Magic xpi

Connector Builder

Contents

はじめに

本書は Connecter Builder の使用方法および同ツールを使用したコンポーネントの開発手順を記述したものです。Connecter Builder を使用することで、.NET Framework（以下 .NET）または Java によるカスタムコンポーネントの開発が可能です。

次章では、開発環境のセットアップ手順を記述しています。「カスタムコンポーネントの作成」章では、コンポーネントの開発手順を説明し、「-Directory Scanner- の実装」章では、実際にサンプルコンポーネントを実装するシナリオを解説しています。

開発環境

コンポーネントの開発には、.NET および Java の開発環境が必要です。本書では、統合開発環境「Visual Studio Express 2015 for Desktop」および「Eclipse (Pleiades All in One)」を使用します。

以下の手順に従って、セットアップを行ってください。

Visual Studio Express 2015 for Desktop のセットアップ

1. Visual Studio Express 2015 for Desktop のダウンロード

以下のサイトにアクセスし、インストーラをダウンロードしてください。
https://www.visualstudio.com/ja/vs/visual-studio-express/

ページをスクロールして「Express for Desktop」の「ダウンロード」をクリックし、ダウンロードを開始してください。

図 1：Visual Studio Express ダウンロードページ

図 2：Express for Desktop のダウンロード

2. Visual Studio Express 2015 for Desktop のインストール

ダウンロードが完了したら、インストーラ「wdexpress__***.exe」を実行し、Visual Studio Express 2015 for Desktop のインストールを開始してください。

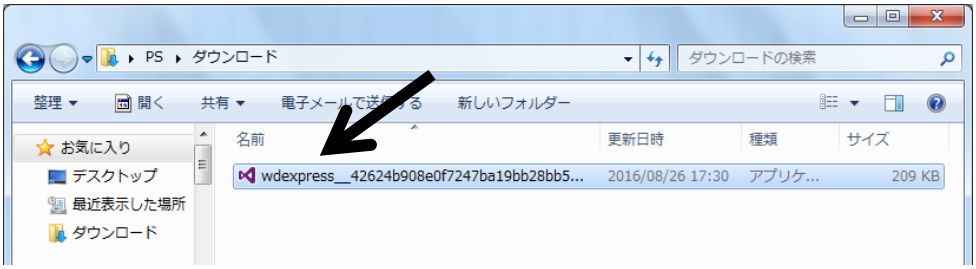


図 3：Visual Studio Express 2015 for Desktop：インストーラの実行

インストールを開始すると下記の画面が表示されます。インストール先を指定し「次へ(N)」をクリックします。

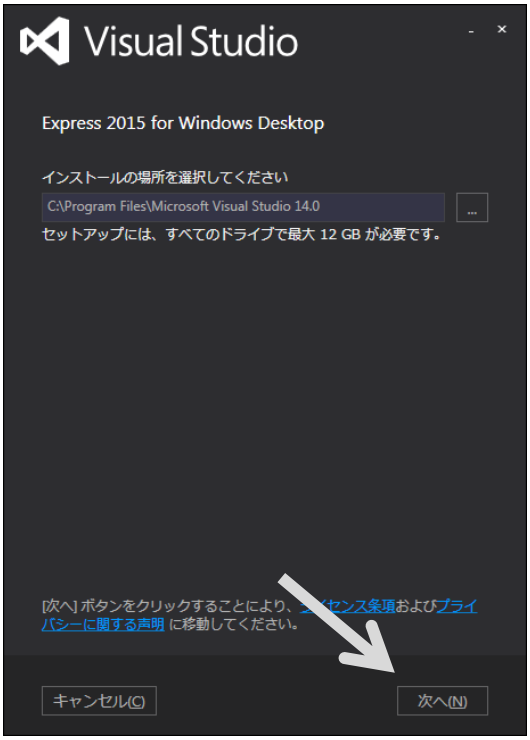


図 4：Visual Studio Express 2015 for Desktop：インストール先の指定

続いて、下記の画面が表示されることを確認し、「インストール(I)」をクリックします。ユーザアカウント制御のダイアログが出た場合は「はい(Y)」をクリックし、インストールを続行します。

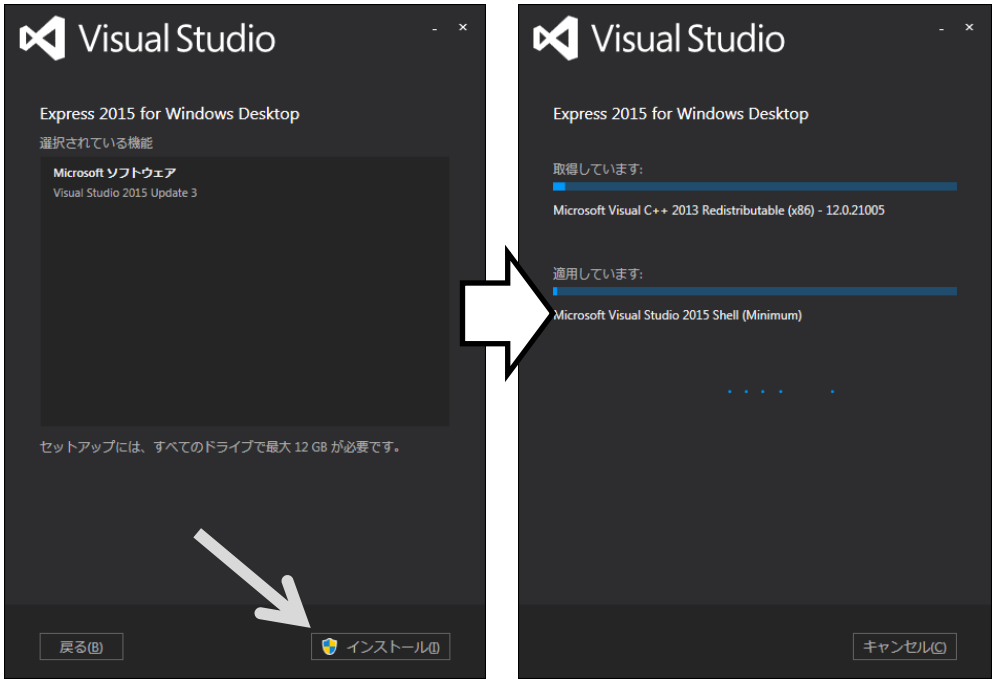


図 5：Visual Studio Express 2015 for Desktop：インストール開始

インストール完了後にコンピュータの再起動を求められた場合は「今すぐ再起動(N)」をクリックし、マシンを再起動します。

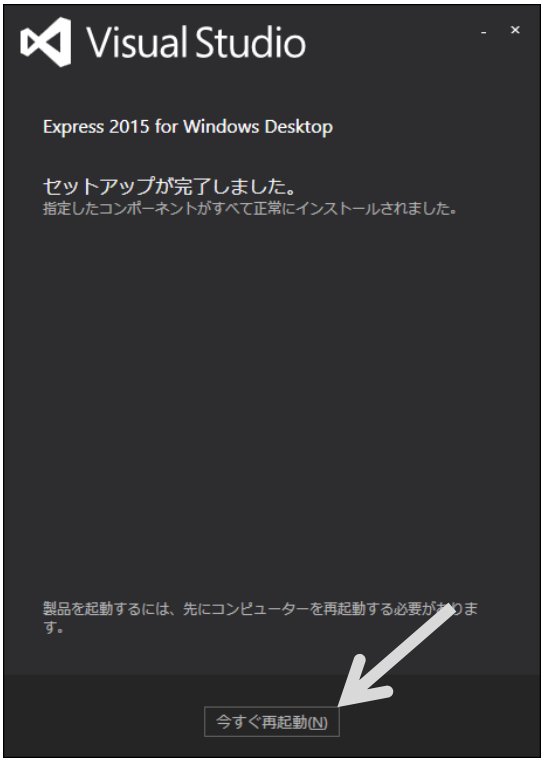


図 6：Visual Studio Express 2015 for Desktop：インストール完了

3. Visual Studio の起動

インストールが完了したら、「起動(L)」ボタンをクリックし、Visual Studio Express 2015 for Desktop を起動します。再起動を行った場合は、スタートメニューから「VS Express for Desktop」を起動します。

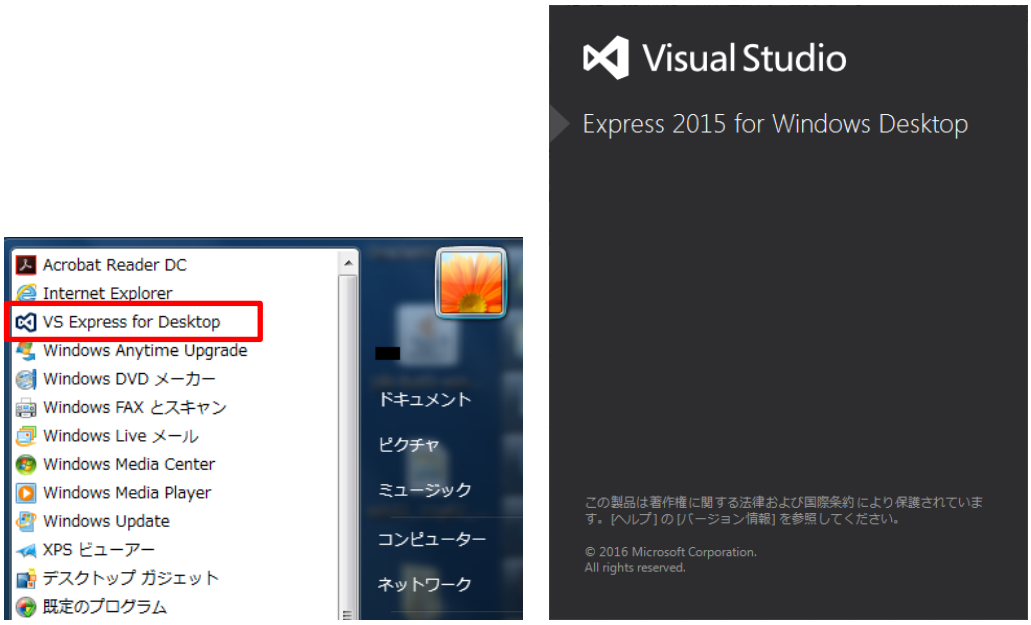


図 7：VS Express for Desktop の起動

VS Express for Desktop を起動すると下記の画面が表示されます。Microsoft アカウントをお持ちの場合は、サインインすることを推奨します。サインインをすることで、VS Express for Desktop の使用期限が無制限となります。サインインをしない場合は、インストール日から 30 日以上の利用ができなくなりますのでご注意ください。

