンテキスト

com.bea.wlpi.evaluator.EvaluationContext インタフェースは、式の要素のための実行時評価パラメータを提供します。このコンテキストは、com.bea.wlpi.server.plugin.PluginField インタフェースと com.bea.wlpi.server.plugin.PluginFunction インタフェースの evaluate() メソッドを介して渡されます。

次の表で、評価コンテキストに関する情報にアクセスするために使用できる EvaluationContext インタフェース メソッドについて説明します。

表 4-20 EvaluationContext インタフェース メソッド

メソッド	説明
public final int getCalendarType()	日付演算を実行するときに使用するカレンダーのタイプを取得する。 このメソッドは、com.bea.wlpi.evaluator.ExecutionContext インタフェースにより定義されているように、次のいずれかのカレンダー タイプを示す整数値を返す。 ■ CALTYPE_ASSIGNEE (1) ―タスクの受託者のカレンダー ■ CALTYPE_GREGORIAN (3) ―グレゴリオ暦 ■ CALTYPE_ORG (0) ―インスタンス オーガニゼーションのカレンダー ■ CALTYPE_SPECIFIC (2) ―固有のカレンダー
public final com.bea.wlpi.server.eventprocessor.EventData getEventData()	現在のイベントのためのデータを取得する。 このメソッドは、現在のイベント データを示す com.bea.wlpi.server.eventprocessor.EventData オブジェクトを返す。

表 4-20 EvaluationContext インタフェース メソッド （続き）

メソッド	説明
`public final com.bea.wlpi.server.common.ExecutionContext getExecutionContext()`	式を評価する実行コンテキストを取得する。 このメソッドは、実行コンテキストを示す `com.bea.wlpi.server.common.ExecutionContext` オブジェクトを返す。実行コンテキストの詳細については、4-111 ページの「実行コンテキスト」を参照。
`public final java.lang.String getTaskID()`	現在のタスクのユーザ ID があれば、それを取得する。 このメソッドは、タスク ID を示す `java.lang.String` オブジェクトを返す。
`public final java.lang.String getUserID()`	現在のトップ レベル API 呼び出しを行ったユーザ ID を取得する。 このメソッドは、ユーザ ID を示す `java.lang.String` オブジェクトを返す。

イベント コンテキスト

`com.bea.wlpi.server.plugin.EventContext` インタフェースは、プラグイン イベントに対して実行時コンテキストとサービスを提供します。このコンテキストは、次のメソッドを介して渡されます。

- `com.bea.wlpi.server.plugin.PluginEvent` インタフェースの `activate()` メソッドと `trigger()` メソッド
- `com.bea.wlpi.server.plugin.PluginStart2` インタフェースの `settrigger()` メソッド

次の表で、イベント コンテキストに関する情報にアクセスするために使用できる `EventContext` インタフェース メソッドについて説明します。

表 4-21 EventContext インタフェース メソッド

メソッド	説明
`public void activateEvent(com.bea.wlpi.server.common.ExecutionContext` *`context`*`, java.lang.String` *`contentType`*`, java.lang.String` *`eventDescriptor`*`, java.lang.String` *`keyValue`*`, java.lang.String` *`condition`*`) throws com.bea.wlpi.common.WorkflowException`	デフォルトのイベント アクティブ化を実行する。 Event Processor がこのワークフロー インスタンスまたはテンプレートに送信されるメッセージを受信し、保持しているかどうかを確認する。メッセージが存在しない場合、イベント ウォッチ レコードがポストされる。メッセージが存在する場合、次の基準を満たしていれば、マッチング イベントが消費され、イベント ノードがトリガされる。 ■ メッセージに、必要なコンテンツ タイプとイベント記述子が入っている。 ■ キー値が呼び出し側により指定されたキー値と一致する（指定されている場合）。 ■ 条件式が `true` と評価される（指定されている場合）。 `text/xml` 以外のコンテンツ タイプを渡す場合、プラグインは、マッチング イベント キー式をイベント キー テーブルに登録しておく必要がある。これは、`com.bea.wlpi.server.admin.Admin` インタフェースの `addEventKey()` メソッドを使用することで行える。プラグインは、このコンテンツ タイプとフォーマットの着信データからのキー値を評価するためのプラグイン フィールドも提供する必要がある。プラグイン フィールドの定義方法の詳細については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照。 メソッド パラメータの定義は次のとおり。 ■ *`context`* ―呼び出し側により渡される実行コンテキストを示す `com.bea.wlpi.server.common.ExecutionContext` オブジェクト。実行コンテキストの詳細については、4-111 ページの「実行コンテキスト」を参照。 ■ *`contentType`* ―メッセージの基本データ タイプを記述する MIME コンテンツ タイプを示す `java.lang.String` オブジェクト。 ■ *`eventDescriptor`* ― `contentType` 値に対応のフォーマットのイベント記述子を示す `java.lang.String` オブジェクト、または null。 ■ *`keyValue`* ―この呼び出しをトリガするために必要なキー値を示す `java.lang.String` オブジェクト、または null。 ■ *`condition`* ―イベントに対して評価する条件式を示す `java.lang.String` オブジェクト、または null。この条件が true として評価されないと、イベントをトリガできない。

表 4-21 EventContext インタフェース メソッド （続き）

メソッド	説明
`public void checkEventKey(java.lang.String` *`contentType`*`, java.lang.String` *`eventDescriptor`*`, java.lang.String` *`keyExpr`*`) throws com.bea.wlpi.common.WorkflowException`	適切なイベント キーが存在することを確認する。 このメソッドは、その親である `com.bea.wlpi.common.plugin.StartInfo` オブジェクトまたは `com.bea.wlpi.common.plugin.EventInfo` オブジェクトの `getFieldInfo()` メソッドを呼び出す。このメソッドは、`FieldInfo` オブジェクトを使用して、適切なイベント キーの存在を確認し、必要であれば、イベント キーの作成または更新を行う。 メソッド パラメータの定義は次のとおり。 ■ *`contentType`* ―メッセージの基本データ タイプを記述する MIME コンテンツ タイプを示す `java.lang.String` オブジェクト。 ■ *`eventDescriptor`* ― `contentType` 値に対応のフォーマットのイベント記述子を示す `java.lang.String` オブジェクト、または null。 ■ *`keyExpr`* ―指定されたコンテンツ タイプとイベント記述子の着信メッセージからキー値を抽出するために使用する式を示す `java.lang.String` オブジェクト。この値は null に設定することもできる。 *`keyExpr`* が null でない場合、メソッドは、イベント・キーが存在しなければ新しいイベント キーを作成し、既存のイベント キーの *`keyExpr`* がプラグインにより提供されたものと一致しなければイベント キーを更新する。*`keyExpr`* が null であり、マッチング イベント キーが存在しない場合、メソッドは、コード `Messages#EVENT_KEY_MISSING` で `WorkflowException` を生成する。このメソッドを呼び出すことができるのは、`com.bea.wlpi.server.plugin.PluginStart2` ノードまたは `com.bea.wlpi.server.plugin.PluginEvent` ノードのみである。 **注意：** これは負荷の高い呼び出しであるので、プラグインは、その使用を最小限にとどめるように手順を組む必要があります。通常は、静的フラグを維持することにより、イベント キーごとに 1 回だけ呼び出すようにします。

表 4-21 EventContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String getNodeId()`	イベント ノードまたは開始ノードの ID を取得する。 この ID は、プラグイン アクションに対して非同期的に実行されるコールバックをサポートするために必要となる。この ID は、`com.bea.wlpi.server.worklist.Worklist` インタフェースの `response()` メソッドに渡す必要がある。 このメソッドは、アクション ID を示す `java.lang.String` オブジェクトを返す。
`public java.lang.String getTemplateDefinitionID()`	ワークフロー テンプレート定義の ID を取得する。 このメソッドは、テンプレート定義 ID を示す `java.lang.String` オブジェクトを返す。
`public java.lang.String getTemplateID()`	ワークフロー テンプレートの ID を取得する。 このメソッドは、テンプレート ID を示す `java.lang.String` オブジェクトを返す。

表 4-21 EventContext インタフェース メソッド （続き）

メソッド	説明
`public void postStartWatch(java.lang.String` *`contentType`*`, java.lang.String` *`eventDescriptor`*`, java.lang.String` *`keyValue`*`, java.lang.String` *`condition`*`)`	指定されたメッセージに関する Event Processor ウォッチ レコードを登録する。 **注意：** このメソッドを呼び出すことができるのは、開始ノードのみです。 `text/xml` 以外のコンテンツ タイプを渡す場合、プラグインは、マッチング イベント キー式がイベント キー テーブルに登録されていることを確認する必要がある。これは、`com.bea.wlpi.server.admin.Admin` インタフェースの `addEventKey()` メソッドを使用することで行える。プラグインは、このコンテンツ タイプとフォーマットの着信データからのキー値を評価するためのプラグイン フィールドも提供する必要がある。プラグイン フィールドの定義方法の詳細については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照。 メソッド パラメータの定義は次のとおり。 ■ *`contentType`* ― メッセージの基本データ タイプを記述する MIME コンテンツ タイプを示す `java.lang.String` オブジェクト。 ■ *`eventDescriptor`* ― `contentType` 値に対応のフォーマットのイベント記述子を示す `java.lang.String` オブジェクト、または null。 ■ *`keyValue`* ― この呼び出しをトリガするために必要なキー値を示す `java.lang.String` オブジェクト、または null。 ■ *`condition`* ― イベントに対して評価する条件式を示す `java.lang.String` オブジェクト、または null。この条件が true として評価されないと、イベントをトリガできない。

表 4-21 EventContext インタフェース メソッド （続き）

メソッド	説明
`public void removeEventWatch(com.bea.wlpi.server.common.ExecutionContext context)`	指定されたメッセージに関する Event Processor ウォッチ レコードの登録を解除する。 **注意：** このメッセージを呼び出せるのはイベント ノードのみです。 メソッド パラメータの定義は次のとおり。 *context* ―呼び出し側により渡される実行コンテキストを示す `com.bea.wlpi.server.common.ExecutionContext` オブジェクト。実行コンテキストの詳細については、4-111 ページの「実行コンテキスト」 を参照。
`public void removeStartWatch()`	指定されたメッセージに関する Event Processor ウォッチ レコードの登録を解除する。 **注意：** このメッセージを呼び出せるのは 開始ノードのみです。

実行コンテキスト

`com.bea.wlpi.server.common.ExecutionContext` インタフェースは、実行中のワークフロー インスタンスのための実行時コンテキストとサービスを提供します。このコンテキストは、次のメソッドを介して渡されます。

- `com.bea.wlpi.server.plugin.PluginAction` インタフェースの `execute()` メソッド、`response()` メソッド、および `startedWorkflowDone()` メソッド
- `com.bea.wlpi.server.plugin.PluginEvent` インタフェースの `activate()` メソッドおよび `trigger()` メソッド
- `com.bea.wlpi.server.plugin.PluginTemplateNode` インタフェースの `activate()` メソッド （開始ノードと完了ノードにより使用される）

次の表で、アクション コンテキストに関する情報にアクセスするために使用できる `ExecutionContext` インタフェース メソッドについて説明します。

表 4-22 ExecutionContext インタフェース メソッド

メソッド	説明
`public void addClientResponse(java.lang.String xml)`	API メソッド戻り値に XML ドキュメントを付加する。 メソッド パラメータの定義は次のとおり。 *xml* ―付加する XML ドキュメントを示す `java.lang.String` オブジェクト。
`public java.lang.String getErrorHandler() throws com.bea.wlpi.common.WorkflowException`	現在のエラー ハンドラの名前を取得する。 このメソッドは、現在のエラー ハンドラを示す `java.lang.String` オブジェクトを返す。
`public com.bea.wlpi.server.eventprocessor.EventData getEventData()`	現在のイベントと関連付けられたデータがあれば、それを取得する。 このメソッドは、イベント データを示す `com.bea.wlpi.server.eventprocessor.EventData` オブジェクト、またはシステム イベント ハンドラを示す空の文字列を返す。
`public int getExceptionNumber()`	イベント ハンドラにより処理されるエラーのメッセージ番号を取得する。 このメソッドは、メッセージ番号を示す整数値を返す。
`public java.lang.Exception getExceptionObject()`	イベント ハンドラにより処理される例外オブジェクトを取得する。 このメソッドは、例外オブジェクトを示す `java.lang.Exception` オブジェクトを返す。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public int getExceptionSeverity()`	イベント ハンドラにより処理されるエラーの例外重大度コードを取得する。 このメソッドは、次のいずれかの `com.bea.wlpi.common.WorkflowException` 重大度コードを示す整数値を返す。 ■ `ERROR_CUSTOM` ―アプリケーションにより、または `Invoke Error handler` アクションを実行中のワークフローにより生成されたカスタム エラー。 ■ `ERROR_SYSTEM` ―ユーザ要求の処理中に発生した致命的例外。 ■ `ERROR_UNKNOWN` ―未知のエラー タイプ（内部的使用のみ）。 ■ `ERROR_WORKFLOW` ―ワークフロー状態の矛盾など、致命的な無効条件。 ■ `WARNING_WORKFLOW` ―ユーザが手動で訂正できる致命的でないワークフロー条件。
`public java.lang.String getExceptionText()`	イベント ハンドラにより処理される例外のメッセージ テキストを取得する。 このメソッドは、メッセージ テキストを示す `java.lang.String` オブジェクトを返す。
`public java.lang.String getExceptionType()`	イベント ハンドラにより処理される例外のメッセージ タイプを取得する。 このメソッドは、メッセージ タイプを示す `java.lang.String` オブジェクトを返す。
`public java.lang.String getInstanceID()`	現在のワークフロー インスタンスの ID を取得する。 このメソッドは、ID を示す `java.lang.String` オブジェクトを返す。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String getOrg() throws com.bea.wlpi.common.WorkflowException`	現在のインスタンスが実行されているオーガニゼーションの ID を取得する。 このメソッドは、オーガニゼーションの ID を示す `java.lang.String` オブジェクトを返す。
`public java.lang.Object getPluginInstanceData(java.lang.String *pluginName*) throws com.bea.wlpi.common.WorkflowException`	名前付きプラグインにより提供されるワークフロー インスタンス データを取得する。 メソッド パラメータの定義は次のとおり。 *`pluginName`* ―プラグイン名を示す `java.lang.String` オブジェクト。 このメソッドは、プラグイン インスタンス データを示す `java.lang.Object` オブジェクトを返す。
`public java.lang.String getRequestor()`	現在の API 要求を行ったユーザの ID を取得する。 このメソッドは、要求者の ID を示す `java.lang.String` オブジェクトを返す。
`public boolean getRollbackOnly()`	現在のユーザ トランザクションがロールバック専用としてマークされているかどうかを調べる。 このメソッドは、トランザクションがロールバック専用として設定されている場合は `true` を、そうでない場合は `false` を返す。
`public java.lang.String getTemplateDefinitionID()`	現在のテンプレート定義の ID を取得する。 このメソッドは、テンプレート定義 ID を示す `java.lang.String` オブジェクトを返す。
`public com.bea.wlpi.common.plugin.PluginObject getTemplateDefintionPluginData(java.lang.String *pluginName*)`	指定されたプラグインに関するテンプレート定義データを取得する。 メソッド パラメータの定義は次のとおり。 *`pluginName`* ―プラグイン名を示す `java.lang.String` オブジェクト。 このメソッドは、テンプレート定義データを示す `com.bea.wlpi.common.plugin.PluginObject` オブジェクトを返す。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String getTemplateID()`	現在のテンプレートの ID を取得する。 このメソッドは、テンプレート ID を示す `java.lang.String` オブジェクトを返す。
`public com.bea.wlpi.common.plugin.PluginObject getTemplatePluginData(java.lang.String pluginName)`	指定されたプラグインに関するテンプレート データを取得する。 メソッド パラメータの定義は次のとおり。 *pluginName* ―プラグイン名を示す `java.lang.String` オブジェクト。 このメソッドは、テンプレート データを示す `com.bea.wlpi.common.plugin.PluginObject` オブジェクトを返す。
`public com.bea.wlpi.common.VariableInfo getVariableInfo(java.lang.String name)`	指定されたプラグイン変数に関する情報を取得する。 メソッド パラメータの定義は次のとおり。 *name* ― 変数名を示す `java.lang.String` オブジェクト。 このメソッドは、変数情報を示す `com.bea.wlpi.common.VariableInfo` オブジェクトを返す。
`public java.lang.Object getVariableValue(java.lang.String name) throws com.bea.wlpi.common.WorkflowException`	指定されたプラグイン変数に関する変数値を取得する。 メソッド パラメータの定義は次のとおり。 *name* ― 変数名を示す `java.lang.String` オブジェクト。 このメソッドは、変数情報を示す `java.lang.Object` オブジェクトを返す。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時の変数のモニタリング」に説明されているように、`com.bea.wlpi.server.admin.Admin` インタフェースの `getInstanceVariable()` メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String instantiate(java.lang.String orgID, java.lang.String initialNode, java.lang.String parentTemplateDefinitionID, java.lang.String parentID, java.lang.String parentNodeID, com.bea.wlpi.server.eventprocessor.EventData eventData, java.util.List lVariableValues, java.util.Map pluginData) throws com.bea.wlpi.common.WorkflowException`	新しいワークフロー インスタンスを作成する。 メソッド パラメータの定義は次のとおり。 ■ *orgID* ―新しいインスタンスと関連付けられるオーガニゼーション ID を示す `java.lang.String` オブジェクト。 ■ *initialNode* ―アクティブ化する開始ノードの ID を示す `java.lang.String` オブジェクト。 ■ *parentTemplateDefinitionID* ―親テンプレート定義の ID を示す `java.lang.String` オブジェクト（サブワークフローがインスタンス化される場合）。 ■ *parentID* ― 親ワークフロー インスタンスの ID を示す `java.lang.String` オブジェクト（サブワークフローがインスタンス化される場合）。 ■ *parentNodeID* ―ワークフローのライフサイクル中にイベントを通知される親ワークフローのノードの ID を示す `java.lang.String` オブジェクト（サブワークフローがインスタンス化される場合）。 ■ *eventData* ―ワークフローで呼び出された開始ノードに渡すイベント データを示す `com.bea.wlpi.server.eventprocessor.EventData` オブジェクト（このパラメータは、*lVariableValues* 値を使用して変数値を明示的に設定する方法の代わりとなる）。 ■ *lVariableValues* ―ワークフロー インスタンス変数を初期化する `com.bea.wlpi.common.VariableInfo` オブジェクトのリストを示す `java.util.List` オブジェクト。すべての必須入力変数の null でない初期値は、このパラメータを通じて渡す必要がある。 ■ *pluginData* ―プラグイン名に対して入力されたすべてのプラグイン インスタンス データのマップを示す `java.util.Map` オブジェクト。 このメソッドは、新しいワークフロー インスタンスの ID を示す `java.lang.String` オブジェクトを返す。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「手動によるワークフローの開始」に説明されているように、`com.bea.wlpi.server.worklist.Worklist` インタフェースの `instantiateWorkflow()` メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public int invokeErrorHandler(java.lang.String handlerName, java.lang.Exception e)`	指定されたイベント ハンドラを呼び出す。 メソッド パラメータの定義は次のとおり。 ■ *handlerName* ―呼び出すイベント ハンドラの名前を示す `java.lang.String` オブジェクト。 ■ *e* ―発生するイベント値を示す `java.lang.Exception` オブジェクト。 このメソッドは、呼び出しのステータスを示す整数値を返す。
`public boolean isAuditEnabled()`	現在のワークフローに対して監査が有効かどうかを調べる。 このメソッドは、監査が有効な場合は `true` を、そうでない場合は `false` を返す。
`public void setErrorHandler(java.lang.String handlerName) throws com.bea.wlpi.common.WorkflowException`	現在のイベント ハンドラを設定する。 メソッド パラメータの定義は次のとおり。 *handlerName* ―設定するイベント ハンドラの名前、null（前のイベント ハンドラを復元する場合）、または空の文字列（システム イベント ハンドラに設定する場合）のいずれかを示す `java.lang.String` オブジェクト。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public void setPluginInstanceData(java.lang.String pluginName, java.lang.Object data) throws com.bea.wlpi.common.WorkflowException`	指定されたプラグインに関するワークフロー インスタンス データを設定する。 メソッド パラメータの定義は次のとおり。 ■ *pluginName* ―プラグイン名を示す java.lang.String オブジェクト。 ■ *data* ―プラグイン データを示す java.lang.Object オブジェクト。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時の変数のモニタリング」に説明されているように、com.bea.wlpi.server.admin.Admin インタフェースの setInstanceVariable() メソッドも参照。
`public void setRollbackOnly()`	ユーザ トランザクションをロールバック専用として設定する。
`public void setVariableValue(java.lang.String name, java.lang.Object value) throws com.bea.wlpi.common.WorkflowException`	変数の値を設定する。 メソッド パラメータの定義は次のとおり。 ■ *orgID* ― 新しいインスタンスと関連付けられるオーガニゼーション ID を示す java.lang.String オブジェクト。 ■ *initialNode* ―アクティブ化する開始ノードの ID を示す java.lang.String オブジェクト。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時の変数のモニタリング」に説明されているように、com.bea.wlpi.server.admin.Admin インタフェースの setInstanceVariable() メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String taskAssign(java.lang.String instanceID, java.lang.String taskID, java.lang.String assigneeID, boolean bRole, boolean bLoadBalance) throws com.bea.wlpi.common.WorkflowException`	参加コンポーネントにワークフロー タスクを割り当てる。 メソッド パラメータの定義は次のとおり。 ■ *instanceID* ― ワークフロー インスタンス ID を示す java.lang.String オブジェクト。 ■ *taskID* ―タスク ID を示す java.lang.String オブジェクト。 ■ *assigneeID* ― 割り当て先（ユーザまたはロール）の ID を示す java.lang.String オブジェクト。 ■ *bRole* ―指定された割り当て先がロールであるか（true）、ユーザであるか（false）を示すブール値。 ■ *bLoadBalance* ― ロールのメンバー間にロード バランシングを適用する必要があるかどうかを示すブール値。このパラメータは、*bRole* が false に設定されている場合は無視される。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である ClientReq.dtd に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによってアクセスできる。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時タスクの管理」に説明されているように、 com.bea.wlpi.server.worklist.Worklist インタフェースの taskAssign() メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String taskDoIt(java.lang.String instanceID, java.lang.String taskID) throws com.bea.wlpi.common.WorkflowException`	ワークフロー タスクを実行する。 メソッド パラメータの定義は次のとおり。 ■ *instanceID* ―ワークフロー インスタンス ID を示す `java.lang.String` オブジェクト。 ■ *taskID* ―タスク ID を示す `java.lang.String` オブジェクト。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である `ClientReq.dtd` に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによってアクセスできる。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時タスクの管理」に説明されているように、`com.bea.wlpi.server.worklist.Worklist` インタフェースの `taskExecute()` メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String taskMarkDone(java.lang.String instanceID, java.lang.String taskID) throws com.bea.wlpi.common.WorkflowException`	ワークフロー タスクを完了してマークする。 メソッド パラメータの定義は次のとおり。 ■ *instanceID* ―ワークフロー インスタンス ID を示す `java.lang.String` オブジェクト。 ■ *taskID* ―タスク ID を示す `java.lang.String` オブジェクト。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である `ClientReq.dtd` に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによってアクセスできる。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時タスクの管理」に説明されているように、`com.bea.wlpi.server.worklist.Worklist` インタフェースの `taskMarkDone()` メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String taskSetProperties(java.lang.String *instanceID*, java.lang.String *taskID*, int *priority*, boolean *doneWithoutExecute*, boolean *executeIfDone*, boolean *unmarkDone*, boolean *modifiable*, boolean *reassignment*) throws com.bea.wlpi.common.WorkflowException`	ワークフロー タスクのプロパティを設定する。 メソッド パラメータの定義は次のとおり。 ■ *instanceID* ―ワークフロー インスタンス ID を示す java.lang.String オブジェクト。 ■ *taskID* ―タスク ID を示す java.lang.String オブジェクト。 ■ *priority* ―タスクの優先順位を 0 (低)、1 (中)、2 (高) で示す整数値。 ■ *doneWithoutExecute* ―この表の前段で説明した taskMarkDone() メソッドを使用してタスクを完了としてマークするパーミッションをユーザが持っているかどうかを示すブール値。 ■ *executeIfDone* ―この表の前段で説明した taskDoIt() メソッドを使用してタスクを実行するパーミッションをユーザが持っているかどうかを示すブール値。 ■ *unmarkDone* ―この表の後段で説明する taskUnmarkDone() メソッドを使用してタスクを未完了としてマークするパーミッションをユーザが持っているかどうかを示すブール値。 ■ *modifiable* ―このメソッドを使用してプロパティを設定するパーミッションをユーザが持っているかどうかを示すブール値。 ■ *reassignable* ―この表の前段で説明した taskAssign() メソッドを使用してタスクを割り当てるパーミッションをユーザが持っているかどうかを示すブール値。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である ClientReq.dtd に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによってアクセスできる。『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時タスクの管理」に説明されているように、com.bea.wlpi.server.worklist.Worklist インタフェースの taskSetProperties() メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String taskUnassign(java.lang.String instanceID, java.lang.String taskID) throws com.bea.wlpi.common.WorkflowException`	ワークフロー タスクの割り当てを解除する。 メソッド パラメータの定義は次のとおり。 ■ *instanceID* ―ワークフロー インスタンス ID を示す `java.lang.String` オブジェクト。 ■ *taskID* ―タスク ID を示す `java.lang.String` オブジェクト。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である `ClientReq.dtd` に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによってアクセスできる。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時タスクの管理」に説明されているように、`com.bea.wlpi.server.worklist.Worklist` インタフェースの `taskUnassign()` メソッドも参照。