図 8：VS Express for Desktop：サインイン

Eclipse (Pleiades All in One) のセットアップ

1. Pleiades All in One のダウンロード

以下のサイトにアクセスし、Pleiades All in One をダウンロードします。
http://mergedoc.osdn.jp/

「Pleiades All in One ダウンロード」から「Eclipse 4.6 Neon Pleiades All in One」をクリックします。

図 9：Pleiades All in One ダウンロードページ

ご使用している OS に合わせて 32bit/64bit のいずれかを選択し、「Java」の「Full Edition」をクリックして、ダウンロードを開始してください。

図 10：Eclipse 4.6 Neon Pleiades All in One のダウンロード

2. ZIP ファイルの展開および Eclipse の起動

ダウンロードが完了したら、ZIP ファイル「pleiades-e4.6-java-XXbit-jre_YYMMDD.zip」を右クリックし、コンテキストメニューから「すべて展開(T)」をクリックします。

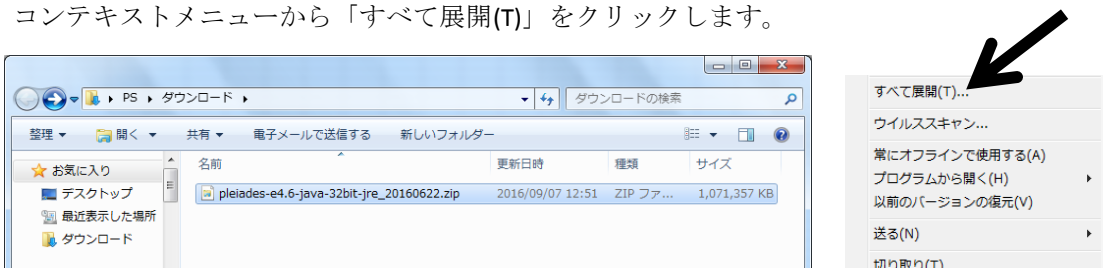


図 11：Eclipse 4.6 Neon Pleiades All in One：ZIP ファイルの展開

「圧縮（ZIP 形式）フォルダの展開」ウィンドウが表示されるので、ご使用している環境に合わせて展開先を指定してください。今回は C ドライブ直下「C:\」を指定します。

※展開先のパスが長すぎると正常に解凍できない場合があります。詳細は Pleiades All in One ダウンロードページの「Windows 上で zip を解凍するときの注意」をご覧ください。

展開が完了すると、指定したフォルダの下に「pleiades」というフォルダが生成されます。

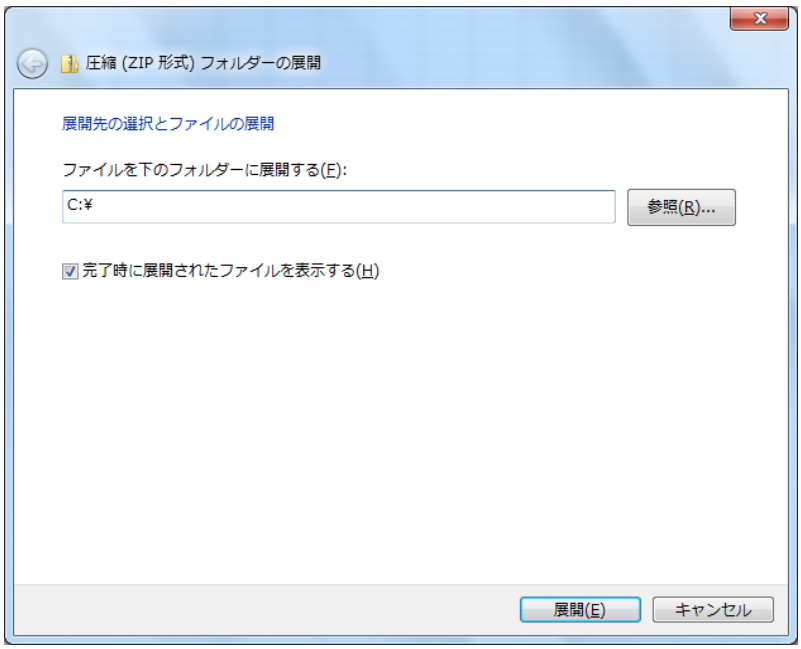


図 12：Eclipse 4.6 Neon Pleiades All in One：ZIP ファイル展開先の指定

次に「pleiades」フォルダの中にある「eclipse」フォルダを開き、「eclipse.exe」を実行します。

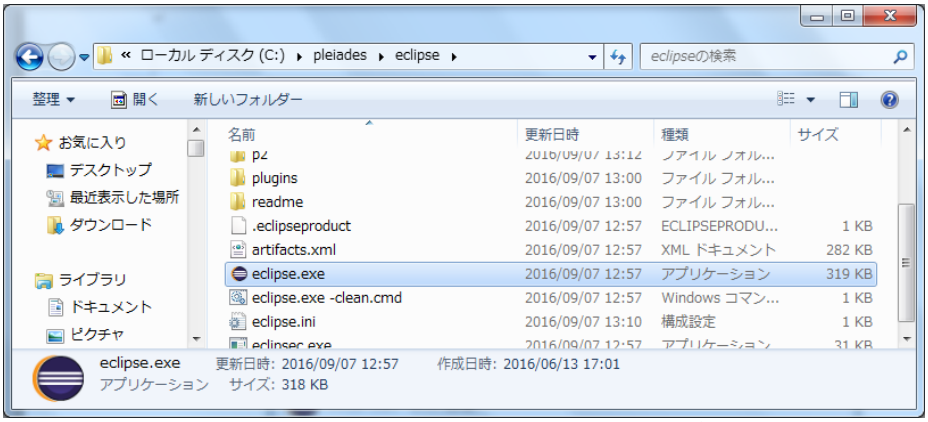


図 13：Eclipse 4.6 Neon Pleiades All in One：展開先フォルダ ”C\pleiades\eclipse”

「eclipse.exe」を実行すると、ワークスペースの選択を求められます。今回はデフォルトのワークスペースを使用するため、そのまま「OK」をクリックします。しばらくすると Eclipse が起動します。

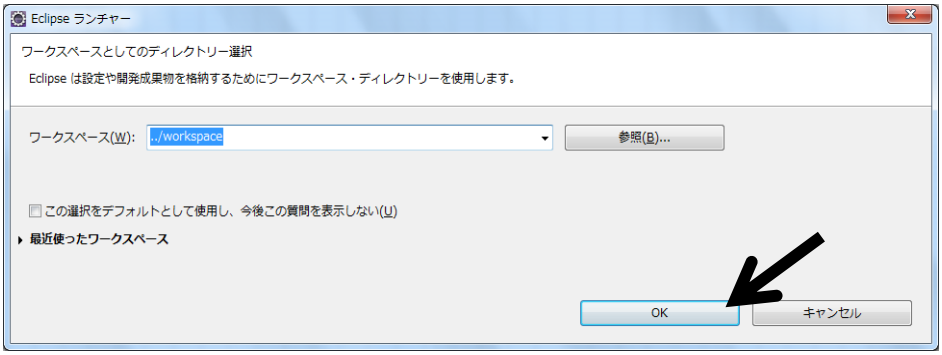


図 14：Eclipse：ワークスペースの選択

図 15：Eclipse の起動

3. Java 開発環境の設定

Eclipse の起動が確認できたら、使用する JRE および JDK の設定を行います。Magic xpi 4.5 は「Java7」を使用しているため、それに合わせて環境設定を変更する必要があります。

Eclipse の「ウィンドウ」メニューから「設定」をクリックします。

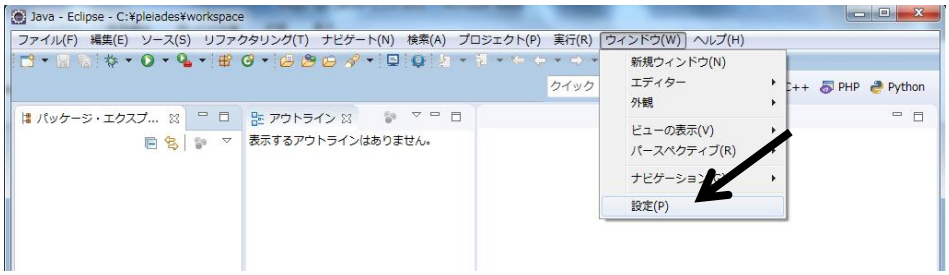


図 16：Eclipse：JRE および JDK の設定

はじめに、使用する JRE の変更を行います。「設定」ウィンドウの左側のメニューから「Java」を展開し、「インストール済みの JRE」を選択します。初期設定では「Java8」が選択されているので、これを「Java7」に変更し「適応」ボタンをクリックします。

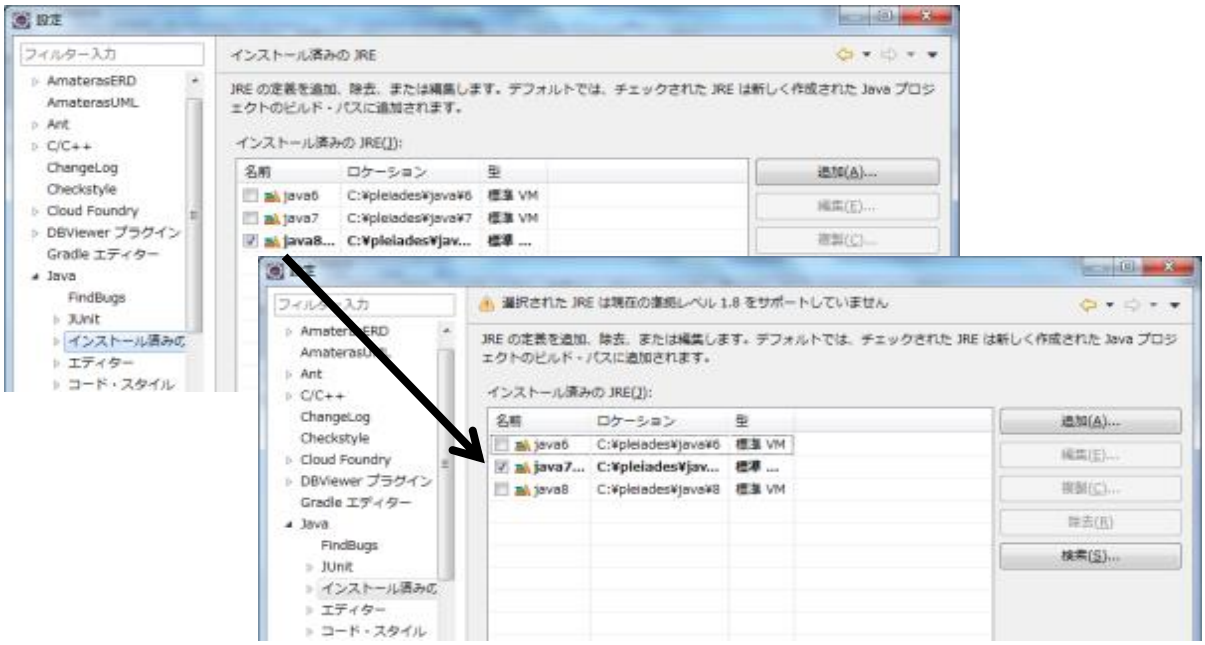


図 17：Eclipse：JRE の変更

次に、使用する JDK の変更を行います。「設定」ウィンドウの左側のメニューから「Java」を展開し、「コンパイラー」を選択します。初期設定では「コンパイラー基準レベル」が「1.8」となっているので、これを「1.7」に変更し「適応」ボタンをクリックします。

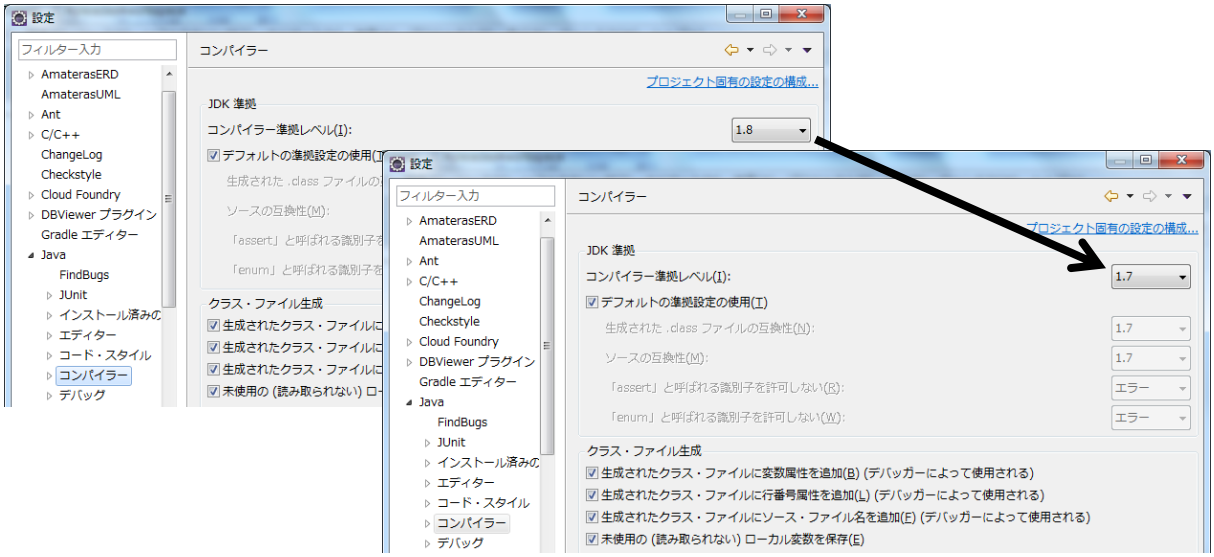


図 18：Eclipse：JDK の変更

カスタムコンポーネントの作成

本章では、Connector Builder を使用したカスタムコンポーネントの開発手順を説明します。カスタムコンポーネントの作成には、大きく分けて以下の３つの工程があります。

- コンポーネントの定義
- ユーザインターフェースの実装
- ランタイムの実装

本書における「ユーザインターフェース（以下、UI と表記）」とは、Magic xpi スタジオでコンポーネントの構成を行う際の設定画面を指します。また「ランタイム」とは、Magic xpi サーバ実行時のコンポーネント動作の実装内容を指します。

コンポーネントの定義

はじめに、Connector Builder からコンポーネントの定義と設定を行います。コンポーネントには、フローエリアに配置する「ステップ」とエディタ上部のトリガーエリアに配置する「トリガー」があります。１つのコンポーネントには、ステップかトリガーのいずれかまたは両方を定義することができます。

コンポーネントの共通設定

スタートメニューまたはデスクトップ上に作成された Magic xpi 4.5 のショートカットから Connector Builder を起動してください。

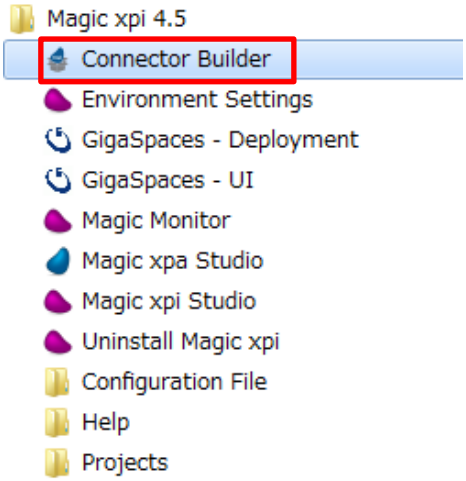


図 19：Connector Builder の起動

「Connector repository」画面が開いたら、画面左下の「Add」ボタンをクリックし、新規コンポーネントを追加します。

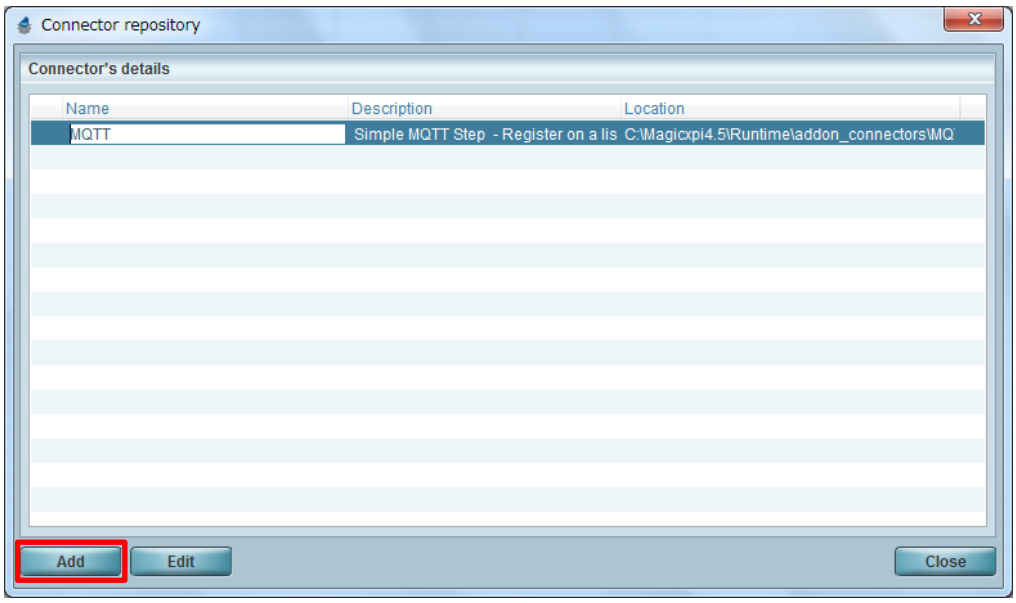


図 20：Connector repository

Connector repository から新規コンポーネントを追加すると、コンポーネントの設定画面が開きます。はじめに、下記の表に従ってコンポーネントの共通設定を行ってください。

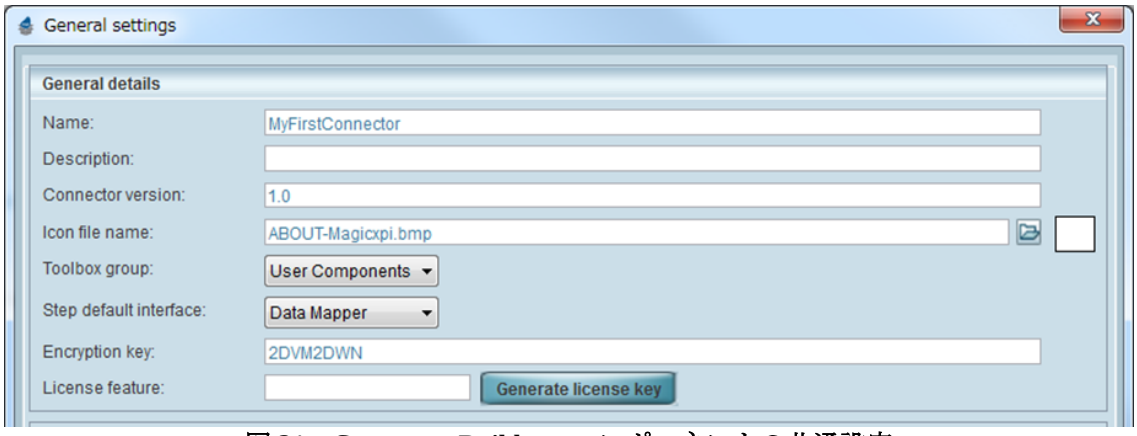


図 21：Connector Builder：コンポーネントの共通設定

設定項目 （* は必須）	説明
General details	
Name*	コンポーネントの名称。 使用できる文字：半角英数字、アンダースコア。 **※全角文字は使用不可（すべての設定項目について同様）。**
Description	コンポーネントの説明。
Connector version*	コンポーネントのバージョン。

Icon file name*	コンポーネントに使用するアイコン。 使用できるファイル：bmp, jpg, gif, png
Toolbox group*	Magic xpi スタジオのツールボックスに表示するグループ。
Step default interface*	コンポーネントをステップとして使用する際のインターフェースタイプ。図 22, 23 を参照。 Data Mapper：マッパー形式を使用する。 Method：メソッド形式を使用する。
Encryption key*	コンポーネントが使用する暗号化キー。
License feature	コンポーネントのライセンス名。 使用できる文字：半角英数字のみ。
Generate license key	暗号化キーと製品のシリアルナンバーから「VENDOR_STRING」を生成します。