表 4-22 ExecutionContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String taskUnmarkDone(java.lang.String instanceID, java.lang.String taskID) throws com.bea.wlpi.common.WorkflowException`	ワークフロー タスクを未完了としてマークする。 メソッド パラメータの定義は次のとおり。 ■ *instanceID* ―ワークフロー インスタンス ID を示す `java.lang.String` オブジェクト。 ■ *taskID* ―タスク ID を示す `java.lang.String` オブジェクト。 このメソッドは、『*BPM クライアント アプリケーション プログラミング ガイド*』の「DTD フォーマット」に説明されているように、クライアント要求 DTD である `ClientReq.dtd` に準拠する XML ドキュメントを返す。この XML ドキュメントには、インスタンス ID やテンプレート定義 ID など、実行中のインスタンスについての情報が入っている。この情報には、SAX (Simple API for XML) パーサなどの XML パーサを使用してドキュメントを解析することによってアクセスできる。 『*BPM クライアント アプリケーション プログラミング ガイド*』の「実行時タスクの管理」に説明されているように、`com.bea.wlpi.server.worklist.Worklist` インタフェースの `taskUnmarkdone()` メソッドも参照。

詳細については、`com.bea.wlpi.server.common.ExecutionContext` Javadoc を参照してください。

PluginPanelContext

`com.bea.wlpi.common.plugin.PluginPanelContext` インタフェースは、設計クライアント（Studio など）のために次のような実行時コンテキストとサービスを提供します。

■ プラグイン テンプレートとテンプレート定義データへのアクセス

- Expression Builder の起動、式の検証と操作、[変数を追加] ダイアログ ボックスの呼び出しなどのサービス

注意： すべてのメソッドを、すべてのダイアログ ボックス コンテキストで使用できるとは限りません。プラグインが無効なコンテキストでメソッドを呼び出した場合、`java.lang.UnsupportedOperationException` 例外が発生します。

プラグイン パネル コンテキストには、4-28 ページの「PluginPanel クラス メソッド」の表に定義されている `com.bea.wlpi.common.plugin.PluginPanel` の get メソッドと set メソッドを使用してアクセスできます。

次の表で、アクション コンテキストに関する情報にアクセスするために使用できる `PluginPanelContext` インタフェース メソッドについて説明します。

表 4-23 PluginPanelContext インタフェース メソッド

メソッド	説明
`public int checkEventKey(java.lang.String *contentType*, java.lang.String *keyExpr*, boolean *update*) throws com.bea.wlpi.common.WorkflowException`	適切なイベント キーが存在することを確認する。 このメソッドは、親である `com.bea.wlpi.common.plugin.PluginTriggerPanel` オブジェクトの eventDescriptor メソッド、または親である `com.bea.wlpi.common.plugin.StartInfo` オブジェクトまたは `com.bea.wlpi.common.plugin.EventInfo` オブジェクトの `getFieldInfo()` メソッドを呼び出す。このメソッドは、戻り値を使用して、適切なイベント キーの存在を確認し、必要であればイベント キーを作成する。 メソッド パラメータの定義は次のとおり。 ■ *contentType* ― メッセージの基本データ タイプを記述する MIME コンテンツ タイプを示す `java.lang.String` オブジェクト。 ■ *keyExpr* ― 指定されたコンテンツ タイプとイベント記述子の着信メッセージからキー値を抽出するために使用する式を示す `java.lang.String` オブジェクト。この値は null に設定することもできる。 ■ *update* ― イベント キーが存在しなければ新たに作成し、*keyExpr* が null でなければ既存のイベント キーを更新するかどうかを示すブール値。 *update* が `true` であり、*keyExpr* が null の場合、ユーザは値を提供するように求められる。そうでない場合、メソッドは新しいイベント キーを作成し、それが有効になったことを示す通知を表示する。 イベント キーがすでに存在するが、*keyExpr* がプラグインにより提供された null でない *keyExpr* と一致しない場合、メソッドはそのイベント キーを更新し、そのことをユーザに通知する。 **注意：** このメソッドに対する呼び出しは、`PluginTriggerPanel` が表示されている間に限り有効です。 このメソッドは、次のいずれかの値を返す。 ■ `EVENT_KEY_CREATED` ― イベント キーが存在しないので作成した。 ■ `EVENT_KEY_EXISTS` ― イベント キーはすでに存在した。 ■ `EVENT_KEY_NOT_EXISTS` ― イベント キーは存在しない。 ■ `EVENT_KEY_UPDATED` ― イベント キーは存在したが、更新された。

表 4-23 PluginPanelContext インタフェース メソッド （続き）

メソッド	説明
`public com.bea.wlpi.common.VariableInfo checkVariable(java.lang.String name, java.lang.String[] validTypes) throws com.bea.wlpi.common.WorkflowException`	変数が存在するかどうかを調べる。変数が存在しない場合、このメソッドは [変数を追加] ダイアログ ボックスを呼び出して、呼び出し側により指定される有効なタイプの新しいワークフロー変数をユーザが作成できるようにする。 メソッド パラメータの定義は次のとおり。 ■ *name* ー 変数名を示す `java.lang.String` オブジェクト。 ■ *validTypes* ー指定された変数の有効なタイプを示す `java.lang.String` オブジェクトの配列、または null。有効なタイプのリストについては、`com.bea.wlpi.common.VariableInfo` Javadoc を参照。 このメソッドは、既存の変数情報または新しい変数情報を示す `com.bea.wlpi.common.VariableInfo` オブジェクト、または null を返す。
`public java.util.Set getEventDescriptors()`	既存のイベント キーにより参照されるイベント記述子の設定を取得する。 このメソッドは、指定されたコンテンツ タイプに関して現在定義されているイベント記述子文字列のセットで構成される `java.util.Set` オブジェクトを返す。このメソッドは、リストとボックスを表示することを目的としている。
`public javax.naming.Context getInitialContext()`	設計クライアントの JNDI コンテキストを取得する。 このメソッドは、JNDI コンテキストを示す `javax.naming.Context` オブジェクトを返す。このコンテキストには、設計クライアントにより使用されるものと同じセキュリティ コンテキストが入っている。 **注意：** 呼び出し側はこのコンテキストを閉じてはなりません。

表 4-23 PluginPanelContext インタフェース メソッド （続き）

メソッド	説明
`public com.bea.wlpi.common.PluginData getPluginTemplateData(java.lang.String pluginName)`	指定されたテンプレートと関連付けられたテンプレートに関するプラグイン データを取得する。 メソッド パラメータの定義は次のとおり。 *pluginName* ―プラグイン名を示す `java.lang.String` オブジェクト。 このメソッドは、プラグイン テンプレートを示す `com.bea.wlpi.common.plugin.PluginData` オブジェクトを返す。
`public com.bea.wlpi.common.PluginData getPluginTemplateDefinitionData(java.lang.String pluginName)`	指定されたテンプレートと関連付けられたテンプレート定義に関するプラグイン データを取得する。 メソッド パラメータの定義は次のとおり。 *pluginName* ― プラグイン名を示す `java.lang.String` オブジェクト。 このメソッドは、プラグイン テンプレート定義を示す `com.bea.wlpi.common.plugin.PluginData` オブジェクトを返す。
`public int getTemplateDefinitionID()`	テンプレート定義のオーナの ID を取得する。 このメソッドは、テンプレート定義 ID を示す `java.lang.String` オブジェクトを返す。
`public int getTemplateID()`	テンプレートのオーナの ID を取得する。 このメソッドは、テンプレート ID を示す `java.lang.String` オブジェクトを返す。
`public java.util.List getVariableList()`	変数とそのタイプのリストを取得する。 このメソッドは、変数を記述する `com.bea.wlpi.common.VariableInfo` オブジェクトのリストの入った `java.util.List` オブジェクトを返す。
`public java.util.List getVariableList(java.lang.String type)`	指定したタイプの変数のリストを取得する。 このメソッドは、指定したタイプの変数を記述する `com.bea.wlpi.common.VariableInfo` オブジェクトのリストの入った `java.util.List` オブジェクトを返す。

表 4-23 PluginPanelContext インタフェース メソッド （続き）

メソッド	説明
`public com.bea.wlpi.common.VariableInfo invokeAddVariableDialog() throws com.bea.wlpi.common.WorkflowException`	ユーザが新しいワークフロー変数を定義できるように、[変数を追加] ダイアログ ボックスを呼び出す。 このメソッドは、新しい変数情報を示す `com.bea.wlpi.common.VariableInfo` オブジェクト、または null を返す。null は、変数を作成するための [OK] ボタンが選択されなかったことを示す。
`public com.bea.wlpi.common.VariableInfo invokeAddVariableDialog(java.lang.String` *`name`*`, java.lang.String[]` *`validTypes`*`) throws com.bea.wlpi.common.WorkflowException`	ユーザが呼び出し側により指定された有効なタイプの新しいワークフロー変数を定義できるように、[変数を追加] ダイアログ ボックスを呼び出す。 メソッド パラメータの定義は次のとおり。 ■ *`name`* ―変数名を示す `java.lang.String` オブジェクト。 ■ *`validTypes`* ―指定された変数の有効なタイプを示す `java.lang.String` オブジェクトの配列、または null。有効なタイプのリストについては、`com.bea.wlpi.common.VariableInfo` Javadoc を参照。 このメソッドは、新しい変数情報を示す `com.bea.wlpi.common.VariableInfo` オブジェクト、または null を返す。null は、変数を作成するための [OK] ボタンが選択されなかったことを示す。

表 4-23 PluginPanelContext インタフェース メソッド （続き）

メソッド	説明
`public void invokeExpressionBuilder(javax.swing.text.JTextComponent *txtInput*, boolean *condition*, com.bea.wlpi.common.plugin.FieldInfo *fieldInfo*, java.lang.String[] *fields*, java.lang.String *eventDescriptor*)`	[Expression Builder] ダイアログ ボックスを呼び出す。 現在の式は、`javax.swing.text.JTextComponent` オブジェクト、または関連付けられたサブクラスに表示する必要があり、クライアントはこのテキスト コンポーネントから Expression Builder を初期化する。 ユーザが [Expression Builder] ダイアログ ボックスで [OK] ボタンを選択すると、クライアントは式を検証し、ダイアログ ボックスを閉じ、修正された式によりテキスト コンポーネントを更新する。 メソッド パラメータの定義は次のとおり。 ■ *txtInput* ―式の入っているテキスト入力コンポーネントを示す `javax.swing.text.JTextComponent` オブジェクト。 ■ *condition* ―条件式を作成するかどうかを示すブール値。 ■ *fieldInfo* ―プラグイン フィールド情報を示す `com.bea.wlpi.common.plugin.FieldInfo` オブジェクト。このパラメータは、フィールドの参照を式に許可する場合に必要。 ■ *fields* ―イベント記述子により指定されるコンポーネントから利用可能なプラグイン フィールド名と一致する有効なプラグイン フィールド名を表す `java.lang.String` オブジェクトの配列。修飾子を必要とするフィールド タイプは、このパラメータで指定するには適していない。 ■ *eventDescriptor* ―イベント記述子を示す `java.lang.String` オブジェクト。

表 4-23 PluginPanelContext インタフェース メソッド （続き）

メソッド	説明
`public boolean isVariableInExpression(java.lang.String expr, java.lang.String var) throws com.bea.wlpi.evaluator.EvaluatorExpression`	指定した変数を式が参照するかどうかを調べる。 メソッド パラメータの定義は次のとおり。 ■ *expr* ―式を示す `java.lang.String` オブジェクト。 ■ *var* ―変数名を示す `java.lang.String` オブジェクト。 このメソッドは、変数が参照されている場合は `true` を、そうでない場合は `false` を返す。
`public java.lang.String renameVariableInExpression(java.lang.String expr, java.lang.String oldName, java.lang.String newName) throws com.bea.wlpi.evaluator.EvaluatorExpression`	名前が変更された変数への式の参照を更新する。 メソッド パラメータの定義は次のとおり。 ■ *expr* ―式を示す java.lang.String オブジェクト。 ■ *oldName* ―古い変数名を示す `java.lang.String` オブジェクト。 ■ *newName* ―新しい変数名を示す `java.lang.String` オブジェクト。 このメソッドは、更新された式テキストを示す `java.lang.String` オブジェクトを返す。

表 4-23 PluginPanelContext インタフェース メソッド （続き）

メソッド	説明
`public java.lang.String validateExpression(java.lang.String *expression*, boolean *allowVariables*, com.bea.wlpi.common.plugin.FieldInfo *fieldInfo*, java.lang.String *eventDescriptor*) throws com.bea.wlpi.evaluator.EvaluatorExpression`	名前が変更された変数への式の参照を更新する。 メソッド パラメータの定義は次のとおり。 ■ *expression* ー 検証する式のテキストを示す `java.lang.String` オブジェクト。 ■ *allowVariables* ー式で変数を使用できるかどうかを示すブール値。 ■ *fieldInfo* ープラグイン フィールド タイプを示す `com.bea.wlpi.common.plugin.FieldInfo` オブジェクト。 ■ *eventDescriptor* ーイベント記述子を示す `java.lang.String` オブジェクト。 このメソッドは、更新された式テキストを示す `java.lang.String` オブジェクトを返す。

詳細については、`com.bea.wlpi.server.plugin.PluginPanelContext` Javadoc を参照してください。

プラグイン コンポーネント値オブジェクトの定義

最後の手順として、さらにコンポーネント データを定義するため、プラグイン コンポーネントの値オブジェクトを定義します。プラグイン コンポーネント値オブジェクトを定義するには、関連付けられているコンストラクタを使用します。2-5 ページの「プラグイン値オブジェクト」の表に記載されているプラグイン値オブジェクトは、それぞれ、オブジェクト データを作成する 1 つまたは複数のコンストラクタを提供します。値オブジェクトを作成するためのコンストラクタの詳細については、B-1 ページの「プラグイン値オブジェクトのまとめ」を参照してください。

3-10 ページの「リモート インタフェース プラグイン情報メソッド」の表の `getPluginCapabilities()` メソッドの説明で定義されているように、`com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` オブジェクトを定義する場合、プラグイン コンポーネントのそれぞれについてプラグイン値オブジェクトを渡す必要があります。

プラグイン サンプルから抜粋した次のコード リストは、`getPluginCapabilitiesInfo()` メソッドを実装する方法を示しています。この抜粋は、*SAMPLES_HOME*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリの `SamplePluginBean.java` ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 4-24　getPluginCapabilitiesInfo() メソッドの実装

```
public PluginCapabilitiesInfo getPluginCapabilitiesInfo(Locale lc,
        CategoryInfo[] info) {

    PluginInfo pi;
    FieldInfo orderFieldInfo;
    FieldInfo confirmFieldInfo;
    FieldInfo[] fieldInfo;
FunctionInfo fi;
FunctionInfo[] functionInfo;
    StartInfo si;
    StartInfo[] startInfo;
```

```
    EventInfo ei;
    EventInfo[] eventInfo;
    SampleBundle bundle = new SampleBundle(lc);

    log("getPluginCapabilities called");

    pi = createPluginInfo(lc);
    orderFieldInfo =
        new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 3,
                      bundle.getString("orderFieldName"),
                      bundle.getString("orderFieldDesc"),
                      SamplePluginConstants.ORDER_FIELD_CLASSES, false);
    confirmFieldInfo =
        new FieldInfo(SamplePluginConstants.PLUGIN_NAME, 4,
                      bundle.getString("confirmFieldName"),
                      bundle.getString("confirmFieldDesc"),
                     SamplePluginConstants.CONFIRM_FIELD_CLASSES, false);
    fieldInfo = new FieldInfo[]{ orderFieldInfo, confirmFieldInfo };
    ei = new EventInfo(SamplePluginConstants.PLUGIN_NAME, 6,
                       bundle.getString("confirmOrderName"),
                   bundle.getString("confirmOrderDesc"), ICON_BYTE_ARRAY,
                       SamplePluginConstants.EVENT_CLASSES,
                       confirmFieldInfo);
    eventInfo = new EventInfo[]{ ei };
    fi = new FunctionInfo(SamplePluginConstants.PLUGIN_NAME, 7,
                          bundle.getString("calcTotalName"),
                          bundle.getString("calcTotalDesc"),
                          bundle.getString("calcTotalHint"),
                          SamplePluginConstants.FUNCTION_CLASSES, 3, 3);
    functionInfo = new FunctionInfo[]{ fi };
    si = new StartInfo(SamplePluginConstants.PLUGIN_NAME, 5,
                       bundle.getString("startOrderName"),
                  bundle.getString("startOrderDesc"), ICICON_BYTE_ARRAYON,
                     SamplePluginConstants.START_CLASSES, orderFieldInfo);
    startInfo = new StartInfo[]{ si };

    PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
                                     getCategoryInfo(bundle), eventInfo,
                                     fieldInfo, functionInfo, startInfo,
                                     null, null, null, null, null);

    return pci;
}
```

5　プラグイン通知の使い方

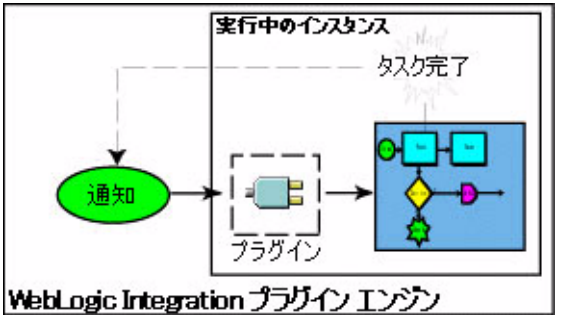
この章では、プラグイン通知を使用する方法について説明します。この章の内容は以下のとおりです。

- 概要
- 通知リスナとしてのプラグインの登録
- 受信した通知に関する情報の取得
- 通知リスナとしてのプラグインの登録解除

概要

BPM は、*通知*のブロードキャストによりプラグインと通信します。通知とは、次の図に示すように、イベントが発生したことを WebLogic Integration プロセス エンジンがプラグインに送信するメッセージです。

図 5-1　WebLogic Integration プロセス エンジンによりプラグインに送信される通知



プラグインを通知リスナとして登録することにより、`com.bea.wlpi.server.plugin` パッケージの一部として提供される最大 4 タイプの通知を受信できます。

注意： パフォーマンスへの影響を最小限に抑えるため、受信が不可欠な通知についてのみ、プラグインを登録してください。特に、実行時インスタンス通知とタスク通知を受信するために登録する場合は注意してください。

次の表では、以下について定義します。

- 4 タイプの通知
- 通知送信を発生させる関連イベント
- 各イベントに対応する `com.bea.wlpi.common.plugin.PluginConstants` インタフェースの整数値。5-4 ページの「通知リスナとしてのプラグインの登録」に説明されているように、プラグインを通知リスナとして登録するときに、この値を指定できます。

表 5-1 通知タイプと関連イベント

通知タイプ	説明	関連イベント	PluginConstants 値
`InstanceNotification`	ワークフロー インスタンス通知	ワークフロー インスタンスの打切り	`INSTANCE_ABORTED`
		ワークフロー インスタンスの完了	`INSTANCE_COMPLETED`
		ワークフロー インスタンスの作成	`INSTANCE_CREATED`
		ワークフロー インスタンスの削除	`INSTANCE_DELETED`
		ワークフロー インスタンスの更新	`INSTANCE_UPDATED`

表 5-1 通知タイプと関連イベント （続き）

通知タイプ	説明	関連イベント	PluginConstants 値
TaskNotification	タスク通知	タスクの割当て	TASK_ASSIGNED
		タスクの完了	TASK_COMPLETED
		タスクの実行	TASK_EXECUTED
		タスクの期限切れ	TASK_OVERDUE
		タスクの開始	TASK_STARTED
		タスクの割り当て解除	TASK_UNASSIGNED
		タスク完了につきマーク解除	TASK_UNMARKED_DONE
		タスクの更新	TASK_UPDATED
TemplateDefinitionNotification	テンプレート定義通知	テンプレート定義の作成	DEFINITION_CREATED
		テンプレート定義の削除	DEFINITION_DELETED
		テンプレート定義の更新	DEFINITION_UPDATED
TemplateNotification	テンプレート通知	テンプレートの作成	TEMPLATE_CREATED
		テンプレートの削除	TEMPLATE_DELETED
		テンプレートの更新	TEMPLATE_UPDATED

次の節では、プラグインを通知リスナとして登録および登録解除する方法、および受信した通知に関する情報を取得する方法について説明します。

通知リスナとしてのプラグインの登録

通常、プラグインはロード時に自分を通知リスナとして登録する必要があります。たとえば、3-7 ページの「ライフサイクル管理メソッド」に説明されているように、`load()` メソッドを実装する場合、必要であれば、このプラグインを通知リスナとして登録します。

プラグインを通知リスナとして登録するには、プラグインに受信させる通知のタイプに基づいて、次の表に定義されている `com.bea.wlpi.server.plugin.PluginManager` メソッドの 1 つ（または必要な複数のメソッド）を使用します。

注意： Plug-in Manager への接続方法については、2-2 ページの「Plug-in Manager への接続」を参照してください。

表 5-2 プラグイン通知登録メソッド

プラグインを登録するリスナ	使用するメソッド
`InstanceNotification` リスナ	`public void addInstanceListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`
`TaskNotification` リスナ	`public void addTaskListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`
`TemplateDefinitionNotification` リスナ	`public void addTemplateDefinitionListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`
`TemplateNotification` リスナ	`public void addTemplateListener( com.bea.wlpi.server.plugin.Plugin` *`plugin`*`, int` *`mask`*`) throws java.rmi.RemoteException`

次の表で、プラグイン通知登録メソッドのために値を指定する必要のあるパラメータについて説明します。

表 5-3 プラグイン通知登録メソッド パラメータ

パラメータ	説明
plugin	プラグイン リモート インタフェースを実装する EJBObject。
mask	登録するイベントを指定する整数ビットマスク。有効な値のリストは、5-2 ページの「通知タイプと関連イベント」の表に示した通知 com.bea.wlpi.common.plugin.PluginConstants 値を参照。mask 値は、特定のカテゴリのすべてのイベントをグローバルに登録 / 登録解除するために、それぞれ EVENT_NOTIFICATION_ALL または EVENT_NOTIFICATION_NONE に対して設定することもできる。 この値は、必要なイベント定数に対してビット演算子 OR を実行することにより作成される。

通知リスナ登録メソッドの詳細については、com.bea.wlpi.server.plugin.PluginManagerCfg Javadoc を参照してください。

プラグイン サンプルから抜粋した次のコード リストは、プラグインを*すべての通知タイプ*および*そのすべての関連イベント*の通知リスナとして登録する方法を示しています。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SamplePluginBean.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 5-1 通知リスナとしてのプラグインの登録

```
      .
      .
      .
public void load(PluginObject pluginData) throws PluginException {
      .
      .
```

```
    .
pm.addInstanceListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
pm.addTaskListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
pm.addTemplateDefinitionListener(plugin,
   PluginConstants.EVENT_NOTIFICATION_ALL);
pm.addTemplateListener(plugin, PluginConstants.EVENT_NOTIFICATION_ALL);
    .
    .
    .
 }
```

プラグイン サンプルの詳細については、10-1 ページの「BPM プラグイン サンプル」を参照してください。

受信した通知に関する情報の取得

通知を受信した後、そのソースなど、通知に関する情報を取得できます。

次の表に、受信した通知に関する情報を取得するために利用できるメソッドを示します。これらのメソッドには、すべての通知タイプを拡張する `com.bea.wlpi.server.plugin.PluginNotification` クラスを介してアクセスできます。

表 5-4 一般的なプラグイン通知情報メソッド

メソッド	説明
`public int getEventType()`	イベント タイプを取得する。 このメソッドは、5-2 ページの「通知タイプと関連イベント」の表に定義されているように、関連イベントに対応する `com.bea.wlpi.common.plugin.PluginConstants` インタフェース整数値を返す。

表 5-4 一般的なプラグイン通知情報メソッド （続き）

メソッド	説明
`public java.lang.Object getSource()`	変更されたワークフロー エンティティ オブジェクトを取得する。 このメソッドは、変更されたワークフロー エンティティを示す `java.lang.Object` オブジェクトを返し、`InstanceInfo`、`TaskInfo`、`TemplateDefinitionInfo`、または `TemplateInfo` のいずれかの `com.bea.wlpi.common.plugin` 値オブジェクトとすることができる。それぞれの値オブジェクトに関する情報を取得するために利用できるメソッドについては、B-1 ページの「プラグイン値オブジェクトのまとめ」 を参照。

これらのメソッドの詳細については、`com.bea.wlpi.server.plugin.PluginNotification` Javadoc を参照してください。

さらに、次の表では、変更されたソース コンポーネントに関する追加情報を取得するために各通知タイプに対して利用できるメソッドを示します。

表 5-5 通知タイプに基づくプラグイン通知情報メソッド

通知タイプ	メソッド	説明
`InstanceNotification`	`public com.bea.wlpi.common.InstanceInfo getInstance()`	変更されたワークフロー インスタンスに関する情報を取得する。 このメソッドは、`com.bea.wlpi.common.InstanceInfo` オブジェクトを返す。ワークフロー インスタンスに関する情報にアクセスするには、『*BPM クライアント アプリケーション プログラミング ガイド*』の「値オブジェクトのまとめ」の「InstanceInfo オブジェクト」に説明されている `InstanceInfo` オブジェクト メソッドを使用する。
`TaskNotification`	`public com.bea.wlpi.common.TaskInfo getTask()`	変更されたタスクに関する情報を取得する。 このメソッドは、`com.bea.wlpi.common.TaskInfo` オブジェクトを返す。タスクに関する情報にアクセスするには、『*BPM クライアント アプリケーション プログラミング ガイド*』の「値オブジェクトのまとめ」の「TaskInfo オブジェクト」に説明されている `TaskInfo` オブジェクト メソッドを使用する。