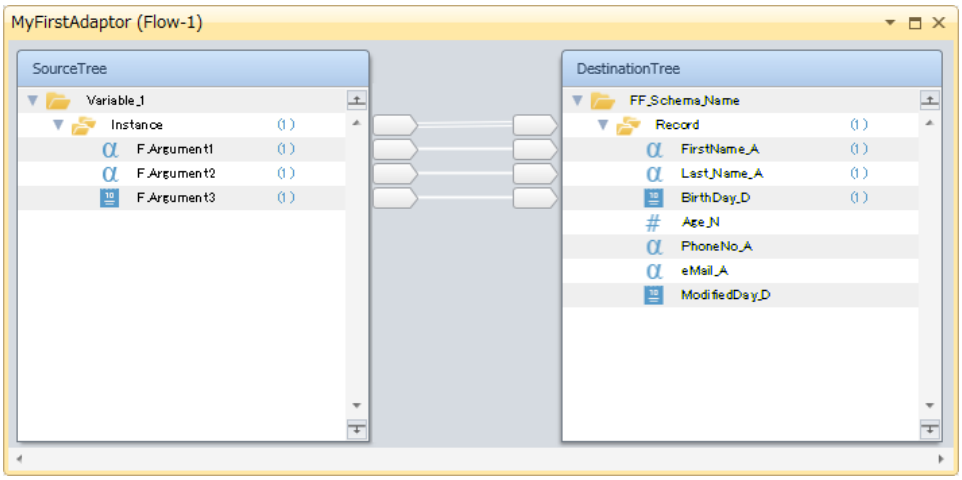


図 22：コンポーネントのインターフェースタイプ：マッパー形式

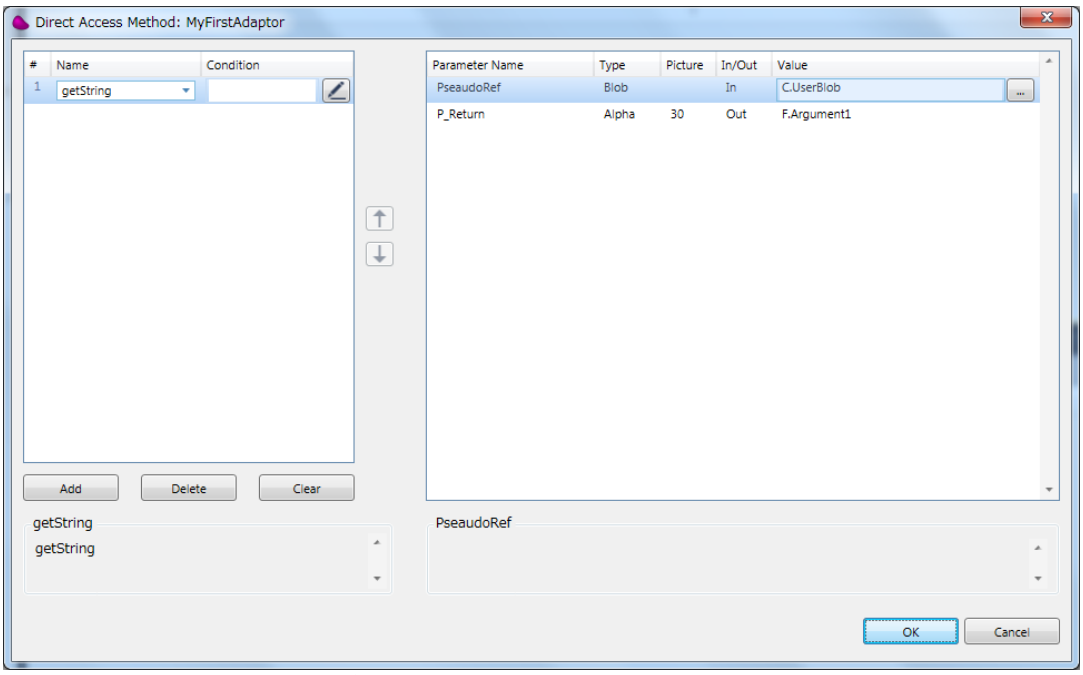


図 23：コンポーネントのインターフェースタイプ：メソッド形式

- **ライセンスの発行**

「Generate license key 」をクリックし、「License key utility」画面を開きます。シリアルナンバーを入力し、「Generate」ボタンをクリックすると、ライセンスの発行に必要な「VENDOR_STRING」が生成されます。

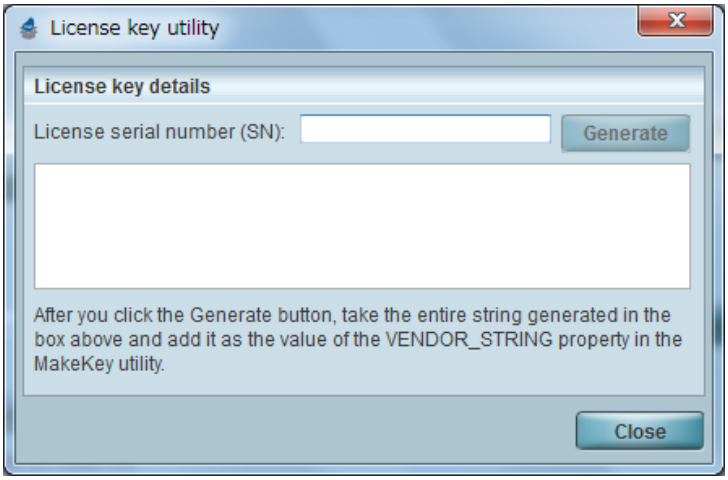


図 24：Connector Builder ：License key utility

次に「<Magic xpi インストールフォルダ>\Runtime\Magic xpa」フォルダを開き、「makekey.exe」を実行します。

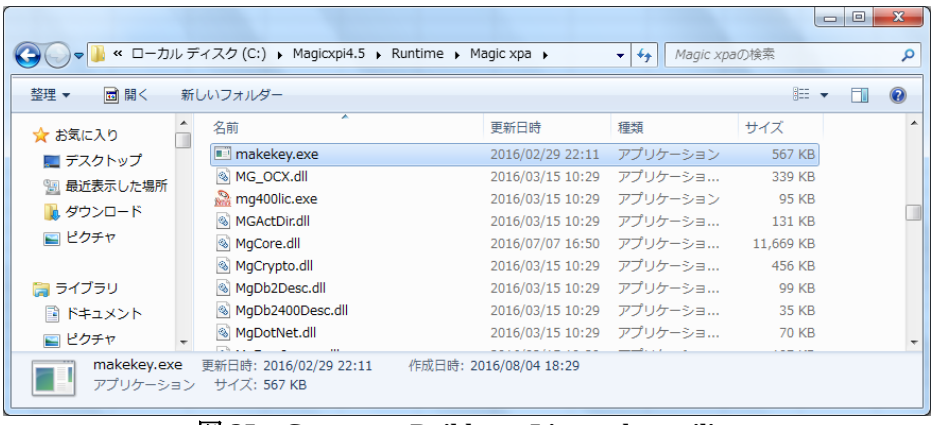


図 25：Connector Builder ：License key utility

「makekey.exe」を実行すると、ライセンス発行用のコマンドラインが立ち上がります。画面の指示に従って必要な情報を入力し、ライセンスを発行してください。詳細は Magic xpi のヘルプから「Home > Reference Guide > Deployment > MakeKey Tool」を参照してください。

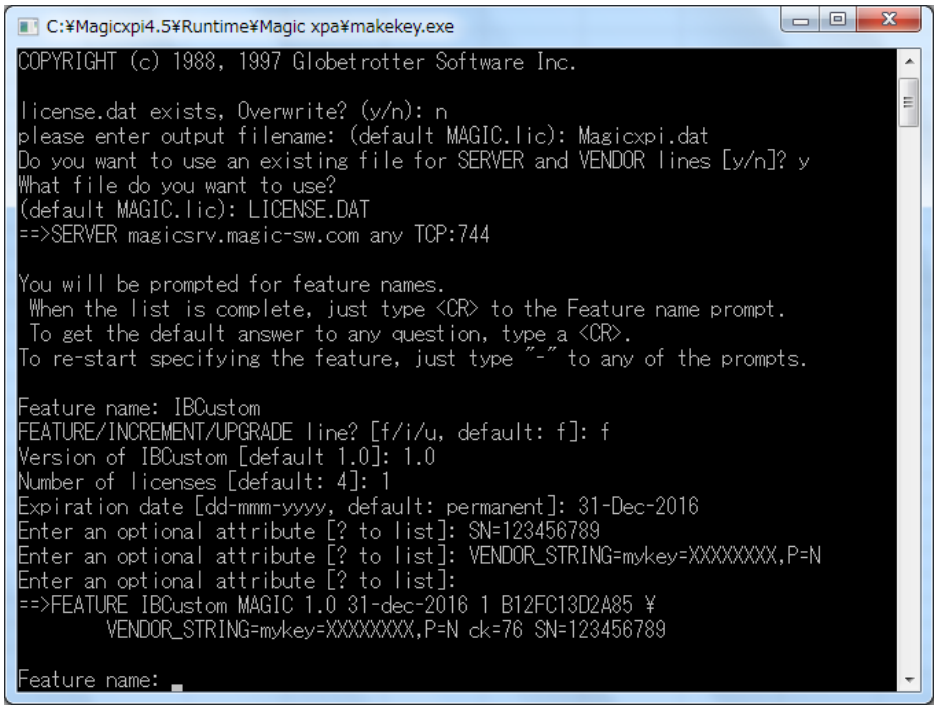


図 26：ライセンスの発行：makekey.exe の実行（完了画面）

最後に、生成したライセンスファイルを開き、発行したライセンス（図 27 赤枠部分）を Magic xpi のライセンスファイルに貼り付けます。

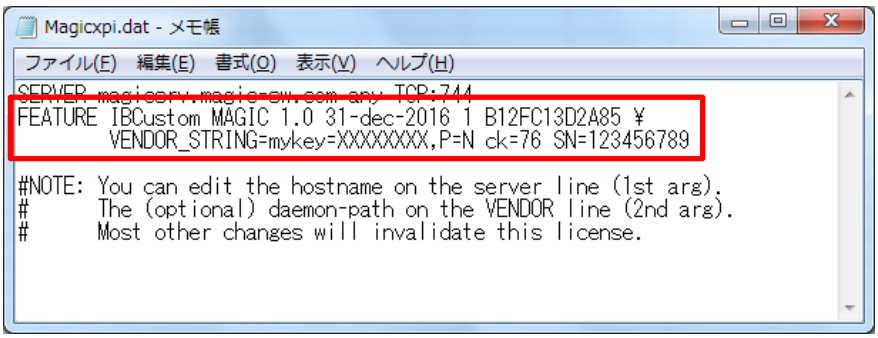


図 27：ライセンスの発行：生成したライセンスファイル "Magicxpi.dat"

ステップの設定

コンポーネントをステップとして使用する場合は「Step」タブを選択し、ステップの設定を行います。下記の表に従って各項目を設定してください。

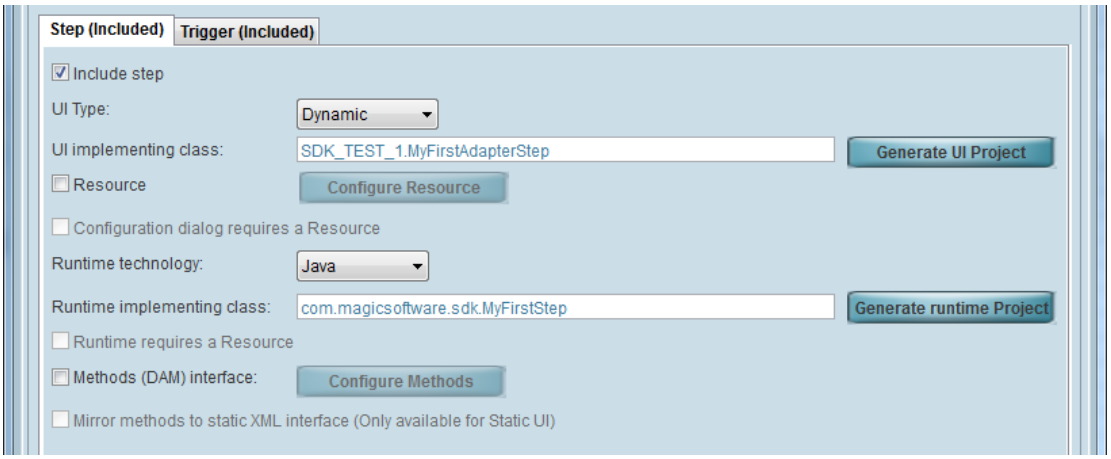


図 28：Connector Builder：ステップの設定

設定項目	説明
Step　※「Include step」を選択した場合のみ有効。	
Include step	コンポーネントをステップとして使用する場合は選択。
UI Type	コンポーネントを実装する際の UI タイプ。 Static：既定の UI を使用する（開発不要）。 Dynamic：開発者が UI を実装する。 **※Step default interface で Method を選択した場合は「Static」、Data Mapper を選択した場合は「Dynamic」を選択。** **※「Dynamic」選択した場合は .NET による UI の実装が必要。**
UI implementing class	UI を実装する際のクラス。.NET でのみ開発可能。 **※UI Type で「Dynamic」を選択した場合は必須。**

Generate UI Project	UI の実装に使用するテンプレートプロジェクトを生成します。 後述「テンプレートプロジェクトの生成」を参照。
Resource	コンポーネントにリソースが必要な場合は選択。
Configure Resource	リソースのプロパティを設定します。 後述「リソースプロパティの設定」を参照。
Configuration dialog requires a Resource	コンポーネントの設定にリソースが必須の場合は選択。
Runtime technology	ランタイムに使用する開発言語。.NET または Java から選択。
Runtime implementing class	ランタイムを実装する際のクラス。 **※UI Type で「Dynamic」を選択した場合は必須。**
Generate runtime Project	ランタイムの実装に使用するテンプレートプロジェクトを生成します。 後述「テンプレートプロジェクトの生成」を参照。
Runtime requires a Resource	コンポーネントの実行にリソースが必須の場合は選択。
Methods (DAM) interface	コンポーネントにメソッドを定義する場合は選択。 **※UI Type で「Static」を選択した場合は必須。**
Configure Methods	コンポーネントメソッドの設定を行います。 後述「メソッドの設定」を参照。
Mirror methods to static XML interface (Only available for Static UI)	インターフェースタイプをメソッド形式（図 23）にした場合に、マッパー形式（図 22）で設定ができるようにする。 **※UI Type で「Static」を選択した場合のみ有効。**

- **テンプレートプロジェクトの生成**

あらかじめ用意されているテンプレートを利用して、開発用のプロジェクトを生成します。

UI implementing class に UI クラスが入力されていることを確認し、「Generate UI Project」をクリックします。プロジェクト名とその格納先を求められるので、「Project name」に任意のプロジェクト名、「Location」に Visual Studio プロジェクトの保存場所を指定し、「OK」ボタンをクリックします。

次に、Runtime implementing class にランタイムクラスが入力されていることを確認し、「Generate runtime Project」をクリックします。同様に、任意のプロジェクト名、Eclipse プロジェクトの保存場所を指定し、「OK」ボタンをクリックします（Runtime technology：Java とした場合）。

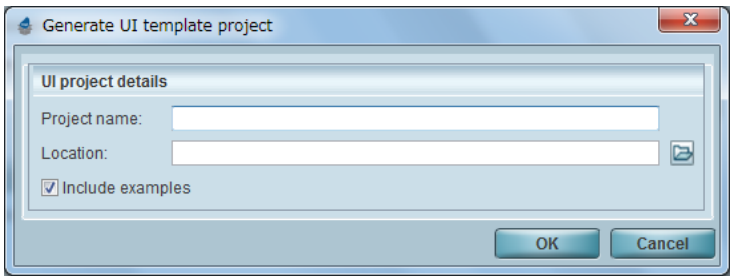


図 29：Connector Builder ：ステップの設定：Generate UI template project

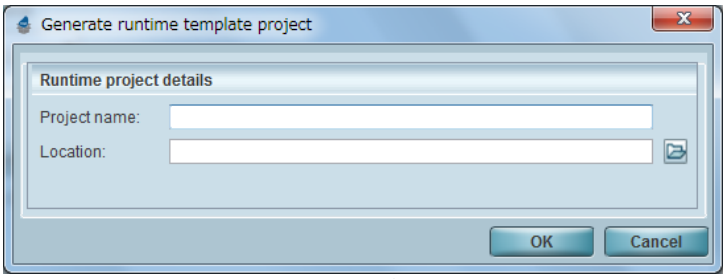


図 30：Connector Builder ：ステップの設定：Generate Runtime template project

- **メソッドの設定　※Step default interface で「Method」を選択した場合のみ**

コンポーネントの共通設定で Step default interface を Method にした場合は、「Configure Methods」からコンポーネントに使用するメソッドを定義します。

「Load methods」画面が開いたら、.NET または Java のいずれかを選択し、DLL または jar ファイルからクラスをロードします。クラス名を入力するとメソッド一覧が表示されるので、使用するメソッドを選択し、「Load」ボタンをクリックします。

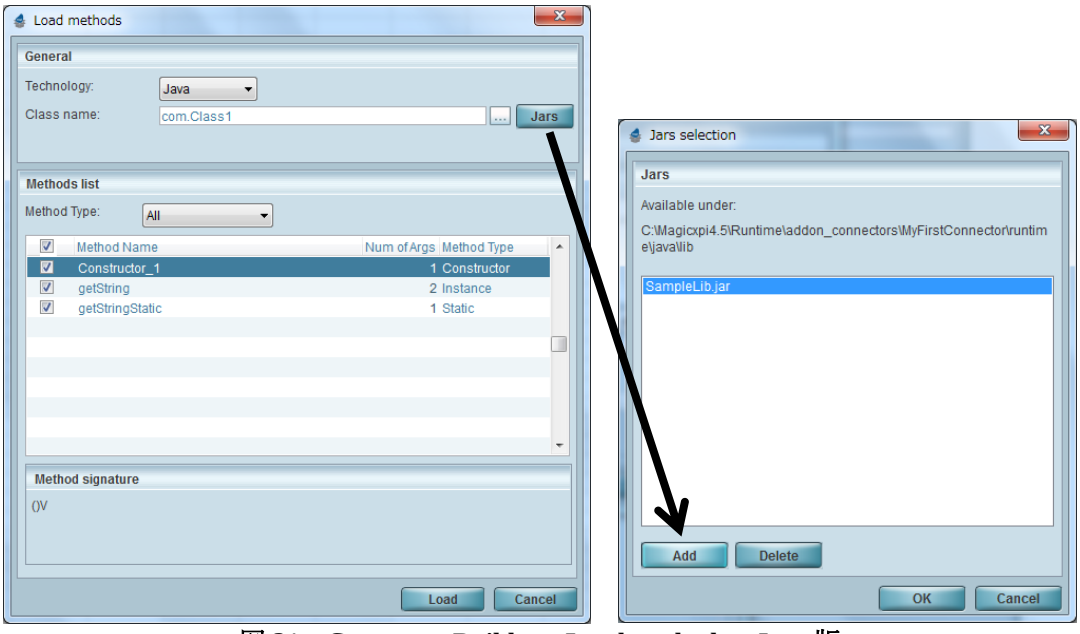


図 31：Connector Builder：Load methods：Java 版

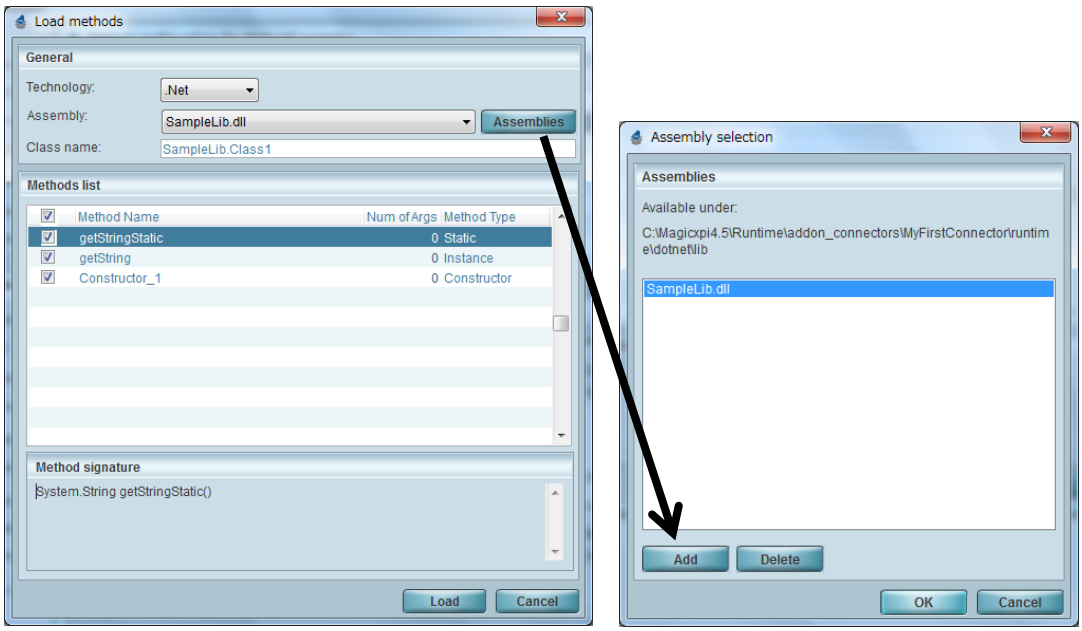


図 32：Connector Builder：Load methods：.NET 版

「Method configuration」画面が開いたら、選択したメソッドがリストに表示されていることを確認します。下記の表を参考に、必要に応じて設定を行います。