表 5-5 通知タイプに基づくプラグイン通知情報メソッド （続き）

通知タイプ	メソッド	説明
TemplateDefinitionNotification	public com.bea.wlpi.common.TemplateDefinitionInfo getTemplateDefinition()	変更されたテンプレート定義に関する情報を取得する。 このメソッドは、com.bea.wlpi.common.TemplateDefinitionInfo オブジェクトを返す。テンプレート定義に関する情報にアクセスするには、『*BPM クライアント アプリケーション プログラミング ガイド*』の「値オブジェクトのまとめ」の「TemplateDefinitionInfo オブジェクト」に説明されている TemplateDefinitionInfo オブジェクト メソッドを使用する。
TemplateNotification	public com.bea.wlpi.common.TemplateInfo getTemplate()	変更されたテンプレートに関する情報を取得する。 このメソッドは、com.bea.wlpi.common.TemplateInfo オブジェクトを返す。テンプレートに関する情報にアクセスするには、『*BPM クライアント アプリケーション プログラミング ガイド*』の「値オブジェクトのまとめ」の「TemplateInfo オブジェクト」に説明されている TemplateInfo オブジェクト メソッドを使用する。

通知リスナとしてのプラグインの登録解除

通常、プラグインはアンロード時に自分の通知リスナとしての登録を解除する必要があります。たとえば、3-7 ページの「ライフサイクル管理メソッド」に説明されているように、unload() メソッドを実装する場合、必要であれば、このプラグインの通知リスナとしての登録を解除します。

プラグインの通知リスナとしての登録を解除するには、削除する通知リスナのタイプに基づいて、次の表に定義されている com.bea.wlpi.server.plugin.PluginManager メソッドの 1 つ（または必要な複数のメソッド）を使用します。

注意： Plug-in Manager への接続方法については、2-2 ページの「Plug-in Manager への接続」を参照してください。

表 5-6 プラグイン通知登録解除メソッド

プラグインを登録解除するリスナ	使用するメソッド
InstanceNotification リスナ	public void removeInstanceListener(com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException
TaskNotification リスナ	public void removeTaskListener(com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException
TemplateDefinitionNotification リスナ	public void removeTemplateDefinitionListener(com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException
TemplateNotification リスナ	public void removeTemplateListener(com.bea.wlpi.server.plugin.Plugin plugin) throws java.rmi.RemoteException

どの場合も、通知リスナとしての登録を解除するプラグインを識別する com.bea.wlpi.server.plugin.Plugin オブジェクトを指定する必要があります。

次のコード リストは、*すべての通知タイプ*および*そのすべての関連イベント*についてプラグインの通知リスナとしての登録を解除する方法を示しています。重要なコード行は、**太字**で示します。

コード リスト 5-2 通知リスナとしてのプラグインの登録解除

```
public void unload() throws PluginException {
      .
      .
      .
   pm.removeInstanceListener(plugin);
   pm.removeTaskListener(plugin);
   pm.removeTemplateDefinitionListener(plugin);
   pm.removeTemplateListener(plugin);
      .
      .
    }
```

6　プラグイン イベントの処理

この章では、プラグイン イベントを処理する方法について説明します。この章の内容は以下のとおりです。

- プラグイン イベントの概要
- EventData クラス
- プラグイン イベント ハンドラの定義
- プラグイン メッセージ タイプの定義
- イベント ウォッチ エントリの定義
- プラグイン イベント ハンドラに対するイベントの送信

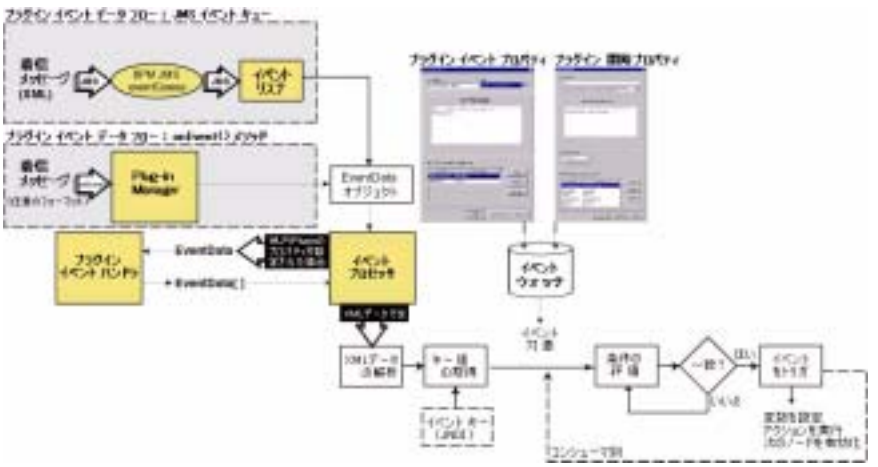
プラグイン イベントの概要

*イベント*とは、別のワークフローから、あるいは別のアプリケーションなどの外部ソースからの非同期通知です。ワークフローの開始、変数の初期化、ワークフロー内のノードのアクティブ化、アクションの実行を行うイベントを定義できます。イベントとイベント プロパティは、WebLogic Integration Studio を使用して、開始ノードとイベント ノードを定義するときに定義します。イベントとイベント プロパティの定義方法の詳細については、『*WebLogic Integration Studio ユーザーズ ガイド*』を参照してください。

BPM フレームワークは、JMS を介して配信される XML フォーマットのイベント メッセージをサポートします。XML フォーマットと非 XML フォーマットの両方のイベント メッセージをサポートするには、*プラグイン イベント*を定義する必要があります。

次の図に、プラグイン イベントのデータ フローを示します。

図 6-1 プラグイン イベント データ フロー



前の図に示したように、プラグイン イベント メッセージは、次の 2 通りの方法のいずれかでシステムに渡されます。

- BPM により提供される JMS eventQueue

  実行時、イベント リスナは、XML フォーマットで受信され、JMS を使用して配信されるイベントをリスンします（JMS eventQueue を介して）。

  eventQueue のために JMS イベント リスナを設定する方法については、『*BPM クライアント アプリケーション プログラミング ガイド*』の「JMS 接続の確立」を参照してください。

- `com.bea.wlpi.server.plugin.PluginManager` EJB の `onEvent()` メソッド

  実行時、Plug-in Manager の `onEvent()` メソッドは、プラグインにより指定されたすべてのデータ フォーマットを受け入れます。

WebLogic Integration プロセス エンジンは、着信プラグイン イベント メッセージを `com.bea.wlpi.server.eventprocessor.EventData` オブジェクトとして格納し、そのオブジェクトを Event Processor に渡します。`EventData` オブジェクトの詳細については、6-5 ページの「EventData クラス」を参照してください。

Event Processor は、`EventData` オブジェクトを受信すると、`WLPIPlugin` プロパティがメッセージの一部として定義されているかどうかをまずチェックします。

`WLPIPlugin` メッセージ プロパティは、次のいずれかの方法で定義できます。

- 6-6 ページの「EventData オブジェクト コンストラクタ」の表に説明されているように、*properties* コンストラクタ パラメータを com.bea.wlpi.server.eventprocessor.EventData オブジェクトに対して定義し、WLPIPlugin というプロパティを含めることにより。
- 次の URL にある『*JMS プログラマーズガイド*』の「WebLogic JMS アプリケーションの開発」の「メッセージ プロパティ フィールドの設定」に説明されているように、WLPIPlugin という JMS メッセージ プロパティを定義することにより。

  http://edocs.beasys.co.jp/e-docs/wls/docs70/jms/implement.html

WLPIPlugin プロパティが定義されている場合、Event Processor は、プラグイン イベント ハンドラに EventData オブジェクトを渡して、プラグイン イベント データを前処理します。前処理されたイベント データは、EventData オブジェクトの配列として Event Processor に返されます。プラグイン イベント ハンドラの定義方法の詳細については、6-12 ページの「プラグイン イベント ハンドラの定義」を参照してください。

次に、Event Processor は、結果のデータが XML フォーマットであるかどうかをチェックし、そうであれば、次の手順を実行します。

1. イベント データを解析します。

   この時点で、Event Processor は、EventData オブジェクトにより提供されるコンテンツ タイプとイベント記述子を取得します。

   コンテンツ タイプは、MIME (Multi-Purpose Internet Mail Exensions) コンテンツ タイプを参照し、メッセージの基本データ タイプを説明します。コンテンツ タイプのデフォルトは text/xml で、データがテキスト フォーマットであり、XML のルールに従っていることを示します。

   イベント記述子は、データ フォーマットの正確な定義を提供し、コンテンツ タイプに基づいて解釈されます。

   たとえば、コンテンツ タイプが text/xml に設定されている場合、イベント記述子は XML ドキュメント タイプです。DOCTYPE タグが設定されている場合、XML ドキュメント タイプは、パブリック ID またはシステム ID（定義されている場合）、あるいはドキュメント要素名です。一方、コンテンツ タイプが application/x-java-object に設定されている場合（シリアライズされた Java オブジェクトを示す）、イベント記述子は Java クラスの完全修飾名です。

2. 指定されている場合、イベント キー値を取得し、それを JNDI に事前定義されているキーと比較します。

   イベント キーは、主キーとして扱う必要のある着信データ フィールドを指定します。また、特定のコンテンツ タイプおよびイベント記述子と関連付けられます。イベント キーは、パフォーマンス向上のために、着信イベント データのフィルタ メカニズムを提供します。イベント キーの定義方法の詳細については、『*WebLogic Integration Studio ユーザーズ ガイド*』を参照してください。

   注意： プラグイン イベント ノードまたは開始ノードが、`com.bea.wlpi.server.plugin.EventContext` インタフェースの `activateEvent()` メソッドまたは `postStartWatch()` メソッドをそれぞれ使用し、`text/xml` 以外のコンテンツ タイプを渡す場合、プラグインは一致するイベント キー式をイベント キー テーブルに登録しておく必要があります。これを行うには、`com.bea.wlpi.server.admin.Admin` インタフェースの `addEventKey()` メソッドを使用します。プラグインは、このコンテンツ タイプとフォーマットの着信データからのキー値を評価するためのプラグイン フィールドも提供する必要があります。プラグイン フィールドの定義方法の詳細については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照してください。

3. コンテンツ タイプ、イベント記述子、およびイベント キー値に基づいて、イベント ウォッチ テーブルでイベント コンテンツの受信候補を検索します。

   プラグイン イベントをイベント ウォッチ テーブルにエントリとして追加する方法の詳細については、6-15 ページの「イベント ウォッチ エントリの定義」を参照してください。

4. 条件が指定されている場合、その条件を評価し、一致するものがあるかどうか調べます。

   条件は、true または false と評価される式で構成されます。条件の使用により、メッセージのコンテンツに基づいて、同じイベントに対して異なる処理を適用する複数ののワークフローを定義できます。たとえば、イベント データに [Order Amount] というフィールドが入っている場合、`Order Amount > $500.00` は有効な条件です。条件の定義方法の詳細については、『*WebLogic Integration Studio ユーザーズ ガイド*』を参照してください。

5. 一致するイベントがある場合、イベントをトリガして、新しいワークフロー インスタンスを開始するか、または既存のワークフロー インスタンスの処理を再開します。

これらの各タスクは、イベントに対してサブスクライブされたコンシューマごとに実行されます。

プラグイン イベントをサポートする手順は次のとおりです。

1. プラグイン イベント データを処理するプラグイン イベント ハンドラを定義します。
2. カスタム メッセージ タイプまたはプラグイン フィールドを定義し、イベント ハンドラが返すイベント データを解析します。
3. イベント ウォッチ エントリを定義します。

次の節では、これらの手順の詳細およびプラグイン イベント ハンドラにメッセージを送信する方法について説明します。

イベント ハンドラを作成する前に、着信プラグイン イベント メッセージを格納する `EventData` コンテナ クラスに関する情報の作成方法とアクセス方法を理解しておく必要があります。

EventData クラス

6-2 ページの「プラグイン イベント データ フロー」の図に示したように、`com.bea.wlpi.server.eventprocessor.EventData` オブジェクトには着信データ メッセージ用のコンテナ クラスがあります。

次の表に、`EventData` オブジェクトを作成するために使用できるコンストラクタを示します。

表 6-1 EventData オブジェクト コンストラクタ

コンストラクタ	説明
`public EventData(org.w3c.dom.Document document)`	XML を含む事前解析された DOM オブジェクトから EventData を作成し、次の値を設定する。 ■ `CONTENT_TYPE_DOM` のコンテンツ タイプ ■ `DOCTYPE` パブリック ID またはシステム ID（定義されている場合）のイベント記述子、あるいはドキュメント要素名。 コンストラクタ パラメータの定義は次のとおり。 *`document`* ― XML コンテンツを示す `org.w3c.dom.Document` オブジェクト。
`public EventData(java.lang.String xml) throws com.bea.wlpi.common.WorkflowException`	XML の入った String オブジェクトから `EventData` オブジェクトを作成し、次の値を設定する。 ■ `CONTENT_TYPE_XML` のコンテンツ タイプ。 ■ `DOCTYPE` パブリック ID またはシステム ID（定義されている場合）のイベント記述子、あるいはドキュメント要素名。 コンストラクタ パラメータの定義は次のとおり。 *`xml`* ― XML コンテンツを示す `java.lang.String` オブジェクト。

表 6-1 EventData オブジェクト コンストラクタ （続き）

コンストラクタ	説明

`public EventData(`
`java.lang.Object` *`content`*`,`
`java.lang.String` *`contentType`*`,`
`java.lang.String` *`eventDescriptor`*`,`
`long` *`expiration`*`, java.lang.String[]`
`templateNames``, java.lang.String[]`
`instanceIDs``, java.util.Map`
`properties``)`
	任意のタイプのオブジェクトを入れる EventData オブジェクトを作成する。
	コンストラクタ パラメータは、次のように定義される。
	■ *`content`* ―コンテンツを示す `java.lang.Object` オブジェクト。
	■ *`contentType`* ―コンテンツ タイプを示す `java.lang.String` オブジェクト。
	■ *`eventDescriptor`* ―イベント記述子を示す `java.lang.String` オブジェクト。
	■ *`expiration`* ― 有効期限を示す long 値。
	■ *`templateNames`* ―イベント データの配信先の名前を示す `java.lang.String` オブジェクトの配列、または null。
	■ *`instanceIDs`* ―イベント データの配信先のワークフロー インスタンスの ID を示す `java.lang.String` オブジェクト、または null。
	■ *`properties`* ―カスタム メッセージ プロパティのマップを示す `java.util.Map` オブジェクト。たとえば、次のカスタム メッセージ プロパティを定義できる。
	- `WLPIContentType` ―コンテンツ タイプ
	- `WLPIEventDescriptor` ―イベント記述子
	- `WLPIPlugin` ―プラグイン名
	注意： `EventData` オブジェクトには、それぞれコンテンツ タイプ、イベント記述子、プラグイン名を定義する `CONTENT_TYPE`、`EVENT_DESCRIPTOR`、`PLUGIN` という定数値もあります。
	`templateNames` または *`instanceIDs`* が null でない場合、イベント データを送信するものとみなされます。

表 6-1 EventData オブジェクト コンストラクタ (続き)

コンストラクタ	説明

`public EventData(`
`java.lang.Object` *`content`*`,`
`java.lang.String` *`contentType`*`,`
`java.lang.String` *`eventDescriptor`*`,`
`java.lang.String` *`startInstanceID`*`,`
`java.util.Map` *`properties`*`)`
	任意のタイプのオブジェクトを入れる EventData オブジェクトを作成する。
	コンストラクタ パラメータは、次のように定義される。
	■ *`content`* ―コンテンツを示す `java.lang.Object` オブジェクト。
	■ *`contentType`* ―コンテンツ タイプを示す `java.lang.String` オブジェクト。
	■ *`eventDescriptor`* ―イベント記述子を示す `java.lang.String` オブジェクト。
	■ *`startInstanceID`* ―イベント データの配信先の開始ワークフロー インスタンスの ID を示す `java.lang.String` オブジェクト、または null。
	■ *`properties`* ―カスタム メッセージ プロパティのマップを示す `java.util.Map` オブジェクト。たとえば、次のカスタム メッセージ プロパティを定義できる。
	- `WLPIContentType` ―コンテンツ タイプ
	- `WLPIEventDescriptor` ―イベント記述子
	- `WLPIPlugin` ―プラグイン名
	注意: `EventData` オブジェクトには、それぞれコンテンツ タイプ、イベント記述子、プラグイン名を定義する `CONTENT_TYPE`、`EVENT_DESCRIPTOR`、`PLUGIN` という定数値もあります。

次の表に、`EventData` オブジェクト情報、その情報を定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスする際に使用するメソッドを示します。

表 6-2 EventData オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
XML コンテンツを入れる org.w3c.dom.Document ドキュメント（DOM）	*document*	public final boolean isDOM()
XML コンテンツを入れる文字列ドキュメント	*xml*	public final boolean isXML()
イベント データ コンテンツを入れる java.lang.Object オブジェクト。	*content*	public final java.lang.Object getContent()
コンテンツ タイプ	*contentType*	public final java.lang.String getContentType()
イベント記述子	*eventDescriptor*	public final java.lang.String getEventDescriptor()
有効期限	*expiration*	public final long getExpiration()
イベント データの送信先のテンプレート名の配列	*templateNames*	public final String[] getTemplateNames()
イベント データの送信先のワークフロー インスタンス ID の配列	*instanceIDs*	public final java.lang.String[] getInstanceIDs()
イベント データの送信先の開始ワークフロー インスタンス ID	*instanceIDs*	public final java.lang.String getStartInstanceID()

表 6-2 EventData オブジェクト情報 (続き)

オブジェクト情報	コンストラクタ パラメータ	get メソッド

プラグイン メッセージ プロパティ	*properties*	`public java.lang.Object getProperty(java.lang.String propertyName) throws com.bea.wlpi.common.WorkflowException`
		
		`public java.util.Set getPropertyNames() throws com.bea.wlpi.common.WorkflowException`

Event Processor により割り当てられるユニークなメッセージ識別子 **注意 :** このメソッドは、送信されるメッセージに対してのみ適用されます。イベント メッセージ データが送信されない場合、このメソッドは null を返します。	該当なし	`public final java.lang.String getMessageID()`
イベント データのキー値 コンテンツ タイプとイベント記述子の組み合わせに対してキー値式が定義されている場合、Event Processor によりキー値が設定される。	該当なし	`public final java.lang.String getKeyValue()`

次の表に、XML データの解析に使用できるメソッドを示します。

表 6-3 XML 解析メソッド

メソッド	説明
`public final void parseXML(boolean validate) throws com.bea.wlpi.common.WorkflowException`	コンテンツが XML である場合、イベント データのコンテンツを解析して `org.w3c.dom.Document` ドキュメント（DOM）に格納し、次の値を設定する。 ■ `#CONTENT_TYPE_DOM` のコンテンツ タイプ ■ `DOCTYPE` パブリック ID またはシステム ID（定義されている場合）に対してイベント記述子、あるいはドキュメント要素名 メソッド パラメータの定義は次のとおり。 *`validate`* ―参照された DTD に対して XML を検証する必要がある場合は true。
`public final void parseXML() throws com.bea.wlpi.common.WorkflowException`	コンテンツが XML である場合、イベント データのコンテンツを解析して、`org.w3c.dom.Document` オブジェクトに格納し、次の値を設定する。 ■ `#CONTENT_TYPE_DOM` のコンテンツ タイプ ■ `DOCTYPE` パブリック ID またはシステム ID（定義されている場合）に対してイベント記述子、あるいはドキュメント要素名 XML は検証されない。

次の節では、着信 `EventData` オブジェクトを前処理するためのプラグイン イベント ハンドラを定義する方法について説明します。

プラグイン イベント ハンドラの定義

6-2 ページの「プラグイン イベント データ フロー」の図に示したように、プラグイン イベント データを前処理するためのプラグイン イベント ハンドラを定義し、Event Processor がデータを解析できるようにする必要があります。

プラグイン イベント ハンドラを定義するには、次のタスクを実行します。

1. 実行ハンドラ コンポーネント クラスを定義します。
2. 手順 1 で定義したイベント ハンドラ オブジェクトの名前を引数としてコンストラクタに渡し、`com.bea.wlpi.server.plugin.EventHandlerInfo` 値オブジェクトを定義します。
3. 手順 2 で定義した `EventHandlerInfo` オブジェクトをコンストラクタ パラメータとしてプラグイン `PluginCapabilitiesInfo` オブジェクトに渡すことにより、イベント ハンドラを登録します。

次の節では、これらの手順についてさらに詳しく説明します。

イベント ハンドラ コンポーネント クラスの定義

プラグイン イベント ハンドラ コンポーネント クラスを定義するには、`com.bea.wlpi.server.plugin.EventHandler` インタフェースを実装します。

次の表で、着信プラグイン イベント データを前処理するために実装する必要のある `EventHandler` インタフェース メソッドについて説明します。

表 6-4 EventHandler インタフェース メソッド

メソッド	説明
`public com.bea.wlpi.server.eventprocessor.EventData[] onEvent(com.bea.wlpi.server.eventprocessor.EventData *eventData*) throws com.bea.wlpi.common.plugin.PluginException`	着信プラグイン イベント データを前処理する。 Event Processor は、特定のプラグインに対して送信されたイベント メッセージ（通常は JMS を介して）を受信すると、このメソッドを呼び出す。 イベント ハンドラは、1 つの着信イベントを、Event Processor により順に処理される複数の発信イベントに変換できる。 メソッド パラメータの定義は次のとおり。 *`eventData`* ―プラグイン イベント データを示す `com.bea.wlpi.server.eventprocessor.EventData` オブジェクト。 このメソッドは、次のいずれかの値を返す。 ■ 変換されたイベント データを含み、適切なコンテンツ タイプおよび属性セットを持つ 1 つまたは複数の `EventData` オブジェクト。 ■ イベント ハンドラが着信プラグイン データを完全に処理し、Event Processor がさらに処理する必要のない場合は null。

イベント ハンドラ値オブジェクトの作成

イベント ハンドラ値オブジェクト `com.bea.wlpi.common.plugin.EventHandlerInfo` を作成するには、B-19 ページの「EventHandlerInfo オブジェクト」に定義されているコンストラクタを使用します。*`classNames`* コンストラクタ パラメータ値として、前の節で定義したイベント ハンドラ コンポーネント クラスの名前を渡す必要があります。

たとえば、次のコード抜粋では、イベント ハンドラ クラス `sample.MyEventHandler` の名前を渡して、新しいイベント ハンドラ値オブジェクトを作成しています。イベント ハンドラ クラスは、6-13 ページの「イベント ハンドラ コンポーネント クラスの定義」に説明されているように、`EventHandler` `com.bea.wlpi.server.plugin.EventHandler` インタフェースを実装します。

```
eh = new EventHandlerInfo(SamplePluginConstants.PLUGIN_NAME,
        bundle.getString("startOrderName"),
        bundle.getString("startOrderDesc")
        sample.MyEventHandler);
eventHandler = new EventHandlerInfo[]{ eh };
```

`EventHandlerInfo` オブジェクトの詳細については、B-19 ページの「EventHandlerInfo オブジェクト」を参照してください。

イベント ハンドラの登録

イベント ハンドラを登録するには、B-31 ページの「PluginCapabilitiesInfo オブジェクト」に定義されているように `com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` 値オブジェクトを作成し、前の節で定義した `EventHandlerInfo` オブジェクトを *`eventHandler`* コンストラクタ パラメータ値として渡します。

たとえば、次のコード抜粋では、引数として `com.bea.wlpi.common.plugin.EventHandlerInfo` オブジェクトの名前を渡して、新しい `PluginCapabilitiesInfo` 値オブジェクトを作成しています。

```
PluginCapabilitiesInfo pci = new PluginCapabilitiesInfo(pi,
      getCategoryInfo(bundle), eventInfo, fieldInfo, functionInfo,
      startInfo, null, null, null, null, eventHandler);
```

PluginCapabilitiesInfo オブジェクトの詳細については、B-31 ページの「PluginCapabilitiesInfo オブジェクト」を参照してください。

プラグイン メッセージ タイプの定義

プロセス エンジンがプラグイン com.bea.wlpi.server.eventprocessor.EventData オブジェクトから情報を抽出できるようにするには、プラグイン メッセージ タイプ、すなわち*プラグイン フィールド*を定義する必要があります。抽出した値は、ワークフロー変数に代入したり、式の一部（キー値や条件式など）として使用できます。

プラグイン メッセージ タイプの定義方法の詳細については、4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」を参照してください。

イベント ウォッチ エントリの定義

イベント ウォッチ エントリにより、Event Processor は着信イベントをプラグイン ノードと照合できます。

次の表に、各プラグイン ノード タイプについて、プロセス エンジンがプラグイン イベント エントリをイベント ウォッチ テーブルに追加する方法を示します。

表 6-5 プラグイン ノード タイプに基づくイベント ウォッチ エントリの追加方法

ノード タイプ	説明
イベント ノード	ワークフロー プロセッサがノードをアクティブ化すると、プラグイン フレームワークは、com.bea.wlpi.server.plugin.PluginEvent オブジェクトの activate() メソッドを呼び出す。activate() メソッドは、com.bea.wlpi.server.plugin.EventContext の activateEvent() メソッドを呼び出すことにより、イベント ウォッチ テーブルにエントリを記録する。 activate() メソッドの実装方法の詳細については、4-79 ページの「PluginEvent インタフェース メソッド」の表を参照。EventContext の詳細については、4-106 ページの「イベント コンテキスト」を参照。
開始ノード	ユーザがテンプレート定義をアクティブに設定すると、プラグイン フレームワークは、com.bea.wlpi.server.plugin.PluginStart2 オブジェクトの activate() メソッドを呼び出す。これにより、setTrigger() メソッドが呼び出される。setTrigger() メソッドは、com.bea.wlpi.server.plugin.EventContext の postStartWatch() メソッドを呼び出すことにより、イベント ウォッチ テーブルにエントリを記録する。 setTrigger() メソッドの実装方法の詳細については、4-95 ページの「PluginStart2 インタフェース メソッド」の表を参照。EventContext の詳細については、4-106 ページの「イベント コンテキスト」を参照。

プラグイン イベント ハンドラに対するイベントの送信

プラグイン イベント ハンドラに対してイベントを送信するには、メッセージ送信側は、com.bea.wlpi.server.eventprocessor.EventData オブジェクトの WLPIPlugin 文字列プロパティをイベント ハンドラ名に設定する必要があります。

たとえば、次のコードは、イベント ハンドラ SamplePlugin の名前に対して WLPIPlugin プロパティを設定し、EventData コンストラクタにこの情報を渡すことにより、プロパティ マップを構築する方法を示しています。

```
Map props = new HashMap();
props.put("WLPIPlugin""SamplePlugin");
EventData eventData = new EventData(data,
  "text/x-appplication/sample",
  "Order", 0, null, null, props);
```

7　プラグインの管理

この章では、プラグインの管理方法について説明します。この章の内容は以下のとおりです。

- プラグインの表示
- プラグインのロード
- プラグインのコンフィグレーション
- プラグインのリストの更新
- Studio によるプラグインの管理

プラグインの表示

インストールされているプラグインを表示するには、次の表に示す `com.bea.wlpi.server.plugin.PluginManager` インタフェース メソッド を使用します。

表 7-1 インストールされているプラグインを表示するための PluginManager インタフェース メソッド

メソッド	説明
`public com.bea.wlpi.common.plugin.PluginInfo getPlugin(java.lang.String pluginName, java.util.Locale lc) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	指定されたプラグインに関するローカライズされた基本情報を取得する。 メソッド パラメータの定義は次のとおり。 ■ *pluginName* ―プラグイン名を示す `java.lang.String` オブジェクト。 ■ *lc* ―表示文字列をローカライズするロケールを示す `java.util.Locale` オブジェクト。 このメソッドは、基本プラグイン情報を示す `com.bea.wlpi.common.plugin.PluginInfo` オブジェクトを返す。詳細については、B-35 ページの「PluginInfo オブジェクト」を参照。
`public com.bea.wlpi.common.plugin.PluginInfo[] getPlugins(java.util.Locale lc) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException`	指定されたロケールのためにインストールされているプラグインのリストを取得する。 メソッド パラメータの定義は次のとおり。 *lc* ―表示文字列をローカライズするロケールを示す `java.util.Locale` オブジェクト。 このメソッドは、インストールされているプラグインを示す `com.bea.wlpi.common.plugin.PluginInfo` オブジェクトの配列を返す。詳細については、B-35 ページの「PluginInfo オブジェクト」を参照。

たとえば、次のコードは、指定されたロケール `lc` のためにインストールされているプラグインのリストを取得し、`plugins[]` 配列にこのリストを保存します。この例では、`pm` は `PluginManager` EJB への `EJBObject` 参照を表します。

```
plugins[]=pm.getPlugins(lc);
```

getPlugin() メソッドおよび getPlugins() メソッドの詳細については、com.bea.wlpi.server.plugin.PluginManager Javadoc を参照してください。

プラグインのロード

インストールされているプラグインをロードするには、次の表に示す com.bea.wlpi.server.plugin.PluginManagerCfg インタフェース メソッドを使用します。

表 7-2 インストールされているプラグインをロードするための PluginManager インタフェース メソッド

メソッド	説明
public void loadPlugin(java.lang.String *pluginName*, com.bea.wlpi.commonVersionInfo *version*) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException	指定されたプラグインをロードし、初期化する。 Plug-in Manager は、そのプラグインの機能をすでにロードされているプラグインの機能と結合する。 メソッド パラメータの定義は次のとおり。 ■ *pluginName* ―プラグイン名を示す java.lang.String オブジェクト。 ■ *version* ― Plug-in Manager バージョンを示す com.bea.wlpi.common.VersionInfo オブジェクト。Plug-in Manager バージョンへのアクセス方法については、2-4 ページの「プラグイン フレームワーク バージョンの取得」を参照。

たとえば、次のコードは、指定された Plug-in Manager バージョン version のためのプラグイン MyPlugin をロードします。この例では、pmCfg は PluginManager EJB への EJBObject 参照を表します。

```
pmCfg.getPlugins(MyPlugin, version);
```

loadPlugin() メソッドの詳細については、com.bea.wlpi.server.plugin.PluginManagerCfg Javadoc を参照してください。

プラグインのコンフィグレーション

デフォルトでは、プラグインをコンフィグレーションするときに、プラグイン起動モードを指定できます。

たとえば、次の図は、WebLogic Intergration Studio でのデフォルトのプラグイン コンフィグレーション ダイアログを示しています。

図 7-1 Studio でのデフォルトのプラグイン コンフィグレーション ダイアログ

起動モードは次のいずれかの値に設定できます。

- Automatic ―システム起動時に自動的に起動される。これはデフォルト設定です。
- Manual ―ユーザが手動で起動する必要がある。
- Disabled ―使用禁止であり、起動できない。

プラグイン コンフィグレーション ダイアログへのアクセス方法については、『*WebLogic Integration Studio ユーザーズ ガイド*』の「ワークフロー リソースのコンフィグレーション」の「プラグインのコンフィグレーション」を参照してください。

必要な場合、プラグイン コンフィグレーション要件をカスタマイズできます。カスタマイズされたプラグイン コンフィグレーション要件は、前の図に示したテキスト「プラグインで定義されたデータが使用できません」の箇所に表示されます。

次の節では、プラグイン コンフィグレーション要件をカスタマイズする方法、およびプラグイン API を使用してコンフィグレーション値を編集する方法を説明します。この節の内容は以下のとおりです。

- プラグイン コンフィグレーション要件のカスタマイズ
- プラグイン コンフィグレーション値の設定
- プラグイン コンフィグレーション値の取得
- プラグイン コンフィグレーション値の削除

プラグイン コンフィグレーション要件のカスタマイズ

次の節では、プラグイン コンフィグレーション要件をカスタマイズするために必要な手順について説明します。この節の内容は以下のとおりです。

- PluginData インタフェースの実装
- PluginPanel クラスの定義
- ConfigurationInfo 値オブジェクトの定義

PluginData インタフェースの実装

XML フォーマットでプラグイン データを読み込み（解析し）、保存するには、4-11 ページの「PluginData インタフェースの実装」に説明されているように、プラグイン データ インタフェースを実装します。

次のコード リストは、プラグイン コンフィグレーションのための `PluginData` インタフェースを実装するクラスを定義する方法を示しています。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 7-1 PluginData インタフェースの実装ープラグイン コンフィグレーション

```
package com.bea.wlpi.test.plugin;

import java.io.IOException;
import com.bea.wlpi.common.plugin.PluginData;
import com.bea.wlpi.common.XMLWriter;
import java.util.List;
import java.util.Map;
import org.xml.sax.*;

public class ConfigData implements PluginData {
    private static final String YESORNO_TAG = "yesorno";

    private String yesOrNo;
```

```
    private transient String lastValue;

    public ConfigData() {
        this.yesOrNo = TestPluginConstants.CONFIG_NO;
    }

    public ConfigData(String yesOrNo) {
        this.yesOrNo = yesOrNo;
    }

    public void load(XMLReader parser) {
    }

    void setYesOrNo(String decision) {
        yesOrNo = decision;
    }

    String getYesOrNo() {
        return yesOrNo;
    }

    public void setDocumentLocator(Locator locator) {
    }

    public void startDocument()
        throws SAXException {
    }

    public void endDocument()
        throws SAXException {
    }

    public void startPrefixMapping(String prefix, String uri)
        throws SAXException {
    }

    public void endPrefixMapping(String prefix)
        throws SAXException {
    }

    public void startElement(String namespaceURI, String localName, String
qName, Attributes atts)
        throws SAXException {
        lastValue = null;
    }

    public void endElement(String namespaceURI, String localName, String name)
        throws SAXException {
        if (name.equals(YESORNO_TAG))
```

```
            yesOrNo = lastValue;
    }

    public void characters(char[] ch, int start, int length)


        throws SAXException {
        String value = new String(ch, start, length);

        if (lastValue == null)
            lastValue = value;
        else
            lastValue = lastValue + value;
    }

    public void ignorableWhitespace(char[] ch, int start, int length)
        throws SAXException {
    }

    public void processingInstruction(String target, String data)
        throws SAXException {
    }

    public void skippedEntity(String name)
        throws SAXException {
    }

    public void save(XMLWriter writer, int indent) throws IOException {
        writer.saveElement(indent, YESORNO_TAG, yesOrNo);
    }

    // TODO:
    public List getReferencedPublishables(Map publishables) {
        return null;
    }

    public String getPrintableData() {
        return toString();
    }

    public String toString() {
        return "ConfigData[yesOrNo=" + yesOrNo + ']';
    }

    public Object clone() {
        return new ConfigData(yesOrNo);
    }
}
```

PluginPanel クラスの定義

設計クライアント内でプラグイン GUI コンポーネントを表示するには、4-27 ページの「PluginPanel クラスの定義」に説明されているように、プラグイン パネル クラスを拡張するクラスを定義します。

次のコード リストは、プラグイン コンフィグレーションのために PluginPanel クラスを拡張するクラスを定義する方法を示しています。重要なコード行は、**太字**で示します。

注意： このクラスは、プラグイン サンプルには含まれていません。

コード リスト 7-2 PluginPanel クラスの定義ープラグイン コンフィグレーション

```
package com.bea.wlpi.test.plugin;

import java.awt.*;
import javax.swing.*;
import javax.swing.border.TitledBorder;
import javax.swing.border.EtchedBorder;
import java.util.List;
import java.util.Locale;
import com.bea.wlpi.common.plugin.PluginPanel;
import com.bea.wlpi.common.plugin.PluginPanelContext;
import com.bea.wlpi.client.studio.Studio;
import com.bea.wlpi.common.VariableInfo;

public class ConfigPanel extends PluginPanel {

    JPanel ButtonPanel = new JPanel();
    ButtonGroup YesNoButtonGroup = new ButtonGroup();
    JRadioButton YesButton = new JRadioButton();
    JRadioButton NoButton = new JRadioButton();
    TitledBorder titledBorder = new TitledBorder(new EtchedBorder());

    public ConfigPanel() {
        super(Locale.getDefault(), "halloween");
//        super(Locale.getDefault(), "pgconfig");
        //TODO: 文字列のためのリソース バンドルを作成
        setLayout(null);
        setBounds(12,12,420,300);
      ButtonPanel.setBorder(titledBorder);
        ButtonPanel.setLayout(null);
```

```
        add(ButtonPanel);
        ButtonPanel.setBounds(72,60,300,144);
        YesButton.setText("JavaHelp");
        YesButton.setSelected(true);
        YesNoButtonGroup.add(YesButton);
        ButtonPanel.add(YesButton);
        YesButton.setBounds(60,36,100,23);
        NoButton.setText("HTML Help");
        YesNoButtonGroup.add(NoButton);
        ButtonPanel.add(NoButton);
        NoButton.setBounds(60,60,100,23);
        titledBorder.setTitle("Online Help");
    }

    public void load() {
        ConfigData myData = (ConfigData)getData();
        if (myData != null) {
            if (myData.getYesOrNo().equals(TestPluginConstants.CONFIG_NO)) {
                NoButton.setSelected(true);
            } else {
                YesButton.setSelected(true);
            }
        }
    }

    public boolean validateAndSave() {
        ConfigData myData = (ConfigData)getData();
        if (myData != null) {
            myData.setYesOrNo(YesButton.isSelected()
                              ? TestPluginConstants.CONFIG_YES
                              : TestPluginConstants.CONFIG_NO);
        }
        return true;
    }
}
```

ConfigurationInfo 値オブジェクトの定義

プラグイン コンポーネント データをさらに詳細に定義するには、プラグイン コンフィグレーションのための com.bea.wlpi.common.plugin.ConfigurationInfo 値オブジェクトを定義します。この場合、基本プラグイン情報を定義するときに、*config* パラメータを使用して、com.bea.wlpi.common.plugin.PluginInfo 値オブジェクトに ConfigurationInfo オブジェクトを渡すことができます。

PluginInfo 値オブジェクトの定義時に *config* パラメータを null に設定した場合、プラグイン固有のコンフィグレーションは定義されません。この場合、7-4 ページの「Studio でのデフォルトのプラグイン コンフィグレーション ダイアログ」の図に示した Studio のプラグイン コンフィグレーション ダイアログが表示されます。PluginInfo 値オブジェクトの定義方法については、B-35 ページの「PluginInfo オブジェクト」を参照してください。

たとえば、次のコード リストは、ConfigurationInfo 値オブジェクトを定義する方法を示しています。

```
ci = new ConfigurationInfo(TestPluginConstants.PLUGIN_NAME, 12,
     "test plugin configuration",
     TestPluginConstants.CONFIG_CLASSES);
```

CONFIG_CLASSES フィールド要素値は、TestPluginConstants.java クラス ファイル内で定義されており、次のようにクラスを定義します。

```
final static String CONFIG_DATA =
    "com.bea.wlpi.test.plugin.ConfigData";

final static String CONFIG_PANEL =
    "com.bea.wlpi.test.plugin.ConfigPanel";
final static String[] CONFIG_CLASSES = {
    CONFIG_DATA, CONFIG_PANEL
};
```

ConfigurationInfo の詳細については、B-16 ページの「ConfigurationInfo オブジェクト」を参照してください。

プラグイン コンフィグレーション値の設定

プラグイン コンフィグレーション値を設定するには、次の表に定義された com.bea.wlpi.server.plugin.PluginManagerCfg インタフェース メソッドを使用します。

表 7-3 コンフィグレーション値を設定するための PluginManagerCfg インタフェース メソッド

メソッド	説明
public void setPluginConfiguration(java.lang.String *pluginName*, com.bea.wlpi.common.VersionInfo *version*, int *startMode*, java.lang.String *config*) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException	プラグイン コンフィグレーション情報を設定する。 メソッド パラメータの定義は次のとおり。 ■ *pluginName* ―プラグイン名を示す java.lang.String オブジェクト。 ■ *version* ―*Plug-in Manager バージョンを示す* com.bea.wlpi.common.VersionInfo オブジェクト。Plug-in Manager バージョンへのアクセス方法については、2-4 ページの「プラグイン フレームワーク バージョンの取得」を参照。 ■ *startMode* ― Plug-in Manager がプラグインを起動するときに指定する整数値。この値は、次の com.bea.wlpi.common.plugin.PluginConstants 値のいずれかに設定できる。 MODE_AUTOMATIC ―プラグインはシステム起動時に自動的に起動される。 MODE_DISABLED ―プラグインは使用禁止であり、起動できない。 MODE_MANUAL ―プラグインはユーザが手動で起動する必要がある。 ■ *config* ― XML ドキュメントとしてプラグイン コンフィグレーション情報を示す java.lang.String オブジェクト。

たとえば、次のコードは、pconfig.xml ファイルを使用して、MyPlugin プラグイン、指定された Plug-in Manager バージョン version、および MODE_AUTOMATIC 起動モードに関するプラグイン コンフィグレーション情報を設定します。この例では、pmCfg は PluginManagerCfg EJB への EJBObject 参照を表します。

```
pmCfg.setPluginConfiguration(MyPlugin, version, MODE_AUTOMATIC,
  pconfig.xml);
```

setPluginConfiguration() メソッドの詳細については、com.bea.wlpi.server.plugin.PluginManagerCfg Javadoc を参照してください。

プラグイン コンフィグレーション値の取得

プラグイン コンフィグレーション値を取得するには、次の表に示す com.bea.wlpi.server.plugin.PluginManager インタフェース メソッドを使用します。

表 7-4 コンフィグレーション値を取得するための PluginManager インタフェース メソッド

メソッド	説明
public com.bea.wlpi.common.plugin.ConfigurationData getPluginConfiguration(java.lang.String *pluginName*, com.bea.wlpi.common.VersionInfo *version*) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException	プラグイン コンフィグレーション情報を取得する。 メソッド パラメータの定義は次のとおり。 ■ *pluginName* ―プラグイン名を示す java.lang.String オブジェクト。 ■ *version* ―Plug-in Manager バージョンを示す com.bea.wlpi.common.VersionInfo オブジェクト。Plug-in Manager バージョンへのアクセス方法については、2-4 ページの「プラグイン フレームワーク バージョンの取得」を参照。 このメソッドは、コンフィグレーション情報を示す com.bea.wlpi.common.plugin.ConfigurationData オブジェクトを返す。特定のコンフィグレーション情報にアクセスするために使用できるメソッドの詳細については、B-14 ページの「ConfigurationData オブジェクト」を参照。

たとえば、次のコードは、MyPlugin と、指定された Plug-in Manager バージョン version に関するプラグイン コンフィグレーション情報を取得します。この例では、pm は PluginManager EJB への EJBObject 参照を表します。.

```
configData=pm.setPluginConfiguration(MyPlugin, version);
```

getPluginConfiguration() メソッドの詳細については、com.bea.wlpi.server.plugin.PluginManager Javadoc を参照してください。

プラグイン コンフィグレーション値の削除

プラグインのコンフィグレーションは、不要になった時点で削除できます。コンフィグレーションを削除する場合、プラグイン自体ではなく、登録されているコンフィグレーションのみを削除します。

プラグイン コンフィグレーション値を削除するには、次の表に示す com.bea.wlpi.server.plugin.PluginManagerCfg インタフェース メソッドを使用します。

表 7-5 コンフィグレーション値を削除するための PluginManagerCfg インタフェース メソッド

メソッド	説明
public void deletePluginConfiguration(java.lang.String *pluginName*, com.bea.wlpi.common.VersionInfo *version*) throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException	プラグイン コンフィグレーション情報を削除する。 メソッド パラメータの定義は次のとおり。 ■ *pluginName* ―プラグイン名を示す java.lang.String オブジェクト。 ■ *version* ― Plug-in Manager バージョンを示す com.bea.wlpi.common.VersionInfo オブジェクト。Plug-in Manager バージョンへのアクセス方法については、2-4 ページの「プラグイン フレームワーク バージョンの取得」 を参照。

たとえば、次のコードは、MyPlugin と、指定された Plug-in Manager バージョン version に関するプラグイン コンフィグレーション値を削除します。この例では、pmCfg は PluginManagerCfg EJB への EJBObject 参照を表します。

```
pmCfg.deletePluginConfiguration(MyPlugin, version);
```

deletePluginConfiguration() メソッドの詳細については、com.bea.wlpi.server.plugin.PluginManagerCfg Javadoc を参照してください。

プラグインのリストの更新

プラグインのリストを更新するには、次の表に示す com.bea.wlpi.server.plugin.PluginManagerCfg インタフェース メソッドを使用します。

表 7-6 プラグインのリストを更新するための PluginManagerCfg インタフェース メソッド

メソッド	説明
public void refresh() throws java.rmi.RemoteException, com.bea.wlpi.common.WorkflowException	キャッシュ内のプラグイン情報を更新する。 **注意：** このメソッドを実行するには大量のリソースが必要なため、その使用を制限する必要があります。 このメソッドは、ロードされているすべてのプラグインを Plug-in Manager に再照会し、その内部プラグイン機能キャッシュを再構築する。プラグイン機能が動的にコンフィグレーションされている場合、プラグインはこのメソッドを呼び出すことができる。

たとえば、次のコードは、ロードされているすべてのプラグインを更新します。この例では、pmCfg は PluginManager EJB への EJBObject 参照を表します。

```
pmCfg.refresh();
```

refresh() メソッドの詳細については、com.bea.wlpi.server.plugin.PluginManagerCfg Javadoc を参照してください。

Studio によるプラグインの管理

Studio 設計クライアント インタフェース内から、プラグインの表示、ロード、コンフィグレーションを行えます。詳細については、『*WebLogic Integration Studio ユーザーズ ガイド*』の「ワークフロー リソースのコンフィグレーション」を参照してください。

8 プラグイン オンライン ヘルプの定義

プラグイン オンライン ヘルプを定義する手順は、次のとおりです。

1. プラグイン オンライン ヘルプ（JavaHelp または HTML）ファイルを定義します。

   例として、
   SAMPLES_HOME/integration/samples/bpm_api/htmlhelp/Sample ディレクトリにある HTML ファイルを参照してください。

2. B-27 ページの「HelpSetInfo オブジェクト」に定義されているコンストラクタを使用して、com.bea.wlpi.common.plugin.HelpSetInfo 値オブジェクトを定義します。

   helpType コンストラクタ パラメータを使用して、JavaHelp ヘルプ セットまたは HTML ヘルプ セットのどちらを定義するか指定し、どちらのヘルプ セットを定義するかに基づいて、*helpNames* コンストラクタ パラメータを使用して、それぞれ次の情報を渡す必要があります。

   - JavaHelp ヘルプ セット（.hs）ファイル名と JavaHelp ヘルプ キー
   - メイン インデックス ページまたは目次の HTML ヘルプ ファイル ルート ディレクトリと HTML ファイル名

   詳細については、B-27 ページの「HelpSetInfo オブジェクト」を参照してください。

3. B-35 ページの「PluginInfo オブジェクト」に定義されているコンストラクタを使用して、com.bea.wlpi.common.plugin.PluginInfo 値オブジェクトを定義します。*helpSet* コンストラクタ パラメータ値として、前の手順で定義した HelpSetInfo 値オブジェクトの名前を渡す必要があります。

プラグイン サンプルから抜粋した次のコード リストは、プラグイン HTML オンライン ヘルプを定義する方法を示しています。このコードでは、createPluginInfo() メソッドを定義します。このメソッドは、まず HelpSetInfo 値オブジェクトを定義してから、この HelpSetInfo オブジェクト

を渡して PluginInfo オブジェクトを定義します。この抜粋は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリの SamplePluginBean.java ファイルから取り出したものです。重要なコード行は、**太字**で示します。

コード リスト 8-1 プラグイン HTML オンライン ヘルプの定義

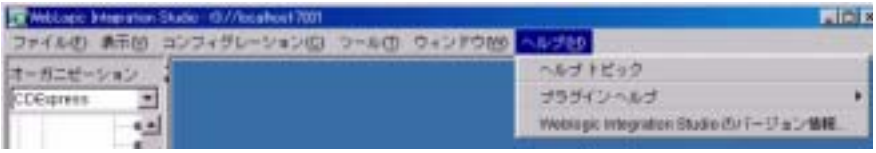
```
private PluginInfo createPluginInfo(Locale lc) {
   HelpSetInfo helpSet;
   PluginInfo pi;
   SampleBundle bundle = new SampleBundle(lc);
   String name = bundle.getString("pluginName");
   String desc = bundle.getString("pluginDesc");
   String helpName = bundle.getString("helpName");
   String helpDesc = bundle.getString("helpDesc");
   helpSet = new HelpSetInfo(
         SamplePluginConstants.PLUGIN_NAME, helpName, helpDesc,
         new String[]{ "htmlhelp/Sample", "index" },
         HelpSetInfo.HELP_HTML);
   pi = new PluginInfo(SamplePluginConstants.PLUGIN_NAME, name,
      desc, lc, SamplePluginConstants.VENDOR_NAME,
      SamplePluginConstants.VENDOR_URL,
      SamplePluginConstants.PLUGIN_VERSION,
      SamplePluginConstants.PLUGIN_FRAMEWORK_VERSION,
      null, null, helpSet);
   return pi;
}
```

この例について説明します。

- HTML ファイルは、他のサポート ファイルとともに、WAR ファイル（たとえば sampleplugin.war）にパッケージ化する必要があります。WAR ファイルは、com.bea.wlpi.SamplePlugin という Name 属性値を持つ WebAppComponent としてデプロイする必要があります。その他のデプロイメント事項については、9-8 ページの「コンフィグレーション ファイルの更新」を参照してください。
- プラグイン フレームワークは、*pluginName* の値に関連する htmlhelp/Sample パスを使用して、ヘルプ セットの URL を作成します。例：
  http://localhost:7001/com.bea.wlpi.SamplePlugin/htmlhelp/Sample/index.htm
- デフォルト ヘルプ ページは、index.htm です。

次の図に、サンプル プラグイン HTML ヘルプ セットにアクセスできる WebLogic Integration Studio の [ヘルプ] メニューを示します。

図 8-1 プラグイン ヘルプ セット



次のコード リストは、プラグイン JavaHelp ヘルプ セットのための `HelpSetInfo` 値オブジェクトを定義する方法を示しています。

コード リスト 8-2 プラグイン JavaHelp オンライン ヘルプの定義

```
javaHelpSet = new HelpSetInfo(SamplePluginConstants.PLUGIN_NAME,
        "Sample Plugin JavaHelp", "Plugin-provided help set",
        new String[] {"javahelp/HolidayHistory", "hol_intro"},
        HelpSetInfo.HELP_JAVA_HELP);
```

この例について説明します。

- このプラグイン フレームワークは、*`pluginName`* の値に関連する `javahelp/HolidayHistory` パスを使用して、このプラグインにより提供されるヘルプ セットの URL を作成します。
  例 :
  `http://localhost:7001/com.bea.wlpi.SamplePlugin/javahelp/HolidayHistory.hs`
- デフォルト ヘルプ トピックは、`hol_intro` です。