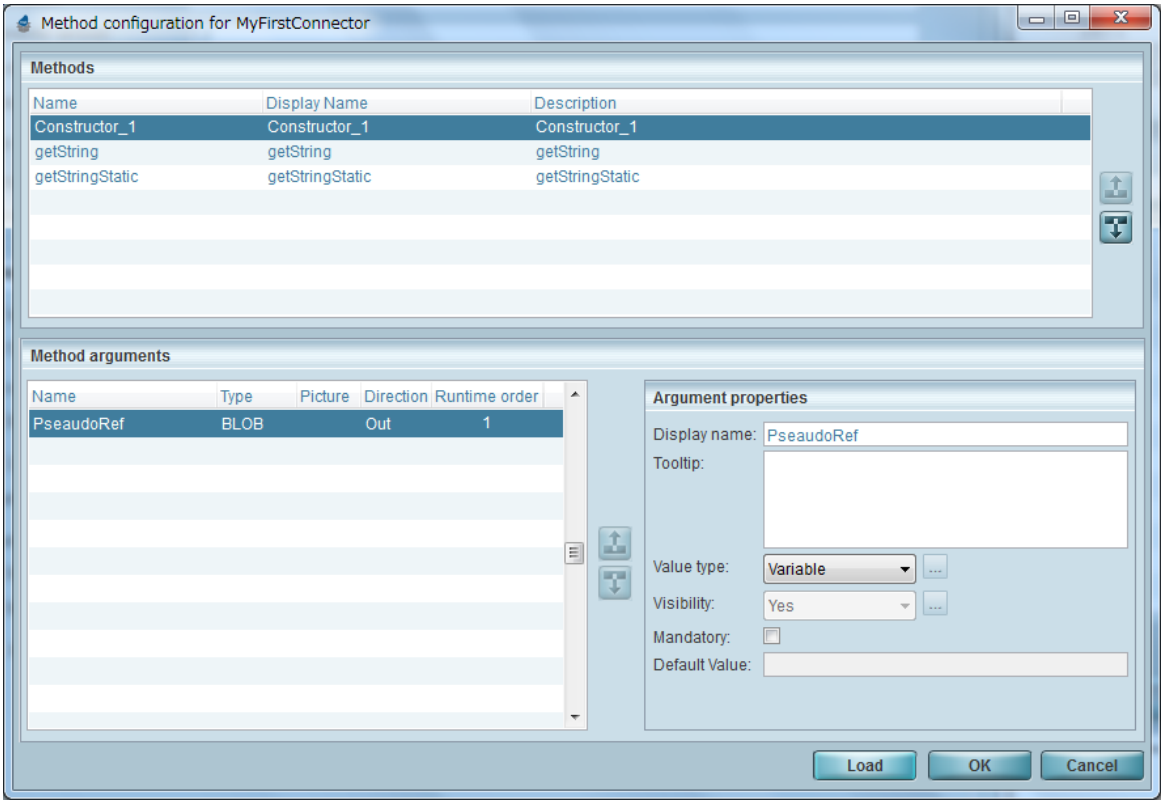


図 33：Connector Builder：Method configuration

設定項目（+は変更可）	説明

Methods	
Name	クラス内で定義されているメソッド名。
Display Name+	UI に表示するメソッド名。
Description+	UI に表示するメソッドの説明。
Method arguments	
Name	メソッド内で定義されているパラメータ名。
Type	パラメータを設定する際のデータ型。
Picture+	パラメータを設定する際の書式。
Direction	パラメータの入出力方向。 In：メソッドへの入力。 Out：メソッドからの出力。
Runtime order	メソッド内で定義されているパラメータの順番。
Argument properties	
Display name+	UI に表示するパラメータ名。
Tooltip+	パラメータのマウスオーバー時に表示するツールチップ。
Value type+	UI 側のパラメータの入力方法。 Variable：変数リストから選択。 Expression：式エディタから入力。 Fixed Value：固定値を指定。**※UI には表示しない。** Combo：セレクトボックスから選択。 Password：テキストボックスから入力し、文字をマスクする。
Visibility+	Yes：パラメータを UI に表示する。 No：パラメータを UI に表示しない。
Mandatory+	パラメータの入力を必須とする場合は選択。
Default Value+	パラメータに設定するデフォルト値。 **※Value type で「Expression」または「Fixed Value」を選択した場合のみ有効。「Fixed Value」を選択した場合は必須。**

- **リソースプロパティの設定（コンポーネントにリソースが必要な場合）**

コンポーネントにリソースを定義する場合は「Resource」を選択し、「Configure Resource」からリソースのプロパティを設定します。

リソースプロパティの設定画面では、「リソース項目」と「アクションボタン」を追加できます。リソース項目には、各種リソースに接続するために必要な設定項目を定義します。また、アクションボタンを追加することで、リソースの設定に必要な動作を実装することができます（後述「UI の実装 > UI クラスの実装」を参照）。下記の表に従って各項目を設定してください。

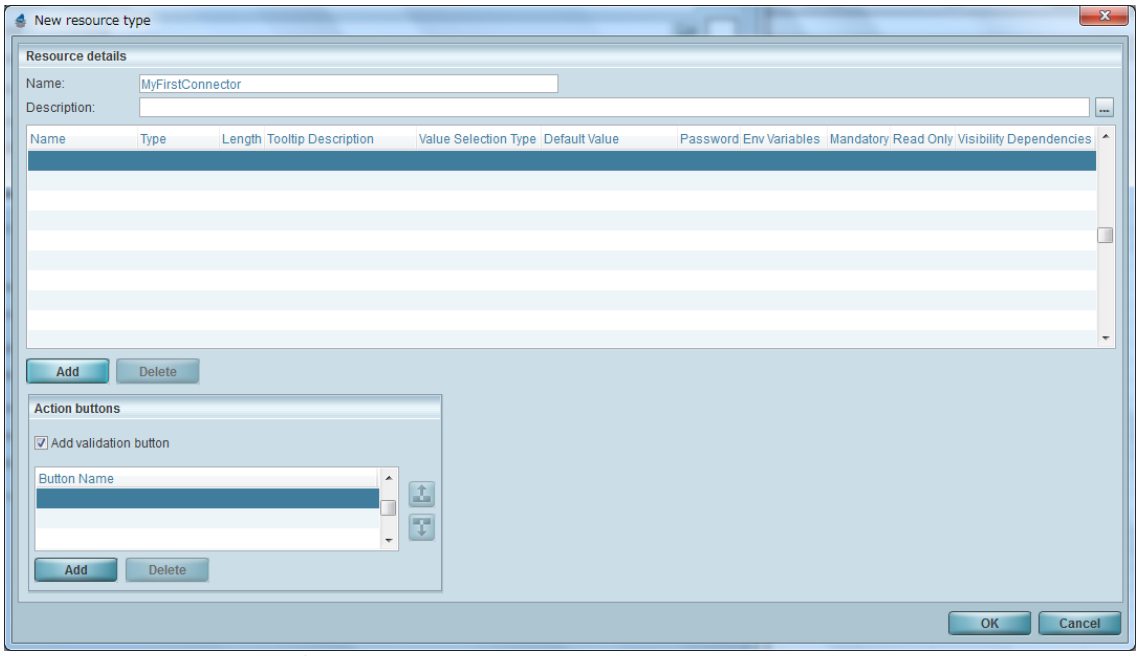


図 34：Connector Builder：New resource type

設定項目（* は必須）	説明
Resource details	
Name*	リソースの名称。
Description	リソースの説明。
リスト（「Add」ボタンから項目を追加）	
Name*	リソース項目の名称。
Type*	リソース項目のデータ型。 Alpha：文字 Numeric：数値 Logical：論理 Date：日付

	Time：時刻
Length*	リソース項目の書式。
Tooltip Description	リソース項目のマウスオーバー時に表示するツールチップ。
Value Selection Type*	UI 側のリソース項目の入力方法。 Combo：セレクトボックスから選択。 Expression：テキストボックスから入力。
Default Value	リソース項目に設定するデフォルト値。
Password	入力文字をマスクする場合は選択。
Env Variables	リソース項目を環境変数リストから選択可能にする場合は選択。
Mandatory	リソース項目の入力を必須とする場合は選択。
Read Only	テキストボックスの編集を不可にする。 **※環境変数リストからの選択は可能。**
Visibility Dependencies*	Yes：リソース項目を UI に表示する。 No：リソース項目を UI に表示しない。 Condition：他のリソース項目の値に応じて表示／非表示を決定。 **※Value Selection Type で「Combo」が選択されている項目のみ。**
Action buttons（「Add」ボタンから項目を追加）	
Add validation button	リソース設定画面に「検証」ボタンを表示する場合は選択。
Button Name	リソース設定画面に表示するアクションボタンの名称。

トリガーの設定

「Trigger」タブに移動します。コンポーネントをトリガーとして使用する場合は「Include Trigger」を選択し、トリガーの設定を有効にします。下記の表に従って各項目を設定してください。

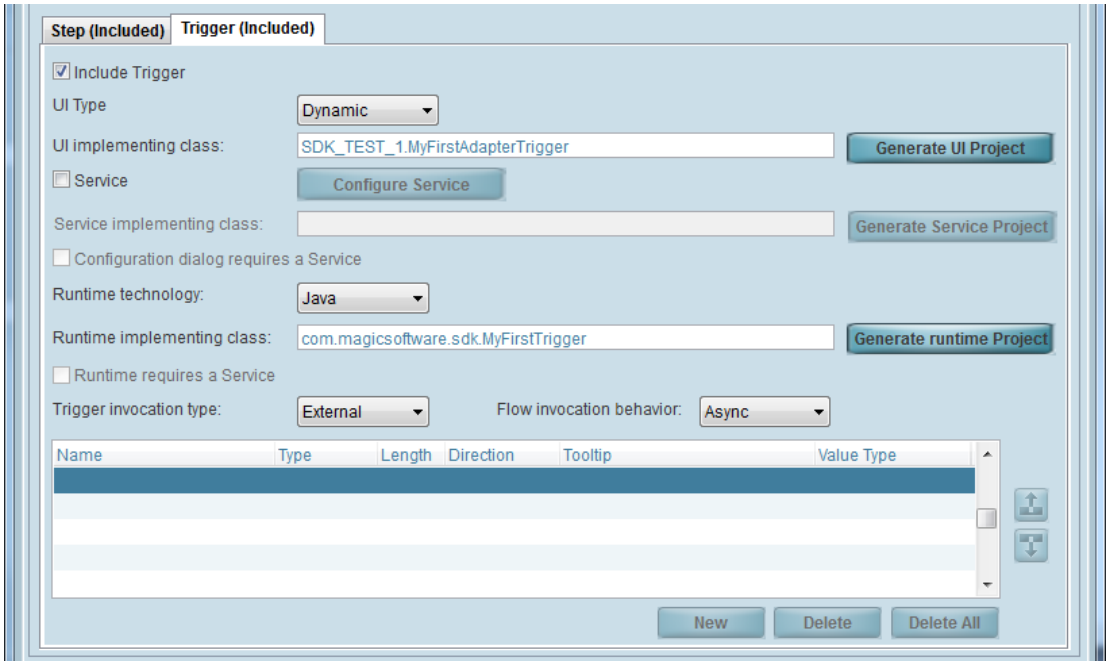


図 35：Connector Builder：トリガーの設定

設定項目	説明
Trigger　※「Include trigger」を選択した場合のみ有効。	
Include trigger	コンポーネントをトリガーとして使用する場合に選択。
UI Type	コンポーネントを実装する際の UI タイプ。 Static：既定の UI を使用する（開発不要）。 Dynamic：開発者が UI を実装する。 **※「Dynamic」選択した場合は .NET による UI の実装が必要。**
UI implementing class	UI を実装する際のクラス。.NET でのみ開発可能。 **※UI Type で「Dynamic」を選択した場合は必須。**
Generate UI Project	UI の実装に使用するテンプレートプロジェクトを生成します。 後述「テンプレートプロジェクトの生成」を参照。
Service	コンポーネントにサービスが必要な場合は選択。
Configure Service	サービスのプロパティを設定します。 後述「サービスプロパティの設定」を参照。
Configuration dialog requires a Service	コンポーネントの設定にサービスが必須の場合は選択。

Runtime technology	ランタイムに使用する開発言語。.NET または Java から選択します。
Runtime implementing class	ランタイムを実装する際のクラス。**※必須。**
Generate runtime Project	ランタイムの実装に使用するテンプレートプロジェクトを生成します。 後述「テンプレートプロジェクトの生成」を参照。
Runtime requires a Service	コンポーネントの実行にサービスが必須の場合は選択。
Trigger invocation type	ランタイムを実装する際のトリガーの動作タイプ。 Polling：監視先に定期的に問い合わせ行い、応答を受け取ったタイミングでフロー呼び出しを開始する。 External：監視先からの入力を受け取ったタイミングでフロー呼び出しを開始する。
Flow invocation behavior	フロー起動後のトリガーの待ち受け動作を選択。 Sync-Wait：トリガーがフローの終了を待ち受ける場合。 ● 現在のフローが終了するまで、次のフローを起動しない。 ● フロー実行中に受け取った呼び出し要求は順次実行する。 Sync-no Wait：トリガーがフローの終了を待ち受ける場合。 ● 現在のフローが終了するまで、次のフローを起動しない。 ● フロー実行中に受け取った呼び出し要求は破棄する。 Async：トリガーがフローの終了を待たない場合。 ● 現在のフローの終了を待たずに、次のフローを起動する（現在のスレッド数が最大インスタンス以下の場合）。 **※Trigger invocation type で「External」を選択した場合のみ有効。**
リスト（「Add」ボタンから項目を追加）※UI Type で「Static」を選択した場合のみ有効。	
Name	パラメータ名。
Type	パラメータを設定する際のデータ型。 Alpha：文字 Numeric：数値 Logical：論理 Date：日付 Time：時刻 BLOB：BLOB

Length	パラメータを設定する際の書式。
Direction	パラメータの入出力方向。 In：トリガーへの入力 Return：トリガーからの出力
Tooltip	パラメータのマウスオーバー時に表示するツールチップ。
Value Type	UI 側のパラメータの入力方法。 Variable：変数リストから選択。 Expression：式エディタから入力。

- **テンプレートプロジェクトの生成**

ステップと同様に、用意されたテンプレートから開発用のプロジェクトを生成します。詳細は「ステップの設定 > テンプレートプロジェクトの生成」を参照してください。

※ステップとトリガーでは、生成されるプロジェクトの内容は異なります。

- **サービスプロパティの設定（コンポーネントにサービスが必要な場合）**

コンポーネントにサービスを定義する場合は「Service」を選択し、「Configure Service」からサービスのプロパティを設定します。

サービスプロパティの設定は、リソースプロパティの設定と同様です。前述「ステップの設定 > リソースプロパティの設定」を参照し、「サービス項目」と「アクションボタン」を追加してください。

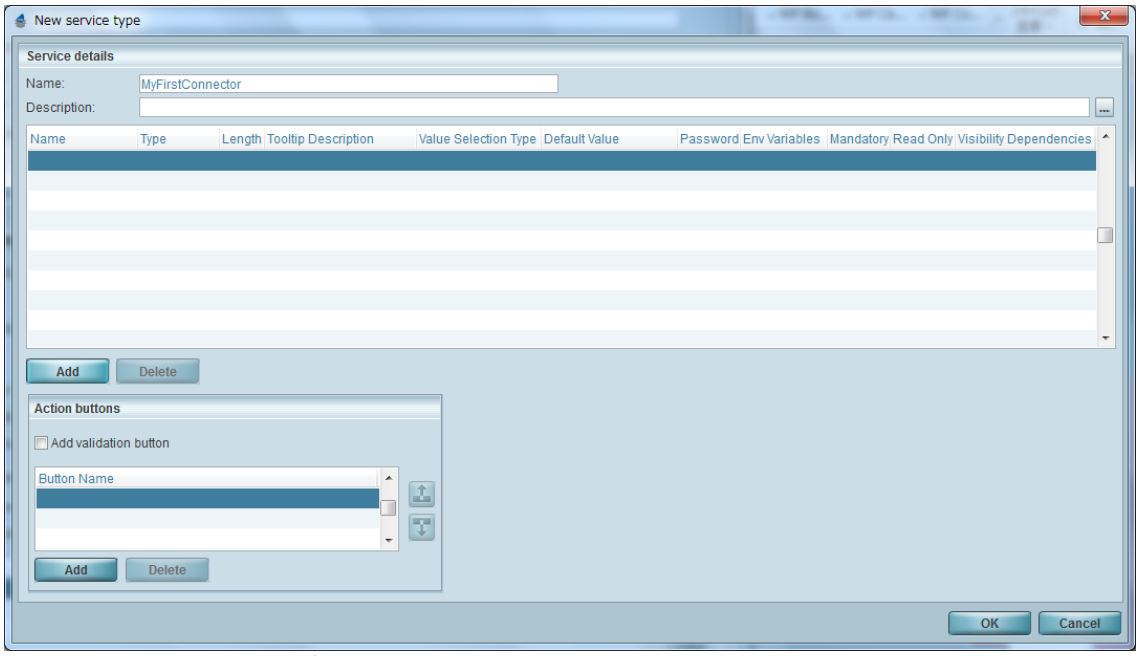


図 36：Connector Builder：New service type

コンポーネントのエラー定義

コンポーネントのランタイムで例外が発生した際のエラーを定義します。

Connector Builder の画面左下の「Errors」ボタンから「Error details」画面を開きます。「New」ボタンから新規行を追加し、エラー内容に従って各項目を設定します。

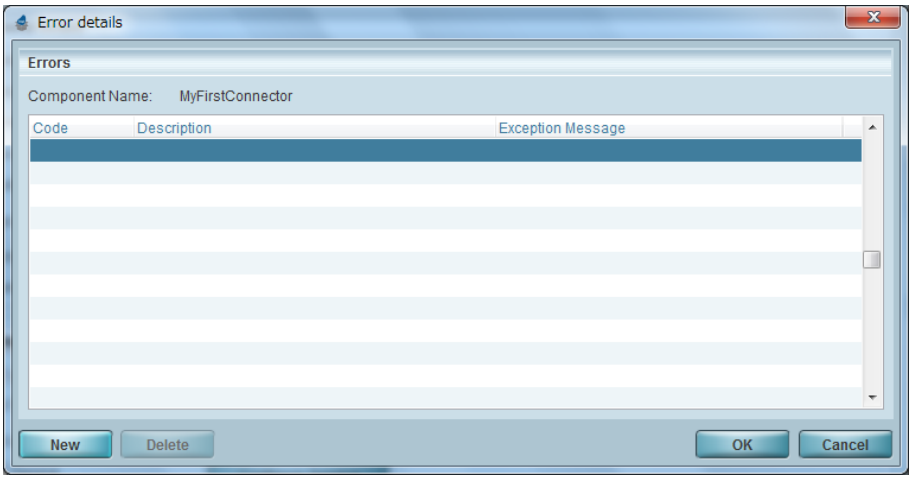


図 37：Connector Builder：Error details

設定項目 （* は必須）	説明
Errors	
Code*	例外発生時のエラーコード。
Description*	エラーリポジトリに表示するエラーの説明。
Exception Message	例外発生時のエラーメッセージ。

コンポーネントの実行中に該当するエラーが発生した場合、Magic xpi モニタのアクティビティログに以下のように表示されます（図 38 を参照）。

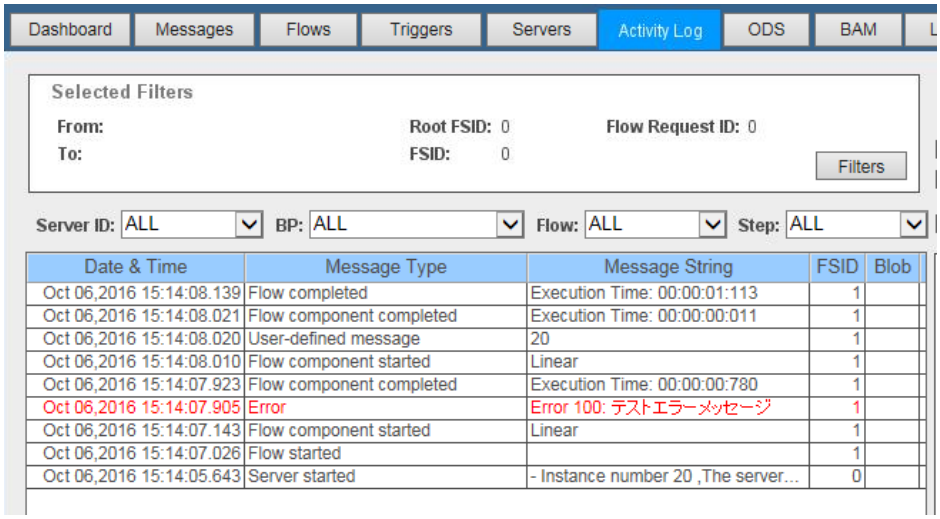


図 38：Magic xpi モニタ：アクティビティログ：エラーメッセージ

設定が完了したら Connector Builder の「OK」ボタンをクリックし、コンポーネントの設定を終了します。作成したコンポーネントは以下のフォルダに保存されています。
<Magic xpi インストールフォルダ>\Runtime\addon_connectors\<作成したコンポーネントの名称>