プラグイン オンライン ヘルプのデプロイメントの詳細については、9-1 ページの「プラグインのデプロイメント」 を参照してください。

9　プラグインのデプロイメント

プラグインはステートレス セッション EJB です。他の EJB と同じようにデプロイされます。この章では、プラグインをデプロイする方法について説明します。この章の内容は以下のとおりです。

- プラグイン デプロイメント記述子ファイルの定義
- プラグインのパッケージ化
- コンフィグレーション ファイルの更新

注意： プラグイン サンプルがデプロイされていますので、ご利用ください。プラグイン サンプル JAR、WAR、デプロイメント記述子ファイルは、インストール時に適切なディレクトリにコピーされます。

プラグイン デプロイメント記述子ファイルの定義

プラグインをデプロイするには、次の節で説明するように、EJB とオンライン ヘルプのデプロイメント プロパティを定義するプラグイン デプロイメント記述子ファイルを定義する必要があります。

プラグイン EJB デプロイメント記述子ファイルの定義

次の表に、プラグイン EJB のデプロイメントを定義するために必要なデプロイメント記述子ファイルを示します。

表 9-1 プラグイン EJB デプロイメント記述子ファイル

定義するファイル	指定内容
ejb-jar.xml	基本 EJB 構造、内部依存関係、およびアプリケーション アセンブリ情報
weblogic-ejb-jar.xml	WebLogic Server キャッシング、クラスタ化、およびパフォーマンスの情報、および WebLogic Server リソース マッピング（セキュリティ、JDBC プール、JMS 接続ファクトリなど、デプロイされる EJB リソースを含む）
weblogic-cmp-rdbms-jar.xml	WebLogic Server コンテナ管理による永続性サービス

EJB デプロイメント記述子ファイルの詳細については、次の URL にある『*WebLogic エンタープライズ Java Beans プログラマーズガイド*』の「WebLogic Server への EJB のデプロイ」を参照してください。

http://edocs.beasys.co.jp/e-docs/wls/docs70/ejb/deploy.html

プラグイン サンプルから抜粋した次のコード リストは、ejb-jar.xml および weblogic-ejb-jar.xml デプロイメント記述子ファイルを定義する方法を示しています。

注意： プラグインは、コンテナ管理によるトランザクション境界設定をサポートしている必要があります。したがって、プラグイン通知リスナ メソッドの trans-attr 要素は、Required、Supports、または Mandatory のいずれかの値である必要があります。

コード リスト 9-1 プラグイン サンプル ejb-jar.xml EJB デプロイメント記述子

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans
2.0//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">

<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>SamplePlugin</ejb-name>
      <home>com.bea.wlpi.server.plugin.PluginHome</home>
```

```
      <remote>com.bea.wlpi.server.plugin.Plugin</remote>
      <ejb-class>com.bea.wlpi.tour.po.plugin.SamplePluginBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-ref>
        <ejb-ref-name>ejb/PluginManagerCfg</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home>com.bea.wlpi.server.plugin.PluginManagerCfgHome</home>
        <remote>com.bea.wlpi.server.plugin.PluginManagerCfg</remote>
        <ejb-link>PluginManagerCfg</ejb-link>
      </ejb-ref>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>SamplePlugin</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

コード リスト 9-2 プラグイン サンプル weblogic-ejb-jar.xml EJB デプロイメント記述子

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-ejb-jar PUBLIC "-//BEA Systems, Inc.//DTD WebLogic 6.0.0
EJB//EN" "http://www.bea.com/servers/wls600/dtd/weblogic-ejb-jar.dtd">

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>SamplePlugin</ejb-name>
    <stateless-session-descriptor>
      <pool>
        <max-beans-in-free-pool>100</max-beans-in-free-pool>
        <initial-beans-in-free-pool>0</initial-beans-in-free-pool>
      </pool>
      <stateless-clustering>
        <stateless-bean-is-clusterable>True</stateless-bean-is-clusterable>

<stateless-bean-methods-are-idempotent>True</stateless-bean-methods-are-idempot
```

```
ent>
      </stateless-clustering>
    </stateless-session-descriptor>
    <reference-descriptor>
      <ejb-reference-description>
        <ejb-ref-name>ejb/PluginManagerCfg</ejb-ref-name>
        <jndi-name>com.bea.wlpi.PluginManagerCfg</jndi-name>
      </ejb-reference-description>
    </reference-descriptor>
    <jndi-name>com.bea.wlpi.tour.po.plugin.SamplePlugin</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

プラグイン オンライン ヘルプ デプロイメント記述子ファイルの定義

次の表に、プラグイン オンライン ヘルプのデプロイメントを定義するために必要なデプロイメント記述子ファイルを示します。

表 9-2 プラグイン オンライン ヘルプ デプロイメント記述子ファイル

定義するファイル	指定内容
`web.xml`	Web アプリケーション コンフィグレーション情報
`weblogic.xml`	`web.xml` ファイル内の指名されたリソースと WebLogic Server の外部にあるリソースに関するリソース マッピング情報、および JSP と HTTP のセッション属性

オンライン ヘルプ（Web アプリケーション）デプロイメント記述子ファイルの詳細については、次の URL にある「*Web アプリケーションのアセンブルとコンフィグレーション*」の「Web アプリケーションのデプロイメント記述子の記述」を参照してください。

http://edocs.beasys.co.jp/e-docs/wls/docs70/webapp/webappdeployment.html

プラグイン サンプルから抜粋した次のコード リストは、`web.xml` および `weblogic.xml` デプロイメント記述子ファイルを定義する方法を示しています。

コード リスト 9-3 プラグイン サンプル オンライン ヘルプ web.xml デプロイメント記述子

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    <display-name>Sample Plugin Online Help</display-name>
    <description>
        This Web Application serves up HTML Help for the
        WebLogic Process Integrator Sample Plugin.
    </description>
    <welcome-file-list>
    <welcome-file>
        com/bea/wlpi/tour/po/plugin/htmlhelp/index.htm
    </welcome-file>
    </welcome-file-list>
</web-app>
```

コード リスト 9-4 プラグイン サンプル オンライン ヘルプ weblogic.xml デプロイメント記述子

```
<!DOCTYPE weblogic-web-app PUBLIC "-//BEA Systems, Inc.//DTD Web
Application 6.0//EN"
"http://www.bea.com/servers/wls600/dtd/weblogic-web-jar.dtd">
<weblogic-web-app>
    <description>
        This Web Application serves up HTML Help for the
        WebLogic Process Integrator Sample Plugin.
    </description>
</weblogic-web-app>
```

プラグインのパッケージ化

WebLogic Server に対してデプロイされる JAR ファイルにプラグインをパッケージ化する手順は、次のとおりです。

1. ビルド ディレクトリを作成し、javac を使用して、このディレクトリにソース ファイルをコンパイルします。

プラグイン サンプルの場合、ソース ファイルは *SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリにあり、*SAMPLES_HOME*/integration/samples/bpm_api/plugin ディレクトリにコンパイルされます。

2. 前の節で作成したデプロイメント記述子（ejb-jar.xml、weblogic-ejb-jar.xml）を、ビルド ディレクトリの META-INF サブディレクトリにコピーします。

   たとえば、デプロイメント記述子ファイルを *SAMPLES_HOME*/integration/samples/bpm_api/plugin/META-INF ディレクトリにコピーします。

3. コンパイルされたソース ファイルとデプロイメント記述子の入ったビルド ディレクトリの JAR ファイルを作成します。

   プラグイン サンプルの場合、結果の JAR ファイルは、*SAMPLES_HOME*/integration/samples/lib/sampleplugin-ejb.jar として格納されます。

4. JAR ファイルで weblogic.ejbc を実行し、WebLogic Server コンテナ クラスを生成します。

   プラグイン サンプルの場合、このユーティリティの出力は、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/ejbcgen ディレクトリに格納されます。

5. 必要であれば、プラグイン オンライン ヘルプとデプロイメント記述子ファイルの入った WAR ファイルを作成します。

   プラグイン サンプルの場合、プラグイン オンライン ヘルプ ファイルは、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/htmlhelp ディレクトリに格納されます。結果の WAR ファイルは、*SAMPLES_HOME*/integration/samples/lib/sampleplugin.war として格納されます。

プラグイン サンプルから抜粋した次のコード リストは、プラグインをパッケージ化するためのビルド スクリプト build.cmd を定義する方法を示しています。このファイルは、*SAMPLES_HOME*/integration/samples/bpm_api/plugin ディレクトリにあります。重要なコード行は、**太字**で示します。

注意： build.cmd スクリプトを実行する前に、setEnv.cmd スクリプトを更新、実行して、環境を設定しておく必要があります。このスクリプトは、*SAMPLES_HOME*/integration/samples/bpm_api/plugin ディレクトリにあります。

コード リスト 9-5 プラグイン サンプル ビルド スクリプト

```
@rem Copyright (c) 2001 BEA Systems, Inc. All rights reserved.
@rem build.cmd ―デプロイ可能な jar ファイル sampleplugin をコンパイルおよび作成する
@echo off


@rem クラスをコンパイルする
setlocal
set JAVAC_ARGS=-d . -g -deprecation
echo Compiling Sample Plugin classes
"%JAVA_HOME%\bin\javac" %JAVAC_ARGS% source\*.java
endlocal

@rem jar を作成する
echo Building Sample Plugin EJB jar for bean classes
erase /f _sampleplugin-ejb.jar 2> nul 1> nul
@copy interfaces.jar _sampleplugin-ejb.jar 2> nul 1> nul
@copy source\Sample.gif com\bea\wlpi\tour\po\plugin\Sample.gif 2> nul 1> nul
@copy source\SamplePlugin.properties
com\bea\wlpi\tour\po\plugin\SamplePlugin.properties 2> nul 1> nul
@rem 標準およびベンダ固有の XML デプロイメント記述子を追加する
"%JAVA_HOME%\bin\jar" -uf _sampleplugin-ejb.jar META-INF\ejb-jar.xml
META-INF\weblogic-ejb-jar.xml
rem Add the bean implementation classes, and helper classes.
"%JAVA_HOME%\bin\jar" -uf _sampleplugin-ejb.jar com
dir /b _sampleplugin-ejb.jar

echo Compiling EJB container classes

erase /f sampleplugin-ejb.jar 2> nul 1> nul
"%JAVA_HOME%\bin\java" -Dweblogic.ejb20.ejbc.debug=1 weblogic.ejbc -compiler
"%JAVA_HOME%\bin\javac" _sampleplugin-ejb.jar sampleplugin-ejb.jar
dir /b sampleplugin-ejb.jar
if not exist sampleplugin-ejb.jar echo *** ERROR: ejbc failed to create the
sampleplugin-ejb.jar file.


echo Building Sample Plugin WAR file for JavaHelp/HTML Help
"%JAVA_HOME%\bin\jar" -cf sampleplugin.war WEB-INF
```

```
"%JAVA_HOME%\bin\jar" -uf sampleplugin.war htmlhelp
dir /b sampleplugin.war
if not exist sampleplugin.war echo *** ERROR: failed to create the
sampleplugin.war file.

del _sampleplugin-ejb.jar
echo Done.
```

コンフィグレーション ファイルの更新

プラグインをデプロイするには、コンフィグレーション ファイル `config.xml` を更新し、関連付けられたデプロイメント記述子ファイルを WebLogic Integration アプリケーションの一部として指定する必要があります。

プラグイン EJB 記述子ファイルを指定するには、`<EJBComponent>` 要素を使用します。`DeploymentOrder` 属性を使用して、EJB JAR ファイルがデプロイされる順序を制御できます。一般に、プラグイン A がプラグイン B に依存している場合、プラグイン B を先にデプロイする必要があります。最終的には、BPM プラグイン フレームワークがプラグインのロード順序を制御します。たとえば、プラグイン フレームワークがプラグイン A をロードしようとしたときに、プラグイン A がプラグイン B に依存しており、それがまだロードされていない場合、プラグイン フレームワークはプラグイン B をロードします。

プラグイン オンライン ヘルプ ファイルを指定するには、`<WebAppComponent>` 要素を使用します。`com.bea.wlpi.common.plugin.HelpSetInfo` オブジェクトの `pluginName` パラメータの値に対して `Name` 属性を設定する必要があります。これは、プラグイン オンライン ヘルプを定義するときに設定されます。プラグイン オンライン ヘルプの定義方法の詳細については、8-1 ページの「プラグイン オンライン ヘルプの定義」 を参照してください。

サンプル ドメイン `config.xml` から抜粋した次のコード リストは、プラグイン サンプルをデプロイするために必要な情報を示しています。このファイルは、*`SAMPLES_HOME`*`/config/samples` ディレクトリにあります。重要なコード行は、**太字**で示します。

コード リスト 9-6 config.xml ファイルにあるプラグイン サンプル EJB のデプロイメント

```
      .
      .
      .
<Application Deployed="true" Name="WLI"
Path="E:\bea\weblogic600\samples\integration\samples\lib">
   <EJBComponent DeploymentOrder="0" Name="repository-ejb.jar"
      Targets="myserver" URI="repository-ejb.jar"/>
   <WebAppComponent Name="XTPlugin" Targets="myserver" URI="wlxtpi.war"/>
   <WebAppComponent Name="wlai" ServletReloadCheckSecs="1"
      Targets="myserver" URI="wlai.war"/>
   <EJBComponent DeploymentOrder="2" Name="wlpi-master-ejb.jar"
      Targets="myserver" URI="wlpi-master-ejb.jar"/>
   <EJBComponent DeploymentOrder="1" Name="wlpi-ejb.jar"
      Targets="myserver" URI="wlpi-ejb.jar"/>
   <EJBComponent DeploymentOrder="4" Name="wlc-wlpi-plugin.jar"
      Targets="myserver" URI="wlc-wlpi-plugin.jar"/>
   <EJBComponent DeploymentOrder="8" Name="wlai-admin-ejb"
      Targets="myserver" URI="wlai-admin-ejb.jar"/>
   <EJBComponent DeploymentOrder="5" Name="pobean.jar"
      Targets="myserver" URI="pobean.jar"/>
   <WebAppComponent Name="b2bconsole" ServletReloadCheckSecs="1"
      Targets="myserver" URI="b2bconsole.war"/>
   <EJBComponent DeploymentOrder="3" Name="wlpi-mdb-ejb.jar"
      Targets="myserver" URI="wlpi-mdb-ejb.jar"/>
   <EJBComponent DeploymentOrder="7" Name="wlai-ejb-server"
      Targets="myserver" URI="wlai-ejb-server.jar"/>
   <EJBComponent DeploymentOrder="6" Name="wlxtpi.jar"
      Targets="myserver" URI="wlxtpi.jar"/>
   <EJBComponent DeploymentOrder="9" Name="wlaiplugin-ejb.jar"
       Targets="myserver" URI="wlaiplugin-ejb.jar"/>
   <WebAppComponent Name="WLAIPlugin" Targets="myserver" URI="wlai-plugin.war"/>
   <EJBComponent DeploymentOrder="10" Name="sampleplugin-ejb.jar"
       Targets="myserver" URI="sampleplugin-ejb.jar"/>
   <WebAppComponent Name="com.bea.wlpi.SamplePlugin"
      Targets="myserver" URI="sampleplugin.war"/>
</Application>
      .
      .
      .
```

前述の例では、Plug-in Manager の入った `wlpi-master-ejb.jar` ファイルは、プラグイン サンプル ファイル `sampleplugin-ejb.jar` より前にデプロイされています。プラグイン サンプルは、依存関係を定義する Plug-in Manager を参照するので、Plug-in Manager ファイルより後にデプロイする必要があります。

`config.xml` ファイルの更新方法の詳細については、次の URL にある「*WebLogic Server コンフィグレーション リファレンス*」を参照してください。

`http://edocs.beasys.co.jp/e-docs/wls/docs70/config_xml/index.html`

10 BPM プラグイン サンプル

この章では、BPM プラグイン サンプルについて詳しく説明します。この章の内容は以下のとおりです。

- プラグイン サンプルの内容
- プラグイン サンプルの使い方

プラグイン サンプルの内容

BPM プラグイン サンプルは、一般的なプラグイン シナリオを表すプラグイン クラスのセットで構成されています。これは、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリにあるソフトウェアに入っています。このサンプルには、Plug-in Order Processing および Plug-in Order Fulfillment という 2 つのワークフロー テンプレートがあります。このマニュアルの各所で、プラグイン サンプルからの抜粋が使用されています。

注意： プラグイン サンプルは、『*WebLogic Integration BPM ユーザーズ ガイド*』の「Business Process Management とサンプル ワークフローの紹介」に詳しく説明されている Web ベースの発注シナリオにおおむね基づいています。

次の表で、1-15 ページの「プラグイン サンプル ワークフロー テンプレート」の図に示したプラグイン サンプルについて説明します。また、ワークフロー コンポーネントと、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリにある関連付けられたサンプル ソース ファイルをリストします。ワークフロー コンポーネント 1 ～ 3 は、Plug-in Order Processing ワークフロー テンプレートの一部です。ワークフロー コンポーネント 4 ～ 5 は、Plug-in Order Fulfillment ワークフロー テンプレートの一部です。

表 10-1 プラグインおよび関係するソース ファイルの説明

#	ワークフロー コンポーネント	説明	関係するソース ファイル
1	開始ノード	プラグイン イベント Start Order によりトリガされ、プラグイン メッセージ タイプ（セミコロン区切りのテキスト文字列）の受信時にワークフローを開始する。	■ `StartObject.java` － XML フォーマットでプラグイン データを読み込むための `PluginObject` インタフェースを実装する。 ■ `StartNodeData.java` －開始ノードに関係するデータを読み込み、保存するための `PluginData` インタフェースを実装する。 ■ `StartNodePanel.java` －設計クライアントでプラグイン GUI コンポーネントを表示するための `PluginTriggerPanel` クラスを定義する。 ■ `StartNode.java` －実行情報を指定するためのプラグイン実行時コンポーネント クラスを定義する。 ■ `OrderField.java` －プラグイン外部イベント Start Order と関連付けられたプラグイン データにアクセスするためのプラグイン フィールドを定義する。

表 10-1 プラグインおよび関係するソース ファイルの説明 （続き）

#	ワークフロー コンポーネント	説明	関係するソース ファイル
2	アクション－ Check Inventory	品目の在庫有無をチェックする。	■ `CheckInventoryActionObject.java` － XML フォーマットでプラグイン データを読み込むための `PluginObject` インタフェースを実装する。 ■ `CheckInventoryActionData.java` － XML フォーマットでプラグイン アクション データを読み込み、保存するための `PluginActionData` インタフェースを実装する。 ■ `CheckInventoryActionPanel.java` －設計クライアントでプラグイン GUI コンポーネントを表示するための `PluginTriggerPanel` インタフェースを定義する。 ■ `CheckInventoryAction.java` －実行情報を指定するための実行時コンポーネント クラスを定義する。

表 10-1 プラグインおよび関係するソース ファイルの説明 （続き）

#	ワークフロー コンポーネント	説明	関係するソース ファイル
3	イベント ノード－ Wait for Confirmation	プラグイン イベント [注文イベントを確認] によりトリガされ、プラグイン メッセージ タイプ（セミコロン区切りのテキスト文字列）の受領までワークフローを中断する。	■ `EventObject.java` － XML フォーマットでプラグイン データを読み込むための `PluginObject` インタフェースを実装する。 ■ `EventNodeData.java` －イベント ノードに関係するデータを読み込み、保存するための `PluginData` インタフェースを実装する。 ■ `EventNodePanel.java` －設計クライアントでプラグイン GUI コンポーネントを表示するための `PluginTriggerPanel` クラスを定義する。 ■ `EventNode.java` －実行情報を指定するためのプラグイン実行時コンポーネント クラスを定義する。 ■ `ConfirmField.java` －プラグイン外部イベント Confirm Order と関連付けられたプラグイン データにアクセスするためのプラグイン フィールドを定義する。
4	関数－ CalculateTotalPrice	品目番号、数量、出荷先の住所に基づいて受注合計額を計算する。	■ `CalculateTotalPriceFunction.java` －品目 ID、数量、出荷先を使用して受注合計額を計算するための実行情報を実装する。出荷先は売上税率を調べるために使用される。

表 10-1 プラグインおよび関係するソース ファイルの説明 （続き）

#	ワークフロー コンポーネント	説明	関係するソース ファイル
5	アクション－ Confirm Order Fulfillment	[注文イベントを確認] ノードをトリガするイベント メッセージを生成する。	■ SendConfirmObject.java － XML フォーマットでプラグイン データを読み込むための PluginObject インタフェースを実装する。 ■ SendConfirmActionData.java － XML フォーマットでプラグイン アクション データを読み込み、保存するための PluginActionData インタフェースを実装する。 ■ SendConfirmActionPanel.java － 設計クライアントでプラグイン GUI コンポーネントを表示するための PluginTriggerPanel インタフェースを実装する。 ■ SendConfirmAction.java －実行情報を定義するための PluginAction インタフェースを実装する。

次の表に、*SAMPLES_HOME*/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin ディレクトリにあるその他のソース ファイルを示します。

表 10-2 その他のプラグイン サンプル ソース ファイル

ファイル	説明
Sample.gif	WebLogic Integration Studio インタフェース ビューが有効な場合に、カスタマイズされたプラグイン プロパティの入っている開始ノードとイベント ノードの右上すみに表示されるカスタム アイコン。3-4 ページの「プラグイン ホーム インタフェース」に説明されているように、リモート プラグイン オブジェクト インタフェースの作成時にカスタム プラグイン アイコンを指定できる。

表 10-2 その他のプラグイン サンプル ソース ファイル （続き）

ファイル	説明
SampleBundle.java	ローカライズされたリソースのバンドル。
SamplePlugin.properties	リソース文字列。
SamplePluginBean.java	プラグイン セッション EJB。
SamplePluginConstants.java	プラグイン定数ファイル。
StartOrderDriver.java	受注処理をトリガするためのドライバ。

プラグイン サンプルの使い方

プラグイン サンプルはそのまま使用できます。プラグイン サンプル JAR ファイル、WAR ファイル、およびデプロイメント記述子ファイルは、WebLogic Integration のインストール時に適切なディレクトリにデプロイされます。

次の節では、サンプル プラグインをインポートし、実行する方法について説明します。

プラグイン サンプルのインポート

プラグイン サンプルをインポートするには、Studio インポート パッケージ ツールを使用します。Studio インポート パッケージ ツールにより、JAR ファイルの形式でワークフロー パッケージをインポートできます。このパッケージには、テンプレート、テンプレート定義、イベント キー、ビジネス オペレーションなどのオブジェクトを任意の数だけ入れることができます。

プラグイン サンプル パッケージをインポートする手順は次のとおりです。

1. 『*WebLogic Integration の起動、停止およびカスタマイズ*』の「はじめに」の「サンプル ドメインのコンフィグレーションと起動」に説明されているように、サンプル ドメインを使用して BEA WebLogic Integration を起動します。
2. Studio 設計クライアントを呼び出します。

詳細については、『*WebLogic Integration Studio ユーザーズ ガイド*』を参照してください。

3. [ツール | パッケージをインポート] を選択します。

   Import ウィザードの [ファイルを選択] ダイアログ ボックスが表示されます。

4. [参照] をクリックし、*SAMPLES_HOME*/integration/samples/bpm_api/plugin ディレクトリに移動して、sample_plug_in.jar ファイルを開きます。

5. [次へ] をクリックします。

   [インポートするコンポーネントを選択] ダイアログ ボックスが表示されます。

6. 対象オーガニゼーションを CDExpress に設定します。

   デフォルトで、インポート ファイル内のすべてのワークフロー オブジェクトが選択されます。

7. プラグイン サンプル ワークフローをアクティブ化するため、[インポート後にワークフローをアクティブ化] チェック ボックスを選択します。

   注意 : ワークフローは、実行前にアクティブ化しておく必要があります。あるいは、『*WebLogic Integration Studio ユーザーズ ガイド*』に説明されているように、ワークフローはインポート後にアクティブ化することもできます。

8. [インポート] をクリックして、パッケージ全体をインポートします。パッケージは、テンプレート、テンプレート定義、ビジネス オペレーション、およびイベント キーで構成されています。

   インポートされているオブジェクトの概略を示す [インポート概略を確認] ダイアログ ボックスが表示されます。

9. [閉じる] をクリックして、ダイアログ ボックスを閉じます。

   これで、インポートされたテンプレートとテンプレート定義がフォルダ ツリーに表示されます。

ワークフロー パッケージのインポートとエクスポートの詳細については、『*WebLogic Integration Studio ユーザーズ ガイド*』の「ワークフロー パッケージのインポートとエクスポート」を参照してください。

プラグイン サンプルの実行

プラグイン サンプルは、プラグイン定義されたイベントを使用して、Plug-in Order Processing ワークフロー テンプレートをトリガすることにより実行できます。汎用ドライバ ファイル `StartOrderDriver.java` が *`SAMPLES_HOME`*`/integration/samples/bpm_api/plugin/src/com/bea/wlpi/tour/po/plugin` ディレクトリに用意されているので、ご利用ください。このドライバは、Plugin Order Processing ワークフローに顧客受注情報を提供するプラグイン定義されたイベントを生成します。この情報は、Plugin Order Processing ワークフロー変数に格納され、受注を処理するために使用されます。

注意： Worklist クライアント アプリケーションは、このリリースの WebLogic Integration から非推奨になっております。これに代わる機能の詳細については、『*WebLogic Integration リリース ノート*』を参照してください。

サンプルを実行する手順は次のとおりです。

1. BEA WebLogic Integration がまだ起動されていない場合は、起動します。
2. BPM Worklist ツールを起動し、次のログインとパスワードを使用してログオンします。
   - ログイン― `admin`
   - パスワード― `security`

   admin ユーザは、サンプル ワークフローで使用されるすべてのロールのメンバーとして定義されています。詳細については、『*WebLogic Integration BPM ユーザーズ ガイド*』の「サンプル ワークフローの実行とモニタ」の「Worklist アプリケーションにログオンする」を参照してください。

   プラグイン サンプルと会話するには、Worklist ツールを使用する必要があります。
3. Worklist ツールのドロップダウン リストからオーガニゼーションとして CDExpress を選択します。
4. `StartOrderDriver` を実行します。
   a. Windows では、*`WL_HOME`*`\samples\integration\samples\bpm_api\plugin\StartOrder.cmd` スクリプトを実行します。

b. UNIX では、次のコマンドを実行して、環境と CLASSPATH 変数を設定し、StartOrderDriver スクリプトを実行します。

```
$WLI_HOME/setenv.sh
CLASSPATH=
$WL_HOME/lib/weblogic.jar:$SAMPLES_HOME/integration/samples/
lib/wlpi-ejb.jar:
   $SAMPLES_HOME/integration/samples/lib/sampleplugin-ejb.jar
$JAVA_HOME/bin/java -classpath "$CLASSPATH"
    com.bea.wlpi.tour.po.plugin.StartOrderDriver
    t3://localhost:7001 joe password
```

admin ユーザのタスク リストに [Check Customer Credit] タスクが表示されます。

5. [Check Customer Credit] タスクをダブルクリックし、このタスクを実行します。

   [Credit Check] メッセージ ボックスが表示されます

6. [Yes] をクリックし、タスクを完了としてマークします。

   次のタスクである Start Order Fulfillment が Order Fulfillment サブワークフローを自動的に開始します。

7. [Shipping] タブを選択し、[Ship Order] タスクをダブルクリックして、このタスクを実行します。

   [Ship Order] タスクが実行されます。

8. [Accounting] タブを選択し、[Generate Invoice] タスクをダブルクリックし、このタスクを実行します。

   以下のタスクが実行されます。

   - Generate Invoice タスクが実行され、完了としてマークされます。
   - Calculate Total Price ビジネス演算により、受注合計額が計算され、Order Processing ワークフローにその値が返されます。
   - Confirm Order Fulfillment タスクが自動的に実行されます。

   この時点で、ビジネス プロセスは完了です。

A プラグイン コンポーネント定義のロードマップ

次の表に、各タイプのプラグイン コンポーネントを定義するために必要な手順の概略を示します。プラグイン コンポーネントの定義方法の詳細については、4-1 ページの「プラグイン コンポーネントの定義」 を参照してください。

表 A-1 プラグイン コンポーネント定義のロードマップ

定義するプラグイン コンポーネント	実行する手順
アクション	1. 4-22 ページの「PluginActionData インタフェースの実装」 に説明されているように、XML フォーマットでプラグイン アクション データを読み込み、保存するための `PluginActionData` インタフェースを実装する。 2. 4-43 ページの「PluginActionPanel クラスの定義」 に説明されているように、設計クライアントでプラグイン アクション GUI コンポーネントを表示するための `PluginActionPanel` クラスを定義する。 3. 4-65 ページの「アクションのための実行時コンポーネント クラスの定義」 に説明されているように、プラグイン アクション実行時コンポーネント クラスを定義する。 4. 2-7 ページの「プラグイン値オブジェクトの定義」 に説明されているように、`ActionCategoryInfo`、`ActionInfo`、`CategoryInfo` 値オブジェクトを定義する。

表 A-1 プラグイン コンポーネント定義のロードマップ （続き）

定義するプラグイン コンポーネント	実行する手順
完了ノード	1. 4-11 ページの「PluginData インタフェースの実装」に説明されているように、XML フォーマットでプラグイン データを読み込み、保存するための PluginData インタフェースを実装する。 2. 4-27 ページの「PluginPanel クラスの定義」に説明されているように、設計クライアントでプラグイン GUI コンポーネントを表示するための PluginPanel クラスを定義する。 3. 4-76 ページの「完了ノードのための実行時コンポーネント クラスの定義」に説明されているように、プラグイン実行時コンポーネント クラスを定義する。 4. 2-7 ページの「プラグイン値オブジェクトの定義」に説明されているように、DoneInfo 値オブジェクトを定義する。
イベント ノード	1. 4-11 ページの「PluginData インタフェースの実装」に説明されているように、XML フォーマットでプラグイン データを読み込み、保存するための PluginData インタフェースを実装する。 2. 4-49 ページの「PluginTriggerPanel クラスの定義」に説明されているように、設計クライアントでプラグイン GUI コンポーネントを表示するための PluginTriggerPanel クラスを定義する。 3. 4-78 ページの「イベント ノードのための実行時コンポーネント クラスの定義」に説明されているように、プラグイン実行時コンポーネント クラスを定義する。 4. 4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」に説明されているように、イベント ノードで受信される外部イベントと関連付けられたプラグイン データにアクセスするためのプラグイン フィールドを定義する。 text/xml 以外のコンテンツ タイプを持ち、キー値をサポートするプラグイン イベント ノードと開始ノードについては、有効なイベント キー式を定義しておく必要がある。これにより、着信イベント データに対してそのキー値を評価するためのプラグイン フィールド実装を提供できる。 5. 2-7 ページの「プラグイン値オブジェクトの定義」に説明されているように、EventInfo 値オブジェクトを定義する。

表 A-1 プラグイン コンポーネント定義のロードマップ （続き）

定義するプラグイン コンポーネント	実行する手順
関数	1. 4-83 ページの「関数のための実行時コンポーネント クラスの定義」に説明されているように、プラグイン実行時コンポーネント クラスを定義する。 2. 2-7 ページの「プラグイン値オブジェクトの定義」に説明されているように、`FunctionInfo` 値オブジェクトを定義する。
メッセージ タイプ（フィールド）	1. 4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」に説明されているように、プラグイン実行時コンポーネント クラスを定義する。 2. 2-7 ページの「プラグイン値オブジェクトの定義」に説明されているように、`FieldInfo` 値オブジェクトを定義する。
開始ノード	1. 4-11 ページの「PluginData インタフェースの実装」に説明されているように、XML フォーマットでプラグイン データを読み込み、保存するための `PluginData` インタフェースを実装する。 2. 4-49 ページの「PluginTriggerPanel クラスの定義」に説明されているように、設計クライアントでプラグイン GUI コンポーネントを表示するための `PluginTriggerPanel` クラスを定義する。 3. 4-94 ページの「開始ノードのための実行時コンポーネント クラスの定義」に説明されているように、プラグイン実行時コンポーネント クラスを定義する。 4. 4-88 ページの「メッセージ タイプのための実行時コンポーネント クラスの定義」に説明されているように、開始ノードで受信される外部イベントと関連付けられたプラグイン データにアクセスするためのプラグイン フィールドを定義する。 `text/xml` 以外のコンテンツ タイプを持ち、キー値をサポートするプラグイン イベント ノードと開始ノードについては、有効なイベント キー式を定義しておく必要がある。これにより、着信イベント データに対してそのキー値を評価するためのプラグイン フィールド実装を提供できる。 5. 2-7 ページの「プラグイン値オブジェクトの定義」に説明されているように、`StartInfo` 値オブジェクトを定義する。

表 A-1 プラグイン コンポーネント定義のロードマップ （続き）

定義するプラグイン コンポーネント	実行する手順
ワークフロー テンプレート定義プロパティ	1. XML フォーマットでプラグイン データを読み込み、保存するための `PluginData` インタフェースを実装する。 2. 4-27 ページの 「PluginPanel クラスの定義」 に説明されているように、設計クライアントでプラグイン GUI コンポーネントを表示するための `PluginPanel` クラスを定義する。 3. 2-7 ページの 「プラグイン値オブジェクトの定義」 に説明されているように、`TemplateDefinitionPropertiesInfo` 値オブジェクトを定義する。
ワークフロー テンプレート プロパティ	1. XML フォーマットでプラグイン データを読み込み、保存するための `PluginData` インタフェースを実装する。 2. 4-27 ページの 「PluginPanel クラスの定義」 に説明されているように、設計クライアントでプラグイン GUI コンポーネントを表示するための `PluginPanel` クラスを定義する。 3. 2-7 ページの 「プラグイン値オブジェクトの定義」 に説明されているように、`TemplatePropertiesInfo` 値オブジェクトを定義する。
変数タイプ	1. 4-58 ページの 「PluginVariablePanel クラスの定義」 に説明されているように、ユーザがプラグイン変数タイプを編集できるようにプラグイン GUI コンポーネントを表示する `PluginVariablePanel` クラスを定義する。 2. 4-61 ページの 「PluginVariableRenderer クラスの定義」 に説明されているように、`javax.swing.JTable` のセルにプラグイン定義された変数タイプの値を表示するための PluginVariableRenderer クラスを定義する。 3. 2-7 ページの 「プラグイン値オブジェクトの定義」 に説明されているように、`VariableTypeInfo` 値オブジェクトを定義する。

B プラグイン値オブジェクトのまとめ

この付録では、BPM プラグイン値（すなわち *Info*）オブジェクトとそのメソッドについて説明します。この付録の内容は以下のとおりです。

- ActionCategoryInfo オブジェクト
- ActionInfo オブジェクト
- CategoryInfo オブジェクト
- ConfigurationData オブジェクト
- ConfigurationInfo オブジェクト
- DoneInfo オブジェクト
- EventHandlerInfo オブジェクト
- EventInfo オブジェクト
- FieldInfo オブジェクト
- FunctionInfo オブジェクト
- HelpSetInfo オブジェクト
- InfoObject オブジェクト
- PluginCapabilitiesInfo オブジェクト
- PluginDependency オブジェクト t
- PluginInfo オブジェクト
- StartInfo オブジェクト
- TemplateDefinitionPropertiesInfo オブジェクト
- TemplateNodeInfo オブジェクト

- TemplatePropertiesInfo オブジェクト
- VariableTypeInfo オブジェクト

値オブジェクト情報の定義方法とアクセス方法の詳細については、2-5 ページの「プラグイン値オブジェクトの使い方」 を参照してください。

ActionCategoryInfo オブジェクト

com.bea.wlpi.common.plugin.ActionCategoryInfo オブジェクトは、プラグイン アクションまたはアクション カテゴリ情報を管理します。

ActionCategoryInfo は、次のオブジェクトの抽象基本クラスです。

- com.bea.wlpi.common.plugin.ActionInfo
- com.bea.wlpi.common.plugin.CategoryInfo

ActionCategoryInfo クラスは、B-30 ページの 「InfoObject オブジェクト」 に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい ActionCategoryInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public ActionCategoryInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  int parentSystemID,
  java.lang.String[] classNames
)
```

次の表に、ActionCategoryInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-1 ActionCategoryInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	`public java.lang.String getPluginName()`
アクションまたはアクション カテゴリ ID	*ID*	`public int getID()`
アクションまたはアクション カテゴリのローカライズされた名前	*name*	`public java.lang.String getName()`
アクションまたはアクション カテゴリのローカライズされた説明	*description*	`public java.lang.String getDescription()`

表 B-1 ActionCategoryInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
アクション ツリーでアクション カテゴリを識別する親カテゴリの ID。 この ID は次のいずれかの値に設定できる。 ■ ID_EXCEPTION — Studio の Exception Handling Actions カテゴリ。 ■ ID_INTEGRATION — Studio の Integration Actions カテゴリ ■ ID_MISCELLANEOUS — Studio の Miscellaneous Actions カテゴリ。 ■ ID_NEW — Initial カテゴリ。 ■ ID_TASK — Studio の Task Actions カテゴリ。 ■ ID_WORKFLOW — Studio の Workflow Actions カテゴリ。 親システム ID に対するアクセス権を持たない新しいアクション カテゴリには、必ず ID_NEW を使用する。たとえば、次のカテゴリを追加する場合は、ID_NEW を使用する必要がある。 ■ ツリーのルートの下の新しいカテゴリ。 ■ Plug-in Manager によりまだ生成されていないシステム ID を持つ新しいカテゴリのサブカテゴリ。 プラグインが親カテゴリを識別した後は、getSystemID() メソッドによりそのシステム ID を取得できる。	*parentSystemID*	public int getParentSystemID()
関係するプラグイン クラスを識別する配列。サブクラスにより提供される KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。	*classNames*	public java.lang.String getClassName(int *key*)

詳細については、`com.bea.wlpi.common.plugin.ActionCategoryInfo` Javadoc を参照してください。

ActionInfo オブジェクト

com.bea.wlpi.common.plugin.ActionInfo オブジェクトは、プラグイン アクション情報を管理します。

ActionCategoryInfo は、ActionInfo オブジェクトの抽象基本クラスです。

ActionInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい ActionInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public ActionInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  int parentSystemID,
  int actionStateMask,
  int actionStateTrans,
  java.lang.String[] subActionLabels,
  java.lang.String[] classNames
)

public ActionInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  int parentSystemID,
  int actionStateMask,
  java.lang.String[] classNames
)
```

次の表に、ActionInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-2 ActionInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	`public java.lang.String getPluginName()`
アクション ID	*ID*	`public int getID()`
アクションのローカライズされた名前	*name*	`public java.lang.String getName()`
アクションのローカライズされた説明	*description*	`public java.lang.String getDescription()`
このプラグインのためのグラフィカル イメージ（アイコン）のバイト配列表現。インタフェース ビューが有効な場合に、このアクションを表すために Studio により使用される。 バイト配列表現の生成方法の詳細については、B-30 ページの「InfoObject オブジェクト」を参照。	*iconByteArray*	`public javax.swing.Icon getIcon()` `public static final byte[] imageStreamToByteArray(java.io.InputStream` *inputStream*`) throws java.io.IOException`

表 B-2 ActionInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
アクション ツリーでアクション カテゴリを識別する親カテゴリの ID。 この ID は次のいずれかの値に設定できる。 ■ ID_EXCEPTION — Studio の Exception Handling Actions カテゴリ。 ■ ID_INTEGRATION — Studio の Integration Actions カテゴリ。 ■ ID_MISCELLANEOUS — Studio の Miscellaneous Actions カテゴリ。 ■ ID_NEW — Initial カテゴリ。 ■ ID_TASK — Studio の Task Actions カテゴリ。 ■ ID_WORKFLOW — Studio の Workflow Actions カテゴリ。 親システム ID に対するアクセス権を持たない新しいアクション カテゴリには、必ず ID_NEW を使用する。たとえば、次のカテゴリを追加する場合は、ID_NEW を使用する必要がある。 ■ ツリーのルートの下の新しいカテゴリ。 ■ Plug-in Manager によりまだ生成されていないシステム ID を持つ新しいカテゴリのサブカテゴリ。 プラグインが親カテゴリを識別した後は、getSystemID() メソッドによりそのシステム ID を取得できる。	*parentSystemID*	public int getParentSystemID()

表 B-2 ActionInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
このアクションが有効であるアクション ステートを示すビット マスク。 この値は、指定された値のビット演算子 OR を実行することにより作成される。 ビット マスクは、アクションが有効なときを指定する次の値の 1 つまたは複数に設定できる。 ■ ACTION_STATE_ALL －アクションは、すべてのアクション ステートに対して有効。 ■ ACTION_STATE_COMMIT －アクションは、例外ハンドラで有効。 ■ ACTION_STATE_COMMIT_ASYNC －アクションは、例外ハンドラ コミット パスで非同期的に実行されるサブアクションとして有効。 ■ ACTION_STATE_COMMIT_SYNC －アクションは、例外ハンドラ コミット パスで同期的に実行されるサブアクションとして有効。 ■ ACTION_STATE_NODE － アクションは、ノードで有効。 ■ ACTION_STATE_NODE_ASYNC －アクションは、ノードで非同期的に実行されるサブアクションとして有効。 ■ ACTION_STATE_NODE_SYNC －アクションは、ノードで同期的に実行されるサブアクションとして有効。 ■ ACTION_STATE_ROLLBACK －アクションは、例外ハンドラ ロールバック パスで有効。 ■ ACTION_STATE_ROLLBACK_ASYNC －アクションは、例外ハンドラ ロールバック パスで非同期的に実行されるサブアクションとして有効。 ■ ACTION_STATE_ROLLBACK_SYNC －アクションは、例外ハンドラ ロールバック パスで同期的に実行されるサブアクションとして有効。	*actionStateMask*	public boolean isValidActionState (int *actionStateMask*)

表 B-2 ActionInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
アクションのサブアクションの実行に起因するアクション ステート遷移。 この ステート は次のいずれかの値に設定できる。 ■ ACTION_STATE_TRANS_NONE －アクションは、サブアクションを持たない。 ■ ACTION_STATE_TRANS_ASYNC －アクションは、システムが非同期的に呼び出すサブアクションを持つ。 ■ ACTION_STATE_ROLLBACK_SYNC －アクションは、システムが同期的に呼び出すサブアクションを持つ。	*actionStateTrans*	public int getActionStateTrans()
ローカライズされたアクション リスト ラベル。	*subActionLabels*	public java.lang.String[] getSubActionLabels()
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。 ■ KEY_ACTION － com.bea.wlpi.server.plugin.PluginAction 実装クラス名を示すキー値。 ■ KEY_DATA － com.bea.wlpi.common.plugin.PluginActionData 実装クラス名を示すキー値。 ■ KEY_PANEL － com.bea.wlpi.common.plugin.PluginActionPanel 実装クラス名を示すキー値。	*classNames*	public java.lang.Object getClass(int *key*)

詳細については、com.bea.wlpi.common.plugin.ActionInfo Javadoc を参照してください。

CategoryInfo オブジェクト

com.bea.wlpi.common.plugin.CategoryInfo オブジェクトは、プラグイン アクション カテゴリに関する情報を管理します。

ActionCategoryInfo は、CategoryInfo オブジェクトの抽象基本クラスです。

The CategoryInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい CategoryInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public CategoryInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  int parentSystemID,
  com.bea.wlpi.common.plugin.ActionCategoryInfo[] subNodes
)
```

次の表に、CategoryInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-3 CategoryInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド	set メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()	なし
アクション カテゴリ ID	*ID*	public int getID()	なし
アクション カテゴリのローカライズされた名前	*name*	public java.lang.String getName()	なし

表 B-3 CategoryInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド	set メソッド
アクション カテゴリのローカライズされた説明	*description*	`public java.lang.String getDescription()`	なし

表 B-3 CategoryInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド	set メソッド
アクション ツリーでカテゴリを識別する親カテゴリの ID この ID は次のいずれかの値に設定できる。 ■ ID_EXCEPTION — Studio の Exception Handling Actions カテゴリ。 ■ ID_INTEGRATION — Studio の Integration Actions カテゴリ。 ■ ID_MISCELLANEOUS — Studio の Miscellaneous Actions カテゴリ。 ■ ID_NEW — Initial カテゴリ。 ■ ID_TASK — Studio の Task Actions カテゴリ。 ■ ID_WORKFLOW — Studio の Workflow Actions カテゴリ。 親システム ID に対するアクセス権を持たない新しいアクション カテゴリには、必ず ID_NEW を使用する。たとえば、次のカテゴリを追加する場合は、ID_NEW を使用する必要がある。 ■ ツリーのルートの下の新しいカテゴリ。 ■ Plug-in Manager によりまだ生成されていないシステム ID を持つ新しいカテゴリのサブカテゴリ。 プラグインが親カテゴリを識別した後は、getSystemID() メソッドによりそのシステム ID を取得できる。	*parentSystemID*	public int getParentSystemID()	なし

表 B-3 CategoryInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド	set メソッド
関係するプラグイン クラスを識別する配列。サブクラスにより提供される KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。	*subNodes*	`public com.bea.wlpi.common.plugin.ActionCategoryInfo[] getSubnodes()`	`public void addSubNode(com.bea.wlpi.common.plugin.ActionCategoryInfo node)`
プラグイン システム ID	なし	`public int getSystemID()`	`public int setSystemID(int systemID)`

注意： 前の表で定義したメソッドの他に、次のメソッドがアクション カテゴリとそのサブカテゴリを再帰的に検索し、システム ID の一致するカテゴリを探します。

詳細については、`com.bea.wlpi.common.plugin.CategoryInfo` Javadoc を参照してください。

ConfigurationData オブジェクト

`com.bea.wlpi.common.plugin.ConfigurationData` オブジェクトは、プラグイン コンフィグレーション情報を管理します。

新しい ConfigurationData オブジェクトを作成するには、次のコンストラクタを使用します。

```
public ConfigurationData(
  java.lang.String pluginName,
  com.bea.wlpi.common.VersionInfo version,
  int status,
  int startMode,
  java.lang.String xml
)
```

次の表に、ConfigurationData オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-4 ConfigurationData オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	`public java.lang.String getPluginName()`
プラグイン バージョン	*version*	`public com.bea.wlpi.common.VersionInfo getVersion()`
プラグイン ステータス	*status*	`public int getStatus()`
プラグイン起動モード	*startMode*	`public int getStartMode()`
XML コンフィグレーション ドキュメント	*xml*	`public java.lang.String getXML()`

詳細については、`com.bea.wlpi.common.plugin.ConfigurationData` Javadoc を参照してください。

ConfigurationInfo オブジェクト

com.bea.wlpi.common.plugin.ConfigurationInfo オブジェクトは、プラグイン コンフィグレーション情報を管理します。

ConfigurationInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい ConfigurationInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public ConfigurationInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String description,
  java.lang.String[] classNames
)
```

次の表に、ConfigurationInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-5 ConfigurationInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
プラグインの説明	*description*	public int getStatus()
プラグイン起動モード	*startMode*	public int getStartMode()

表 B-5 ConfigurationInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ （Java クラスの完全修飾名） が入る。 ■ KEY_DATA — com.bea.wlpi.common.plugin.PluginData 実装クラス名を示すキー値。 ■ KEY_PANEL — com.bea.wlpi.common.plugin.PluginPanel 実装クラス名を示すキー値。	*classNames*	public java.lang.String getClassName(int *key*)

詳細については、com.bea.wlpi.common.plugin.ConfigurationInfo Javadoc を参照してください。

DoneInfo オブジェクト

com.bea.wlpi.common.plugin.DoneInfo オブジェクトは、プラグイン完了ノードに関する情報を管理します。

DoneInfo クラスは、次のクラスを拡張します。

- B-30 ページの 「InfoObject オブジェクト」 に説明されている com.bea.wlpi.common.plugin.InfoObject クラス
- B-41 ページの 「TemplateNodeInfo オブジェクト」 に説明されている com.bea.wlpi.common.plugin.TemplateNodeInfo クラス

新しい DoneInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public DoneInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
```

```
  byte[] iconByteArray,
  java.lang.String[] classNames
)
```

次の表に、DoneInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-6 DoneInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
完了ノードのローカライズされた名前	*name*	public java.lang.String getName()
完了ノードのローカライズされた説明	*description*	public java.lang.String getDescription()
このプラグインのためのグラフィカル イメージ（アイコン）のバイト配列表現。インタフェース ビューが有効な場合に、このアクションを表すために Studio により使用される。 バイト配列表現の生成方法の詳細については、B-30 ページの「InfoObject オブジェクト」を参照。	*iconByteArray*	public javax.swing.Icon getIcon() public static final byte[] imageStreamToByteArray(java.io.InputStream *inputStream*) throws java.io.IOException

表 B-6 DoneInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。 ■ KEY_DATA — com.bea.wlpi.common.plugin.PluginData 実装クラス名を示すキー値。 ■ KEY_PANEL — com.bea.wlpi.common.plugin.PluginPanel 実装クラス名を示すキー値。 ■ KEY_DONE — com.bea.wlpi.server.plugin.PluginDone 実装クラス名を示すキー値。	*classNames*	public java.lang.String getClassName(int *key*)

詳細については、com.bea.wlpi.common.plugin.DoneInfo Javadoc を参照してください。

EventHandlerInfo オブジェクト

com.bea.wlpi.common.plugin.EventHandlerInfo オブジェクトは、プラグイン イベント ハンドラに関する情報を管理します。

EventHandlerInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい EventHandlerInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public EventHandlerInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
```

```
  java.lang.String[] classNames
)
```

次の表に、EventHandlerInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-7 EventHandlerInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
イベント ハンドラのローカライズされた名前	*name*	public java.lang.String getName()
イベント ハンドラのローカライズされた説明	*description*	public java.lang.String getDescription()
関係するプラグイン クラスを識別する配列。次の KEY_* 値について 1 つのエントリ（Java クラスの完全修飾名）が入る。 KEY_HANDLER ― com.bea.wlpi.server.plugin.EventHandler 実装クラス名を示すキー値。	*classNames*	public java.lang.String getClassName(int *key*)

詳細については、com.bea.wlpi.common.plugin.EventHandlerInfo Javadoc を参照してください。

EventInfo オブジェクト

com.bea.wlpi.common.plugin.EventInfo オブジェクトは、プラグイン イベント ハンドラに関する情報を管理します。

EventInfo クラスは、次のクラスを拡張します。

- B-30 ページの「InfoObject オブジェクト」に説明されている com.bea.wlpi.common.plugin.InfoObject クラス
- B-41 ページの「TemplateNodeInfo オブジェクト」に定義されている com.bea.wlpi.common.plugin.TemplateNodeInfo クラス

新しい EventInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public EventInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  java.lang.String[] classNames,
  com.bea.wlpi.common.plugin.FieldInfo fieldInfo
)
```

次の表に、EventInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-8 EventInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
イベントのローカライズされた名前	*name*	public java.lang.String getName()

表 B-8 EventInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
イベントのローカライズされた説明	*description*	public java.lang.String getDescription()
このプラグインのためのグラフィカル イメージ（アイコン）のバイト配列表現。インタフェース ビューが有効な場合に、このアクションを表すために Studio により使用される。 バイト配列表現の生成方法の詳細については、B-30 ページの「InfoObject オブジェクト」を参照。	*iconByteArray*	public javax.swing.Icon getIcon() public static final byte[] imageStreamToByteArray(java.io.InputStream *inputStream*) throws java.io.IOException
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。 ■ KEY_DATA — com.bea.wlpi.common.plugin.PluginData 実装クラス名を示すキー値。 ■ KEY_PANEL — com.bea.wlpi.common.plugin.PluginTriggerPanel 実装クラス名を示すキー値。 ■ KEY_EVENT — com.bea.wlpi.server.plugin.PluginEvent 実装クラス名を示すキー値。	*classNames*	public java.lang.String getClassName(int *key*)
プラグイン フィールド情報	*fieldInfo*	public com.bea.wlpi.common.plugin.FieldInfo getFieldInfo()

詳細については、com.bea.wlpi.common.plugin.EventInfo Javadoc を参照してください。

FieldInfo オブジェクト

com.bea.wlpi.common.plugin.FieldInfo オブジェクトは、プラグイン フィールドに関する情報を管理します。

FieldInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい FieldInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public FieldInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames,
  boolean supportsQualifiers
)
```

次の表に、FieldInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-9 FieldInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
フィールドのローカライズされた名前	*name*	public java.lang.String getName()
フィールドのローカライズされた説明	*description*	public java.lang.String getDescription()

表 B-9 FieldInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ （Java クラスの完全修飾名） が入る。 ■ KEY_FIELD ― com.bea.wlpi.common.plugin.PluginField 実装クラス名を示すキー値。 ■ DEFAULT_FIELD ―式の検証のみをサポートし、評価を行わない組み込みフィールド タイプを示すキー値。このフィールド タイプは修飾子をサポートし、プラグイン自体が現在ロードされていない場合に、プラグイン データ フィールドを参照する式を検証するために使用される。 ■ XMLFIELD ― XML 単一要素値を返す組み込みフィールド タイプを示すキー値。このフィールド タイプは修飾子をサポートする （XML は階層構造であるため）。	*classNames*	public java.lang.String getClassName(int *key*)
フィールド タイプがドット区切りの名前をサポートするかどうかを示すフラグ。たとえば、PostalCode フィールドが Address フィールドに埋め込まれている場合、そのフィールドの名前は Address.PostalCode となる。 式エバリュエータは、プラグインの 開始、イベント、または イベント キーの式でフィールド参照が有効であるかどうかを調べるためにこのフラグを使用する。	*supportsQualifiers*	public boolean supportsQualifiers()

詳細については、com.bea.wlpi.common.plugin.FieldInfo Javadoc を参照してください。

FunctionInfo オブジェクト

com.bea.wlpi.common.plugin.FunctionInfo オブジェクトは、プラグイン関数に関する情報を管理します。

FunctionInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい FunctionInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public FunctionInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  java.lang.String prototype,
  java.lang.String[] classNames,
  int argcmin,
  int argcmax
)
```

次の表に、FunctionInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-10 FunctionInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
関数のローカライズされた名前	*name*	public java.lang.String getName()
関数のローカライズされた説明	*description*	public java.lang.String getDescription()
この関数のローカライズされたプロトタイプ	*prototype*	public java.lang.String prototype()

表 B-10 FunctionInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
関係するプラグイン クラスを識別する配列。次の KEY_* 値について 1 つのエントリ（Java クラスの完全修飾名）が入る。 KEY_EVALUATOR ― com.bea.wlpi.common.plugin.PluginFunction 実装クラス名を示すキー値。	*classNames*	public java.lang.String getClassName(int *key*)
許される引数の最小数。式エバリュエータは、この関数に対するコールを検証するためにこの情報を使用する。	*argcmin*	public int getMinArgCount()
許される引数の最大数。式エバリュエータは、この関数に対するコールを検証するためにこの情報を使用する。	*argcmax*	public int getMaxArgCount()

詳細については、com.bea.wlpi.common.plugin.FunctionInfo Javadoc を参照してください。

HelpSetInfo オブジェクト

com.bea.wlpi.common.plugin.HelpSetInfo オブジェクトは、プラグイン オンライン ヘルプに関する情報を管理します。プラグインは、HTML および JavaHelp の両方のオンライン ヘルプ システムをサポートできます。プラグイン オンライン ヘルプ ファイルは、WAR ファイルにパッケージ化し、プロセス エンジンの一部としてデプロイする必要があります。BPM クライアント アプリケーションが適切なヘルプ ファイルを取得できるようにするため、オンライン ヘルプ WAR ファイルは、それが関係するプラグインの名前の下でデプロイする必要があります。

BPM クライアント アプリケーションは、Plug-in Manager （または他の EJB）の ClassLoader を使用して、プロセス エンジンの URL を確認できます。クライアント アプリケーションは、HelpSetInfo オブジェクト値を使用して、http または https を介してヘルプ ファイルにアクセスするための完全な URL を取得できます。

HelpSetInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい HelpSetInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public HelpSetInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] helpNames,
  int helpType
)
```

次の表に、HelpSetInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-11 HelpSetInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
オンライン ヘルプ セットを提供するプラグイン名（逆引き DNS バージョン）。 BPM クライアント アプリケーションがヘルプ セットの正しい URL を作成するためには、この名前は、WAR ファイルをデプロイする Web アプリケーション と一致している必要があります。	*pluginName*	public java.lang.String getPluginName()
オンライン ヘルプのローカライズされた名前。 この値は、コンテキスト依存でないヘルプにアクセスするために使用されるユーザ インタフェース メニュー オプションのラベル値として使用される。	*name*	public java.lang.String getName()
オンライン ヘルプのローカライズされた説明	*description*	public java.lang.String getDescription()
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。 ■ KEY_HELP_SET ― JavaHelp ヘルプ セット（.hs）ファイル名または HTML ヘルプ ファイル ルート ディレクトリを取得するためのキー値。 ■ KEY_HELP_ID ―メイン インデックス ページまたは目次の JavaHelp ヘルプ キーまたは HTML ファイル名を取得するためのキー値。 各エントリの値は、B-29 ページの「ヘルプ タイプおよび関係するプラグイン クラス」の表に定義されているように、*helpType* パラメータの値に従って解釈される。	*helpNames*	public java.lang.String getClassName(int *key*)

表 B-11 HelpSetInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグインにより提供されるヘルプのタイプ。次のいずれかの値に設定できる。 ■ HELP_JAVA_HELP ― JavaHelp ヘルプ セットの名前、またはメイン インデックス ページまたは目次のためのヘルプ キー。 ■ HELP_HTML ― HTML ヘルプ ファイル ルート ディレクトリの名前、あるいはメイン インデックス ページまたは目次の名前。KEY_HELP_SET エントリにより指定されるディレクトリに対する相対値。 各エントリの値は、B-29 ページの「ヘルプ タイプおよび関係するプラグイン クラス」の表に定義されているように、*helpNames* パラメータの値に従って解釈される。	*helpType*	public int getHelpType()

次の表に、ヘルプ タイプ（*helpType* 値）および関係するプラグイン クラス（*helpNames* 値）を示します。

表 B-12 ヘルプ タイプおよび関係するプラグイン クラス

helpType 値	*helpNames* 値	
	KEY_HELP_SET	KEY_HELP_ID
HELP_JAVA_HELP	JavaHelp セット ファイルの名前。ヘルプ ファイルの入っている WAR ファイルのルートに対する相対値（たとえば、javahelp/MyPluginHelpSet.hs）。拡張子がない場合、JavaHelp により自動的に hs 拡張子が追加される。	メイン インデックス ページまたは目次のための JavaHelp ヘルプ キー。

表 B-12 ヘルプ タイプおよび関係するプラグイン クラス （続き）

helpType 値	*helpNames* 値	
	KEY_HELP_SET	KEY_HELP_ID
HELP_HTML	HTML ヘルプ ファイルのルート ディレクトリの名前。末尾に必ずスラッシュを付ける （たとえば、htmlhelp/）。	メイン インデックス ページまたは目次の入っている HTML ファイルの名前（必須の .htm 拡張子を含まない）。KEY_HELP_SET エントリにより指定されるディレクトリに対する相対値。

詳細については、com.bea.wlpi.common.plugin.HelpSetInfo Javadoc を参照してください。

InfoObject オブジェクト

com.bea.wlpi.common.plugin.InfoObject オブジェクトは、すべてのプラグイン値オブジェクトのための抽象基本クラスを提供します。

新しい InfoObject オブジェクトを作成するには、次のコンストラクタを使用します。

```
public InfoObject(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames
)
```

次の表に、InfoObject オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-13 InfoObject オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
オブジェクトのローカライズされた名前	*name*	public java.lang.String getName()
オブジェクトのローカライズされた説明	*description*	public java.lang.String getDescription()
関係するプラグイン クラスを識別する配列。サブクラスにより提供される KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。	*classNames*	public java.lang.String getClassName(int *key*)

また、InfoObject オブジェクトは、ActionInfo、DoneInfo、EventInfo、StartInfo、および TemplateDefinitionPropertiesInfo の各オブジェクトをコンストラクトする際に使用できる *iconByteArray* 値を入力ストリームから生成するための次のメソッドも提供します。

```
public static final byte[]
imageStreamToByteArray(java.io.InputStream inputStream) throws
java.io.IOException
```

詳細については、com.bea.wlpi.common.plugin.InfoObject Javadoc を参照してください。

PluginCapabilitiesInfo オブジェクト

com.bea.wlpi.common.plugin.PluginCapabilitiesInfo オブジェクトは、プラグイン機能に関する情報を管理します。

PluginCapabilitiesInfo オブジェクトは、プラグインがロードされた後、プラグイン機能の完全なセットの説明を提供します。ロード前でも、B-35 ページの「PluginInfo オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.PluginInfo オブジェクトを使用して基本プラグイン情報にアクセスできます。

新しい PluginCapabilitiesInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public PluginCapabilitiesInfo(
  com.bea.wlpi.common.plugin.PluginInfo info,
  com.bea.wlpi.common.plugin.ActionCategoryInfo[] actions,
  com.bea.wlpi.common.plugin.EventInfo[] events,
  com.bea.wlpi.common.plugin.FieldInfo[] fields,
  com.bea.wlpi.common.plugin.FunctionInfo[] functions,
  com.bea.wlpi.common.plugin.StartInfo[] starts,
  com.bea.wlpi.common.plugin.DoneInfo[] dones,
  com.bea.wlpi.common.plugin.VariableTypeInfo[] variableTypes,
  com.bea.wlpi.common.plugin.TemplatePropertiesInfo[] template,
  com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo[]
    templateDefinition,
  com.bea.wlpi.common.plugin.EventHandlerInfo eventHandler
)
```

次の表に、PluginCapabilitiesInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-14 PluginCapabilitiesInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
基本プラグイン情報	*info*	public java.lang.String getPluginInfo()
プラグインにより提供されるアクションとアクション カテゴリ	*actions*	public com.bea.wlpi.common.plugin.ActionCategoryInfo[] getActionInfo()
プラグインにより提供されるイベント	*events*	public com.bea.wlpi.common.plugin.EventInfo[] getEventInfo()

表 B-14 PluginCapabilitiesInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグインにより提供されるフィールド	*fields*	public com.bea.wlpi.common.plugin.FieldInfo[] getFieldInfo()
プラグインにより提供される関数	*functions*	public com.bea.wlpi.common.plugin.FunctionInfo[] getFunctionInfo()
プラグインにより提供される開始ノード	*starts*	public com.bea.wlpi.common.plugin.StartInfo[] getStartInfo()
プラグインにより提供される完了ノード	*dones*	public com.bea.wlpi.common.plugin.DoneInfo[] getDoneInfo()
プラグインにより提供される変数	*variableTypes*	public com.bea.wlpi.common.plugin.VariableTypesInfo[] getVariableTypesInfo()
プラグインにより提供されるワークフロー テンプレート プロパティ	*template*	public com.bea.wlpi.common.plugin.TemplatePropertiesInfo[] getTemplateInfo()
プラグインにより提供されるワークフロー テンプレート定義プロパティ	*templateDefinition*	public com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo[] getTemplateDefinitionInfo()
イベント ハンドラ情報	*eventHandler*	public com.bea.wlpi.common.plugin.EventHandlerInfo getEventHandlerInfo()

詳細については、com.bea.wlpi.common.plugin.PluginCapabilitiesInfo Javadoc を参照してください。

PluginDependency オブジェクト t

com.bea.wlpi.common.plugin.PluginDependency オブジェクトは、プラグイン依存関係に関する情報を管理します。

PluginDependency クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい PluginDependency オブジェクトを作成するには、次のコンストラクタを使用します。

```
public PluginDependency(
  java.lang.String pluginName,
  java.lang.String description,
  java.lang.String masterPluginName,
  java.lang.String vendor,
  com.bea.wlpi.common.VersionInfo version
)
```

次の表に、PluginDependency オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-15 PluginDependency オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
オブジェクトのローカライズされた説明	*description*	public java.lang.String getDescription()
マスター プラグイン名（逆引き DNS バージョン）	*masterPluginName*	public java.lang.String getMasterPluginName()

表 B-15 PluginDependency オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
マスター プラグイン ベンダ名	*vendor*	`public java.lang.String getVendor()`
マスター プラグイン バージョン	*version*	`public com.bea.wlpi.common.VersionInfo getVersion()`

詳細については、`com.bea.wlpi.common.plugin.PluginDependency` Javadoc を参照してください。

PluginInfo オブジェクト

`com.bea.wlpi.common.plugin.PluginInfo` オブジェクトは、プラグインに関する基本情報を管理します。

`PluginInfo` オブジェクトは、プラグインがロードされる前に、プラグイン機能の基本セットの説明を提供します。ロード後は、B-31 ページの「PluginCapabilitiesInfo オブジェクト」に説明されているように、`com.bea.wlpi.common.plugin.PluginCapabilitiesInfo` オブジェクトを使用して基本プラグイン情報にアクセスできます。

`PluginInfo` クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、`com.bea.wlpi.common.plugin.InfoObject` クラスを拡張します。

新しい `PluginInfo` オブジェクトを作成するには、次のコンストラクタを使用します。

```
public PluginInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.util.Locale lc,
  java.lang.String vendor,
  java.lang.String url,
  com.bea.wlpi.common.VersionInfo version,
  com.bea.wlpi.common.VersionInfo pluginFrameworkVersion,
  com.bea.wlpi.common.plugin.PluginDependency[] dependencies,
  com.bea.wlpi.common.plugin.ConfigurationInfo config,
```

```
  com.bea.wlpi.common.plugin.HelpSetInfo helpSet
)
```

次の表に、PluginInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-16 PluginInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
オブジェクトのローカライズされた名前	*name*	public java.lang.String getName()
表示文字列をローカライズするためのロケール	*lc*	public java.lang.String getLocale()
プラグイン ベンダの名前	*vendor*	public java.lang.String getVendor()
プラグイン ベンダの URL	*url*	public java.lang.String getURL()
プラグイン バージョン	*version*	public com.bea.wlpi.common.VersionInfo getVersion()
プラグイン フレームワーク バージョン	*pluginFrameworkVersion*	public com.bea.wlpi.common.VersionInfo getPluginFrameworkVersion()
プラグイン依存関係	*dependencies*	public com.bea.wlpi.common.plugin.PluginDependency[] getDependencyInfo()
プラグイン コンフィグレーション情報	*config*	public com.bea.wlpi.common.plugin.ConfigurationInfo getConfigurationInfo()

表 B-16 PluginInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグインにより提供される JavaHelp ヘルプセット	*helpSet*	`public com.bea.wlpi.common.plugin.HelpSetInfo getHelpSetInfo()`

詳細については、`com.bea.wlpi.common.plugin.PluginInfo` Javadoc を参照してください。

StartInfo オブジェクト

`com.bea.wlpi.common.plugin.StartInfo` オブジェクトは、プラグイン開始ノードに関する情報を管理します。

`StartInfo` クラスは、次のクラスを拡張します。

- B-30 ページの 「InfoObject オブジェクト」 に説明されている `com.bea.wlpi.common.plugin.InfoObject` クラス
- B-41 ページの 「TemplateNodeInfo オブジェクト」 に定義されている `com.bea.wlpi.common.plugin.TemplateNodeInfo` クラス

新しい `StartInfo` オブジェクトを作成するには、次のコンストラクタを使用します。

```
public StartInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  java.lang.String[] classNames,
  com.bea.wlpi.common.plugin.FieldInfo fieldInfo
)
```

次の表に、`StartInfo` オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-17 StartInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	`public java.lang.String getPluginName()`
プラグイン ID	*ID*	`public int getID()`
開始ノードのローカライズされた名前	*name*	`public java.lang.String getName()`
開始ノードのローカライズされた説明	*description*	`public java.lang.String getDescription()`
このプラグインのためのグラフィカル イメージ（アイコン）のバイト配列表現。インタフェース ビューが有効な場合に、このアクションを表すために Studio により使用される。 バイト配列表現の生成方法の詳細については、B-30 ページの「InfoObject オブジェクト」を参照。	*iconByteArray*	`public javax.swing.Icon getIcon()` `public static final byte[] imageStreamToByteArray(java.io.InputStream inputStream) throws java.io.IOException`
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。 ■ KEY_DATA — `com.bea.wlpi.common.plugin.PluginData` 実装クラス名を示すキー値。 ■ KEY_PANEL — `com.bea.wlpi.common.plugin.PluginPanel` 実装クラス名を示すキー値。 ■ KEY_START — `com.bea.wlpi.server.plugin.PluginStart2` 実装クラス名を示すキー値。	*classNames*	`public java.lang.String getClassName(int key)`
プラグイン フィールド情報	*fieldInfo*	`public com.bea.wlpi.common.plugin.FieldInfo getFieldInfo()`

詳細については、`com.bea.wlpi.common.plugin.StartInfo` Javadoc を参照してください。

TemplateDefinitionPropertiesInfo オブジェクト

com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo オブジェクトは、プラグイン テンプレート定義プロパティに関する情報を管理します。

TemplateDefinitionPropertiesInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい TemplateDefinitionPropertiesInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public DoneInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames
)
```

次の表に、TemplateDefinitionPropertiesInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-18 TemplateDefinitionPropertiesInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
オブジェクトのローカライズされた名前。この文字列は、[テンプレート定義のプロパティ] ダイアログ ボックスの [プラグイン] タブの内容を定義する。	*name*	public java.lang.String getName()
オブジェクトのローカライズされた説明	*description*	public java.lang.String getDescription()

表 B-18 TemplateDefinitionPropertiesInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ （Java クラスの完全修飾名） が入る。 ■ KEY_DATA — com.bea.wlpi.common.plugin.PluginData 実装クラス名を示すキー値。 ■ KEY_PANEL — com.bea.wlpi.common.plugin.PluginPanel 実装クラス名を示すキー値。	*classNames*	public java.lang.String getClassName(int *key*)

詳細については、com.bea.wlpi.common.plugin.TemplateDefinitionPropertiesInfo Javadoc を参照してください。

TemplateNodeInfo オブジェクト

com.bea.wlpi.common.plugin.TemplateNodeInfo オブジェクトは、プラグイン テンプレート定義ノードに関する情報を管理します。

TemplateNodeInfo は、次のクラスにより拡張されます。

- com.bea.wlpi.common.plugin.DoneInfo
- com.bea.wlpi.common.plugin.EventInfo
- com.bea.wlpi.common.plugin.StartInfo

TemplateNodeInfo クラスは、B-30 ページの「InfoObject オブジェクト」 に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい TemplateNodeInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public TemplateNodeInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  byte[] iconByteArray,
  java.lang.String[] classNames
)
```

次の表に、TemplateNodeInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-19 TemplateNodeInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
テンプレート定義ノードのローカライズされた名前	*name*	public java.lang.String getName()
テンプレート定義ノードのローカライズされた説明	*description*	public java.lang.String getDescription()
このプラグインのためのグラフィカル イメージ（アイコン）のバイト配列表現。インタフェース ビューが有効な場合に、このアクションを表すために Studio により使用される。 バイト配列表現の生成方法の詳細については、B-30 ページの「InfoObject オブジェクト」を参照。	*iconByteArray*	public javax.swing.Icon getIcon() public static final byte[] imageStreamToByteArray(java.io.InputStream *inputStream*) throws java.io.IOException
関係するプラグイン クラスを識別する配列。対応するサブクラスのために定義される KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。	*classNames*	public java.lang.String getClassName(int *key*)

詳細については、com.bea.wlpi.common.plugin.TemplateNodeInfo Javadoc を参照してください。

TemplatePropertiesInfo オブジェクト

com.bea.wlpi.common.plugin.TemplatePropertiesInfo オブジェクトは、プラグイン テンプレート プロパティに関する情報を管理します。

TemplatePropertiesInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい TemplatePropertiesInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public TemplatePropertiesInfo(
  java.lang.String pluginName,
  java.lang.String name,
  java.lang.String description,
  java.lang.String[] classNames
)
```

次の表に、TemplatePropertiesInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-20 TemplatePropertiesInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
オブジェクトのローカライズされた名前。この文字列は、[テンプレート プロパティ] ダイアログ ボックスの [プラグイン] タブの内容を定義する。	*name*	public java.lang.String getName()
オブジェクトのローカライズされた説明	*description*	public java.lang.String getDescription()

表 B-20 TemplatePropertiesInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。 ■ KEY_DATA — com.bea.wlpi.common.plugin.PluginData 実装クラス名を示すキー値。 ■ KEY_PANEL — com.bea.wlpi.common.plugin.PluginPanel 実装クラス名を示すキー値。	*classNames*	public java.lang.String getClassName(int *key*)

詳細については、com.bea.wlpi.common.plugin.TemplatePropertiesInfo Javadoc を参照してください。

VariableTypeInfo オブジェクト

com.bea.wlpi.common.plugin.VariableTypeInfo オブジェクトは、プラグイン変数に関する情報を管理します。

VariableTypeInfo クラスは、B-30 ページの「InfoObject オブジェクト」に説明されているように、com.bea.wlpi.common.plugin.InfoObject クラスを拡張します。

新しい VariableTypeInfo オブジェクトを作成するには、次のコンストラクタを使用します。

```
public VariableTypeInfo(
  java.lang.String pluginName,
  int ID,
  java.lang.String name,
  java.lang.String description,
  int variableType,
  java.lang.Class valueClass,
  java.lang.String[] classNames
)
```

次の表に、VariableTypeInfo オブジェクト情報、そのデータを定義する際に使用するコンストラクタ パラメータ、オブジェクト定義後にこの情報にアクセスするために使用できるメソッドを示します。

表 B-21 VariableTypeInfo オブジェクト情報

オブジェクト情報	コンストラクタ パラメータ	get メソッド
プラグイン名（逆引き DNS バージョン）	*pluginName*	public java.lang.String getPluginName()
プラグイン ID	*ID*	public int getID()
変数タイプのローカライズされた名前	*name*	public java.lang.String getName()
変数タイプのローカライズされた説明	*description*	public java.lang.String getDescription()
変数タイプは次のいずれかの整数値に設定できる。 ■ TYPE_ENTITY －エンティティ EJB に対するリモート参照。 ■ TYPE_OBJECT －ローカル Java オブジェクト。 ■ TYPE_SESSION －セッション EJB に対するリモート参照。	*variableType*	public int getVariableType()
許される値タイプの完全修飾 Java クラス	*valueClass*	public java.lang.Class getValueClass()

表 B-21 VariableTypeInfo オブジェクト情報 （続き）

オブジェクト情報	コンストラクタ パラメータ	get メソッド
関係するプラグイン クラスを識別する配列。次の KEY_* 値のそれぞれについて 1 つのエントリ（Java クラスの完全修飾名）が入る。 ■ KEY_RENDERER — com.bea.wlpi.common.plugin.PluginVariableRenderer 実装クラス名を示すキー値。 ■ KEY_PANEL — com.bea.wlpi.common.plugin.PluginPanel 実装クラス名を示すキー値。	*classNames*	public java.lang.*String getClassName(int key*)

詳細については、com.bea.wlpi.common.plugin.VariableTypeInfo Javadoc を参照してください。

C　BPM グラフィカル ユーザ インタフェース スタイル シート

この付録では、Java Swing クラスに基づいてカスタム プラグインを設計する際に役立つ情報を示します。この付録の内容は以下のとおりです。

- プラグインの設計
- 対話式コンポーネントの操作
- プレゼンテーション コンポーネントの操作
- 推奨参考文献

プラグインの設計

ユーザの学習ニーズや情報ニーズに合うプラグインを設計するには、C-3 ページの「対話式コンポーネントの操作」や C-10 ページの「プレゼンテーション コンポーネントの操作」に説明した原則を、慎重にプランニングし、系統的に適用する必要があります。

プラグイン パネルを設計する手順は、次のとおりです。

1. BPM の機能をどのように拡張するかを決定します。たとえば、次のプログラミング構成要素の定義と実行時動作をプラグインにより変更できます。
   - アクション
   - 完了ノード
   - イベント ノード
   - 関数
   - メッセージ タイプ
   - 開始

- テンプレート プロパティ
- テンプレート定義プロパティ
- 変数タイプ

2. 構成要素を変更するために必要なタスクを調べます。この場合、各構成要素には個別のパネルが必要です。たとえば、テンプレート プロパティを変更する場合、パネル タスクとして、新しいプロパティの追加、更新、削除の機能（デフォルトでは利用できない機能）を含めることができます。
3. 各タスクを表すコントロールを選択します。次の表に、タスク カテゴリの各タイプの推奨コントロールを示します。

表 C-1 タスク カテゴリの推奨コントロール

タスク カテゴリ	コントロール
相互に排他的なオプションの選択	ラジオ ボタン
排他的でないオプションの選択	チェック ボックス
アクションの実行	コマンド ボタン
セットからの項目の選択	リスト ボックスまたはドロップダウン リスト ボックス
同時に大量の情報の入力または表示	テーブル
属性値の設定	テキスト入力フィールド

4. 次のプレゼンテーション要素を適切に使用して、パネル コントロールを表示する方法を決定します。
   - コントロール タイプを区別したり、視覚的に変化を付けるには、カラーを使用します。たとえば、テーブルの行を 1 行おきに目立たせるには、補色関係の 2 色を使用します。このテクニックにより、視覚的に変化がついて、読みやすくなります。
   - すべてのコントロール ラベルに 12 ポイント Arial のレギュラー フォントを使用します。
   - パネル情報の流れは横または縦のいずれかにします。たとえば、項目間を移動できるリスト ボックスでは、横のレイアウトが適しています。一

方、縦のレイアウトとしては、テキスト入力フィールドを縦に並べて配置する場合が最適の例です。

- パネルの垂直、水平、対角線の軸に対して、各コントロールをバランス良く配置します。たとえば、パネルの右下にコマンド ボタンを置いたら、他のコントロールは対角線に沿って配置してバランスを取ります。

5. デフォルトのグラフィックを使用しない場合は、タスクに割り当てたプラグイン アクションを表す 32 × 32 ピクセルのアイコンを作成します。
6. 各コントロールの値を検証し、構文とデータ型のエラーをチェックします。各エラー条件ごとに、問題を明確に指摘し、解決方法を示唆するエラー メッセージ ダイアログ ボックスを作成します。

対話式コンポーネントの操作

対話式コンポーネントにより、ユーザはプラグイン パネルと対話できます。適切なコンポーネントを選択するには、単に原則に従うのではなく、業界標準、会社標準、ユーザ ニーズの間のバランスを取る必要があります。

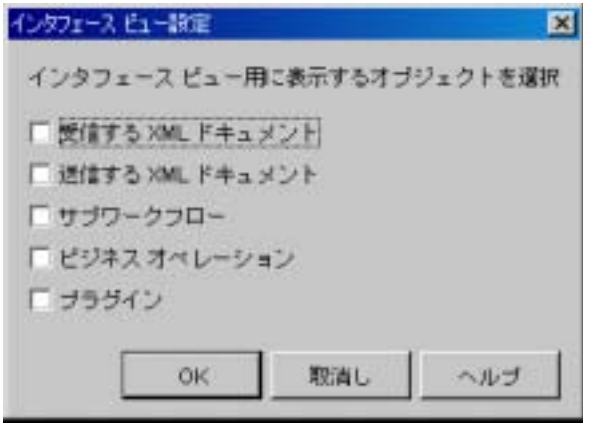
次の節では、対話式コンポーネントを設計するためのガイドラインを示します。

- チェック ボックス
- コマンド ボタン
- リスト ボックス
- ラジオ ボタン
- テーブル
- テキスト入力フィールド

チェック ボックス

チェック ボックスは、データ入力フィールドの代わりとして使用でき、複数の項目を迅速に選択する方法を提供します。次の BPM の例で、適切なチェック ボックスの設計を示します。

図 C-1　チェック ボックスの設計



チェック ボックスを設計するには、次のガイドラインに従ってください。

- ユーザがグループから複数の項目を選択できるようにしたり、機能のオン / オフを切り帰られるうにするには、チェック ボックスを使用します。
- グループ ボックスを使用して、複数のチェック ボックスをまとめて表示します。グループ ボックスには、分かりやいラベルを付けます。
- チェック ボックスは縦に並べます。
- チェック ボックスの数は 10 個以下にします。

コマンド ボタン

コマンド ボタンは、ユーザがダイアログ ボックス内での作業を完了するときに最もよく使用される対話式コンポーネントです。次の BPM の例で、適切なコマンド ボタンの設計を示します。

図 C-2　コマンド ボタンの設計

コマンド ボタンを設計するには、次のガイドラインに従ってください。

- コマンド ボタンは、[OK]、[取消し]、[ヘルプ] のような頻繁に使用されるアクションに対してのみ使用します。
- コマンド ボタンのラベルは慎重に選定します。アクションの意味を伝える必要がある場合は、複数の単語を使用します。
- コマンド ボタンのサイズは、互いに対応の取れるようにします。一連のボタンのラベルの長さがほぼ同じである場合、すべてのボタンのサイズを最大のボタンに合わせます。一連のボタンのラベルの長さに違いがある場合、2 種類のボタン サイズを使用し、1 つは短いラベル用、1 つは長いラベル用とします。
- 類似した機能を持つコマンド ボタンはグループにまとめます。
- 他のコントロールとコマンド ボタンとの間は余白で分離します。

リスト ボックス

リスト ボックスは、データ入力の代わりの手段を提供します。次の BPM の例で、適切なリスト ボックスの設計を示します。

図 C-3　リスト ボックスの設計

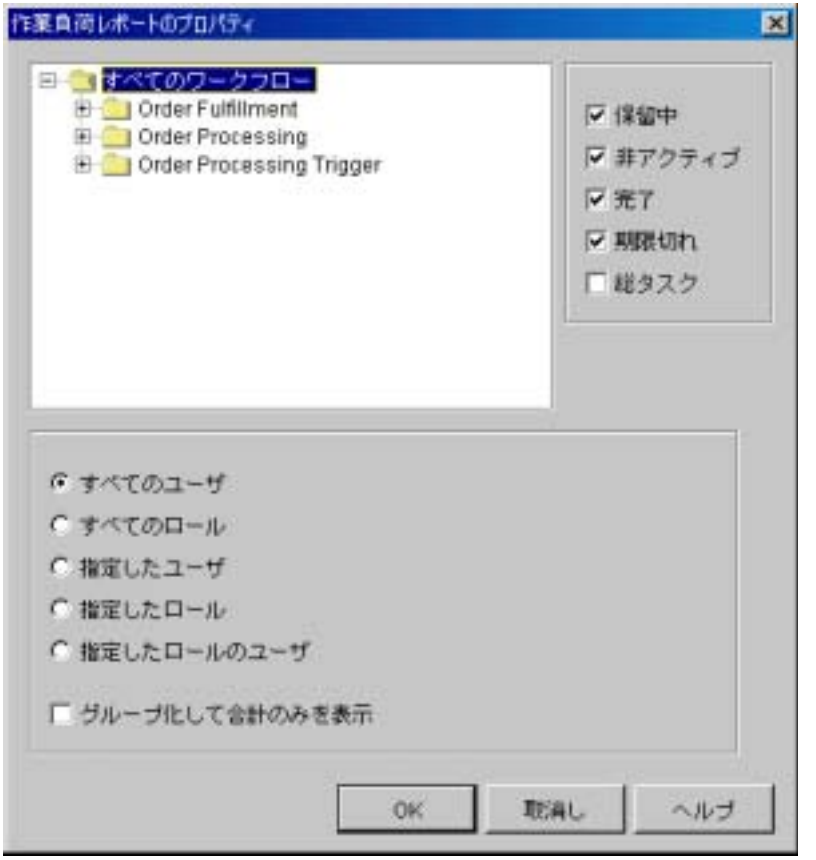
リスト ボックスを設計するには、次のガイドラインに従ってください。

- 7 個以上の項目がある場合は、できる限りラジオ ボタンではなくリスト ボックスを使用します。
- 少なくとも 3 個、ただし 7 個以内の項目を同時に表示します。9 個以上の項目がある場合は、縦スクロール バーを追加します。
- リストの項目を説明するようなラベルを選択します。リスト ボックスの上部のラベルは、左詰めにし、最後にコロンを付けます。たとえば、前の図では、ラベル [割り当てるタスク] がリストの上に表示されています。
- 大部分のユーザがリストの最初の項目を選択するような場合は、ドロップ・ダウン リスト ボックスにして、縦方向のスペースを節約します。

ラジオ ボタン

ラジオ ボタンは、多くのデータ入力フィールドの代わりに使用できます。次の BPM の例で、適切なラジオ ボタンの設計を示します。

図 C-4　ラジオ ボタンの設計



ラジオ ボタンを設計するには、次のガイドラインに従ってください。

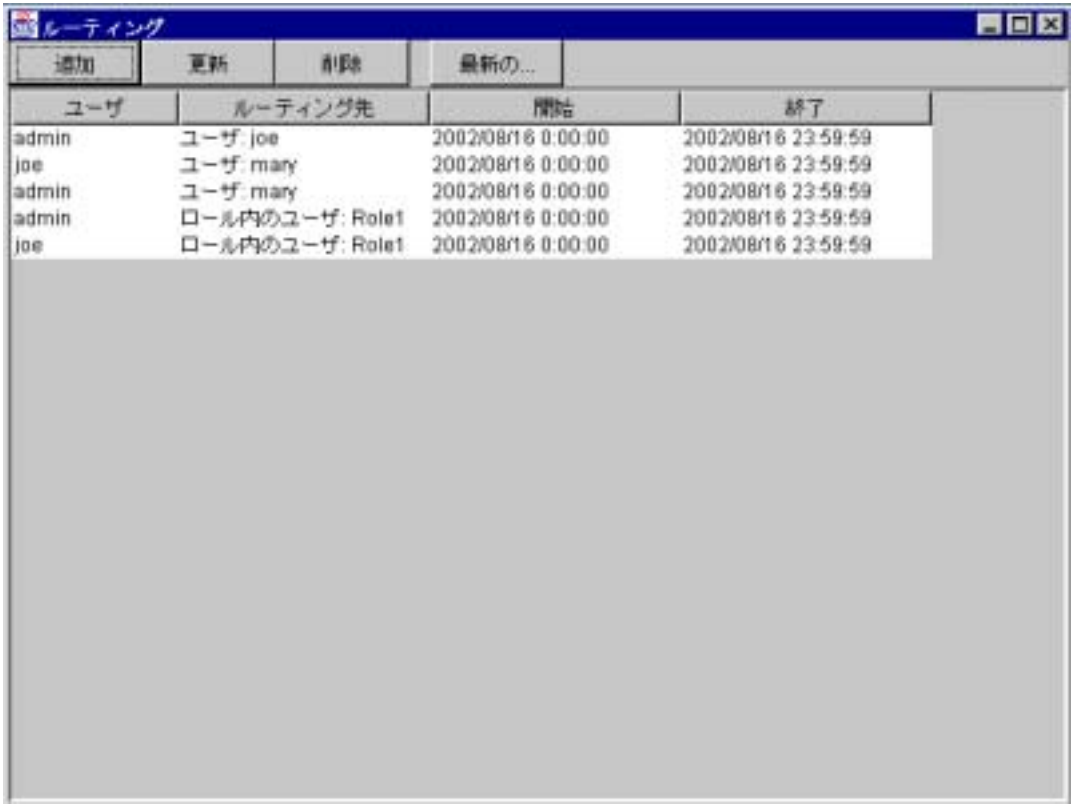
- ラジオ ボタンは、複数の項目から 1 つのみをユーザに選択させる必要がある場合に使用します。
- 各ラジオ ボタンに対して分かりやすいラベルを付けます。

- グループ ボックスを使用して、ラジオ ボックスをまとめて表示します。グループ ボックスには、分かりやいラベルを付けます。
- ラジオ ボタンは縦に並べます。
- ラジオ ボタンの数は 6 個以下にします。

テーブル

テーブルにより、同時にかなりの量の情報を入力したり表示したりできます。次の BPM の例で、適切なテーブルの設計を示します。

図 C-5　テーブルの設計



テーブルを設計するには、次のガイドラインに従ってください。

- ユーザがデータの複数の部分を比較する必要がある場合は、テーブルを使用します。
- データの内容を正確に表すカラム ラベルを選択します。
- カラム ラベルは、すべて左詰めで表示します。カラム ラベルの後にコロンを使用しないでください。

テキスト入力フィールド

テキスト入力フィールドは、データ入力時に最もよく使用される対話式コンポーネントです。次の BPM の例で、適切なテキスト入力フィールドの設計を示します。

図 C-6　テキスト入力フィールドの設計

テキスト入力フィールドを設計するには、次のガイドラインに従ってください。

- ユーザがデータの入力や編集を行えることを示すために、枠付きのテキストボックスを使用します。たとえば、前の図では、[名前] というラベルのボックスがテキスト ボックスです。
- 適切なデータの長さを表すようなフィールド長にします。

- マージンが不統一にならないように、フィールドの左端を合わせて配置します。
- 類似した情報に関係するフィールドはグループ ボックスにまとめます。グループには、分かりやいラベルを付けます。
- 各フィールドに対して分かりやすいラベルを付けます。ラベルはフィールドの左に置き、左揃えにします。

プレゼンテーション コンポーネントの操作

プレゼンテーション コンポーネントは、ダイアログ ボックスにデータがどのように表示されるかを制御します。プラグインを設計するときには、ユーザがデータを使用して何を行う必要があるかを考慮してください。たとえば、いくつかの情報を比較したり、特定の基準に基づいて選択を行ったりする必要があるかどうかを検討します。情報を適切に表示できるかどうかにより、ユーザがアプリケーションの有用性をどのように感じるかに大きな影響を与えることがあります。

次の節では、プレゼンテーション コンポーネントを設計するためのガイドラインを示します。

- カラー
- ダイアログ ボックスのレイアウト
- フォント
- アイコン
- メッセージ
- 視覚的バランス

カラー

カラーは、ユーザの注意を引くための重要な要素です。カラーを上手に使用し、ダイアログ ボックスの使い勝手を向上させることができます。

カラーを決定する際には、次のガイドラインに従ってください。

- 見た目の美しさ以外にも、カラーを使用する適切な理由を検討します。たとえば、カラーを使用して、サーバの状態が起動中（緑）であるか停止中（赤）であるかを示すことができます。
- 彩度の低い、純色でない色合いを選択します。純色は、画面で明るすぎることが多く、目を疲れさせます。
- 色覚障害のユーザに配慮します。色覚障害では、赤 / 緑と青 / 黄は茶色に見えます。したがって、重要な視覚的サインの提供をカラーだけに頼らないでください。
- 対象ユーザの文化における色の意味を考慮します。たとえば、米国では、赤は熱い、危険、停止を意味します。一方、中国では、赤は喜びやお祝いを意味します。
- 似た色、すなわち、緑と青、紫と青のような色環で隣り合うカラー、あるいは灰色と黒のような中性色を使用します。くすんだ色合いは他のどのカラーとも良く調和します。

ダイアログ ボックスのレイアウト

ダイアログ ボックスのレイアウトは、アプリケーションの使いやすさを左右する重要な要素です。レイアウトは、顧客がアプリケーションを使いやすいと感じるかかどうかに影響を与えることがあります。

ダイアログ ボックスのレイアウトを作成するには、次のガイドラインに従ってください。

- ダイアログ ボックスは、ユーザのワークフローに合うように構成します。たとえば、プラグイン機能をサポートする場合、どのようなダイアログ ボックスが必要であるか、どのような順序で表示するか、などを検討します。
- ダイアログ ボックスに複数のタスクのためのコントロールを詰め込まないようにします。各ダイアログ ボックスは、ユーザのワークフローの 1 つのタスクを表すようにしてください。
- 情報の流れは、横または縦のいずれか一方を選択します。各ダイアログ ボックスで同じ流れを使用する必要はありません。ただし、同じダイアログ ボックスに横の流れと縦の流れを混在させないでください。

横の流れにすると、一般に、ダイアログ ボックスは横に長くなります。この場合、ユーザは、左上すみから情報の処理を開始し、左から右へ移動します。

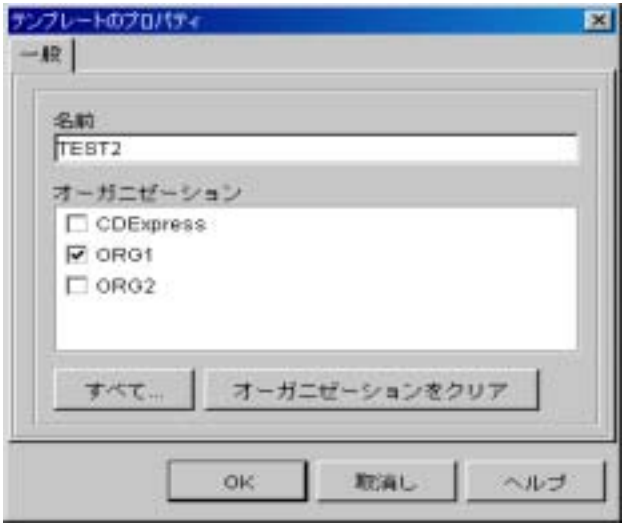
縦の流れにすると、一般に、ダイアログ ボックスは縦に長くなります。この場合、ユーザは、左上すみから情報の処理を開始し、上から下へ移動します。次の BPM の例で、縦の情報の流れを示します。

図 C-7　縦の流れ

フォント

フォントを適切に選択すると、ダイアログ ボックスのコントロール ラベルが読みやすくなります。次の BPM の例で、適切なフォントの選択を示します。

図 C-8　フォントの選択



フォントを選択するには、次のガイドラインに従ってください。

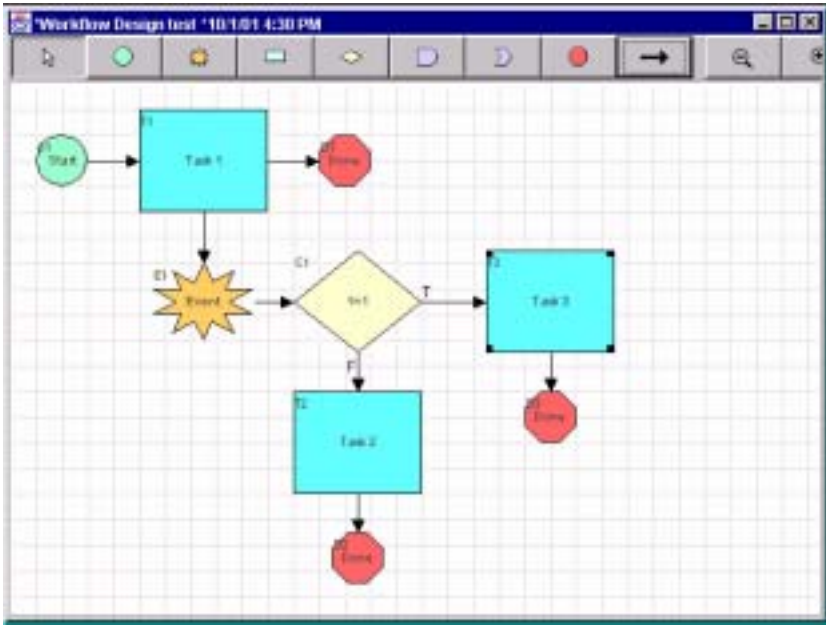
- すべてのコントロール ラベルに 12 ポイント Arial のレギュラー フォントを使用します。
- テキストでのカラー フォントの使用は避けます。もっとも読みやすいテキストは、白地に黒文字です。
- 強調目的での斜体や下線の使用は避けます。

アイコン

アイコンは、プラグイン設計で重要な役割を果たします。プラグインがロードされると、インタフェース ビューの開始ノードとイベント ノードの右上すみに、ワークフローがプラグイン定義されたトリガを持っていることを示す 16 × 16 ピクセルのアイコンが表示されます。

アイコンは、インタフェース ビューでプラグイン アクションをタスクに割り当てるときにも表示されます。この場合、プラグイン アクションのために用意されているデフォルトのアイコンを使用することも、独自の 32 × 32 ピクセル アイコンを作成することもできます。次の BPM の例に、Inventory という名前のワークフローで、[開始]、[タスク]、[イベント]、[分岐]、[完了] 表す各アイコンを示します。

図 C-9　[開始]、[タスク]、[イベント]、[分岐]、[完了] を表す各アイコン



独自の 32 × 32 ピクセル プラグイン アクション アイコンを作成するには、次のガイドラインに従ってください。

- 識別のために必要な最低限の情報のみを含めます。写真のように見えるアイコンは避けてください。必要以上に詳細なアイコンは、理解しにくくなります。
- 同じコンセプトを表すには 1 つのアイコンのみを使用します。たとえば、インタフェース ビューの別の部分で、同じプラグイン アクションに対して異なるプレゼンテーションを使用しないでください。

- 意味が明確でないアイコンは作り直します。視覚的イメージを強化するには、ツール ヒントを追加します。
- 国際的に通用するアイコンを設計します。設計の焦点をユニバーサルに受け入れられるプレゼンテーションに置いてください。不快なジェスチャーを使用しないでください。たとえば、文化によっては、指で指すことは不快なこととみなされます。

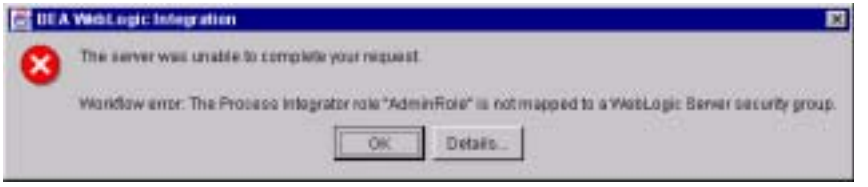
メッセージ

アプリケーション メッセージは、さまざまなコンテキストで表示され、重大度により分類されます。次に例を示します。

- 通知メッセージは、決定に役立つヒントをユーザに提供します。
- 警告は、失敗につながる可能性のある状況をユーザに警告します。
- エラー メッセージは、1 つまたは複数のコンポーネントが失敗したことを示します。

次の BPM の例で、適切なメッセージの設計を示します。

図 C-10　エラー メッセージ



メッセージ テキストを書く場合、ユーザがどのような対応を取るかを予測し、それを理解することが重要です。ユーザにとって意味のあるメッセージを作成するには、次のガイドラインに従ってください。

- ユーザがよく知っている用語を使用します。暗号のようなテキストや理解しにくいテキストは避けます。
- 簡潔で明確なメッセージを作成します。テキストは 2 ～ 3 文の短いものにします。

- エラー メッセージは、2 つの部分に分けます。最初の部分では、どのようなエラーが発生したかをユーザに伝えます。2 番目の部分では、エラーの解決方法をユーザに示します。

視覚的バランス

バランスとは、水平、垂直、対角線の軸に対して、設計の中で要素の比重がどのように構成されているかを表します。軸のどちらの側にあるコントロールも、サイズ、カラー、類似性、配置などの点で、互いにバランスよく見えるようにします。

ダイアログ ボックスにコントロールをバランスよく配置するには、次のガイドラインに従ってください。

- コマンド ボタンは、他のコントロールとバランスが取れるようにダイアログ ボックスに配置します。

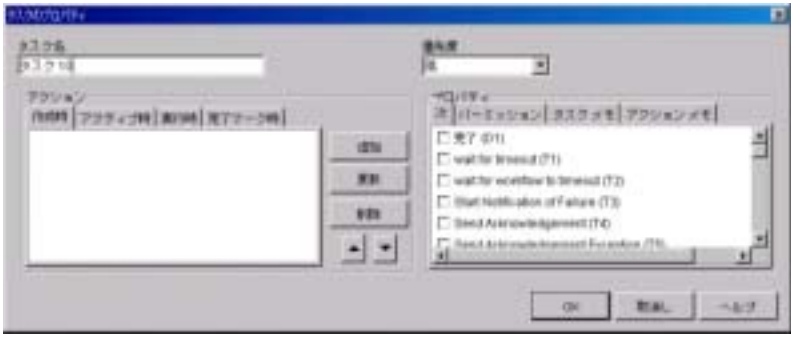
  情報の流れが横である場合、コマンド ボタンを縦に並べ、ダイアログ ボックスの右側に置きます。ボタンをダイアログ ボックスの上部、中央、下部にボタンを移動しながら、左側のもっと大きなコントロールや暗いコントロールとのバランスを確認します。ボタンと他のコントロールが垂直軸に対して対称になるように調整を続けてください。

  情報の流れが縦である場合、コマンド ボタンを横に並べ、ダイアログ ボックスの下部に置きます。この場合、ボタンをダイアログの左端または右端に移動しながら、上部のもっと大きなコントロールや暗いコントロールとのバランスを確認します。ボタンと他のコントロールが水平軸に対して対称になるように調整を続けてください。

- ダイアログ ボックスのサイズは、できるだけ黄金比（1:1.6）になるようにします。この比率は、特に芸術、建築、数学（フィボナッチ数列）で使用されてきたもので、すべての図形の中で最も視覚的満足感のあるものの 1 つです。

- 情報の流れが横のダイアログ ボックスでは、2 列に並べたレイアウトを使用します。それぞれの列で、各コントロールの上と下のラインができるだけ揃うように、バランスよくコントロールを配置します。次の BPM の例で、2 列に構成した横の情報の流れを示します。

図 C-11　横のレイアウト



推奨参考文献

直感的で使いやすいインタフェース設計の詳細については、次の文献を参考にしてください。

- 『*About Face: The Essentials of User Interface Design*』（Alan Cooper、1995 年）
- 『*GUI Bloopers: Don’ts and Do’s for Software Developers and Web Designers*』（Jeff Johnson、2000 年）
- 『*The Elements of User Interface Design*』（Theo Mandel、1997 年）
- 『*The Usability Engineering Lifecycle: A Practioner’s Handbook for User Interface Design*』（Deborah Mayhew、1999 年）
- 『*Usability Engineering*』（Jakob Nielsen、1994 年）
- 『*User and Task Analysis for Interface Design*』（Joann Hackos and Janice Redish、1998 年）

索引

H

I

J

L

O

P

R

S

T

U

V

X

あ

い

お

か

き

く

け

こ

さ

し

へ

ほ

め

ら

り

れ

ろ

わ