コンポーネントフォルダの中で主に使用するのは、以下の３つのフォルダです。

- ui\lib　　：UI ビルド時に生成される DLL 等のファイルを配置します。
- runtime\java\lib　　：作成したランタイム（Jar）を配置します。
- runtime\dotnet\lib　　：作成したランタイム（DLL）を配置します。

UI の実装

コンポーネントの設定画面を実装します。ここでは、指定されたクラス、メソッドを実装する必要があります。以下で説明するクラス、メソッドは、テンプレートプロジェクトから削除しないでください。UI は .NET でのみ実装することが可能です。詳細な実装方法に関しては「-Directory Scanner-の実装」を参照してください。

スタートメニューから VS Express for Desktop を起動し、Connector Builder で生成した UI テンプレートプロジェクトを開いてください。

○　データクラスの実装（MyData.cs）

データクラスでは UI の設定項目への入出力を行います。また、Magic xpi の変数または式エディタで設定された値への入出力も可能です。ここでは Magic xpi の基本データ型に対応する以下のクラスを使用し、設定項目のプロパティを定義します。

プロパティクラス一覧

- Alpha　　：「文字」クラス。
- Numeric　　：「数値」クラス。
- Date　　：「日付」クラス。
- Time　　：「時刻」クラス。
- Logical　　：「論理」クラス。
- Variable　　：Magic xpi の「変数」に対応するクラス。
- Expression　　：Magic xpi の「式エディタ」に対応するクラス。

これらのプロパティクラスは、UI の動作に合わせて属性を指定する必要があります。

プロパティクラスの属性

[AllowEmptyExpression]	UI の設定項目が未入力であっても、Magic xpi スタジオのチェッカー実行時にエラーを表示しない。
[PrimitiveDataType(DataType)]	UI の設定項目に入力可能な Magic xpi の基本データ型を指定する（Variable, Expression のみ）。

さらにランタイムにおける入出力の属性を指定する必要があります。これはコンポーネントの動作がステップかトリガーかによって異なります。

ステップ動作時の入出力属性

[In]	UI で設定した値をランタイムに渡す場合。

[Out]	ランタイムの実行結果を UI で設定した変数に保存する場合（Variable のみ）。
[InOut]	In, Out の両方が必要になる場合（Variable のみ）。

トリガー動作時の入出力属性

[TriggerIn]	ランタイムの実行結果を UI で設定した変数に保存する場合（Variable のみ）。
[TriggerReturn]	UI で設定した値を「フローが終了した時点」でランタイムに渡す場合（Variable, Expression のみ）。
[UseForConfigration]	UI で設定した値を「トリガー起動時」にランタイムに渡す場合。

○ UI クラスの実装（<UI クラス名>.cs）

UI クラスでは、コンポーネント構成時の UI およびリソース／サービスの設定に必要な動作の実装を行います。ここでは以下のメソッドを実装する必要があります。

CreateDataObject()

データクラスのインスタンスを返すメソッドです。テンプレートでは既に実装されています。

Configure()

コンポーネントの構成画面が開かれた際に、最初に実行されるメソッドです。このメソッドでは、主に以下の処理を実装します。

- リソースのプロパティを取得する。
- コンポーネントの構成画面を開く。
- 設定内容の変更を判別し、設定変更フラグをセットする。

Configure()メソッドの詳細

引数	
object dataObject	設定された値が保存されているデータオブジェクト。
ISDKStudioUtils utils	ユーティリティクラス。
IReadOnlyResourceConfiguration resourceData	リソースプロパティクラスの読み取り専用オブジェクト。 ※ステップのみ
IreadOnlyServiceConfiguration resourceData	サービスプロパティクラスの読み取り専用オブジェクト。 ※トリガーのみ
object navigateTo	調査中

bool configurationChanged	設定変更フラグ。
戻り値	
Boolean return	True　：設定が正常に終了した場合。 False　：設定が異常終了した場合。

UI からの設定変更が正常に行われた場合は、設定変更フラグ「configurationChanged」および戻り値の値を True にすることで、変更内容が保存され、GetSchema()メソッドが実行されます。このいずれかまたは両方を False にした場合、変更内容は破棄されます。

GetSchema()

Configure()メソッドの設定が正常に行われた際に実行されるメソッドです。このメソッドでは、Data Mapper の送り先情報を SchemaInfo 型で返却する処理を実装します。送り先の形式として、以下の３つの SchemaInfo クラスが用意されています。それぞれの SchemaInfo クラスに応じて、必要な送り先情報を設定します。

SchemaInfo クラス一覧

- XMLSchemaInfo　：XML 形式
- FlatFileSchemaInfo　：フラットファイル形式。
- JsonSchemaInfo　：JSON 形式。

ValidateResource()　※ステップのみ

リソース設定画面から「検証」ボタンをクリックしたときに呼ばれるメソッドです。このメソッドでは、リソースで定義した情報の妥当性をチェックする処理を実装します。このメソッドは Boolean 型を返却します。

- True　：定義した情報が正常な場合。
- False　：定義した情報が不正な場合。引数の errorMsg に設定された文字をエラーダイアログに表示します。

ValidateService()　※トリガーのみ

サービス設定画面から「検証」ボタンをクリックしたときに呼ばれるメソッドです。実装する処理は ValidateResource()と同様です。

InvokeResourceHelper()　※ステップのみ

リソース設定画面に追加したアクションボタンをクリックしたときに呼ばれるメソッドです。引数からボタンの名称および更新可能なリソース項目を取得できます。アクションボタンからダイアログボックスを表示し、接続情報を更新することができます。

InvokeServiceHelper()　※トリガーのみ

サービス設定画面に追加したボタンをクリックしたときに呼ばれるメソッドです。実装する処理は InvokeResourceHelper()と同様です。

Check()

Magic xpi スタジオでチェッカーを実行したときに呼ばれるメソッドです。このメソッドでは、作成したコンポーネントに対して、開発者が独自のチェック項目を追加することができます。例えば、条件必須項目の入力チェックやファイルパスの妥当性を確認することができます。

ランタイムの実装

コンポーネントのランタイムを実装します。ここでは、指定されたクラス、メソッドを実装する必要があります。以下で説明するクラス、メソッドは、テンプレートプロジェクトから削除しないでください。ランタイムは .NET または Java で実装することができますが、実装する内容は同様です。詳細な実装方法に関しては「-Directory Scanner- の実装」を参照してください。

VS Express for Desktop または Eclipse を起動し、Connector Builder で生成したランタイムテンプレートプロジェクトを開いてください。

○　ランタイムクラスの実装（<ランタイムクラス>.cs、<ランタイムクラス>.java）

ランタイムクラスでは、サーバ実行時のコンポーネントの動作を実装します。これはコンポーネントの動作がステップかトリガーかによって実装するメソッドが異なります。

ステップの実装

invoke()

ステップが起動する際に実行されるメソッドです。このメソッドでは、データクラスに保存された設定値や Data Mapper でマッピングされた値を取得し、ランタイムで実行すべき処理を実装します。各領域の設定値を取得するには、引数「StepGeneralParams params」の各メソッドを用いて以下のように取得します。

1. **Data Mapper で設定した値。**
   Data Mapper でマッピングされた値は、送り先情報として設定した SchemaInfo の形式に合わせて、BLOB またはファイル形式でランタイムに渡されます。これらの値を取得するには、以下のメソッドを使用します。

   - BLOB　　：byte[] getPayloadOBject()
   - ファイル　：String getPayloadFile()

   n、取得する形式は、UI クラスの各 SchemaInfo クラスの中で定義されているプロパティで指定することができます。
   SchemaInfo.DataDestinationType = 0 / 1 （0=BLOB, 1=ファイル）

2. **データクラスのプロパティ。**
   データクラスのプロパティを取得するには、以下のメソッドを使用します。

   - プロパティ指定　：UserProperty getUserProperty(“<プロパティ名>”)
   - プロパティすべて：HashMap<String, UserProperty> getUserProperties()

   UserProerty クラスには Magic xpi の基本データ型や値が格納されています。値の取得には、UserProperty の以下のメソッドを使用します。

   Object getValue()

3. **リソースのプロパティ。**
   リソースのプロパティを取得するには、以下のメソッドを使用します。

   HashMap<String, String> getResourceObject()

   取得した HashMap に対して、Connector Builder で定義したリソース項目の名称をキーとし、値を取得することができます。

4. **環境設定の値。**
   サーバ実行時の環境設定の値（プロジェクトフォルダのパス、プロジェクト名、コンポーネントフォルダのパス、etc…）を取得するには、以下のメソッドを使用します。

   HashMap<String, String> getEnviromentSettings()

また、ランタイムの実行結果をデータクラスのプロパティに設定する場合は UserProerty を介して行うことができます。なお、値を設定できるのは [Out] または [InOut] 属性を付与した Variable クラスのプロパティのみです。プロパティのデータ型に合わせて、以下のメソッドを使用します。

- setAlpha(String value)
- setBlob(Byte[] args)
- setData(Date d)
- setTime(String t)
- setLogical(Boolean b)
- setNumeric(Double d)

トリガーの実装

トリガーの実装では、前述「コンポーネントの定義 > トリガーの設定」において、Trigger invocation type で選択したトリガーの動作タイプが「External」か「Polling」かによって、必要なメソッドが異なります。

◆ External トリガー

はじめに、トリガーの動作タイプを External とした場合の実装方法を説明します。

load()

トリガーが起動する際に実行されるメソッドです。このメソッドでは、データクラスに保存された値を取得し、ランタイムで実行すべき処理を実装します。各領域に設定された値を取得するには、引数「TriggerGeneralParams generalParams」の各メソッドを用いて以下のように取得します。

1. **データクラスのプロパティ。**
   データクラスのプロパティを取得するには、以下のメソッドを使用します。

   - プロパティ指定　：UserProperty getConfigurations().get (“<プロパティ名>”)
   - プロパティすべて：HashMap<String, UserProperty> getUserProperties()

   UserProerty クラスには Magic xpi の基本データ型や値が格納されています。値の取得には、UserProperty の以下のメソッドを使用します。

   Object getValue()

2. **リソースまたはサービスのプロパティ。**
   リソースまたはサービスのプロパティを取得するには、以下のメソッドを使用します。

   - リソース　：HashMap<String, String> getResourceObject()
   - サービス　：HashMap<String, String> getServiceObject()

   取得した HashMap に対して、Connector Builder で定義したサービスの項目名をキーとすることで値を取得することができます。

3. **環境設定の値。**
   サーバ実行時の環境設定の値（プロジェクトフォルダのパス、プロジェクト名、コンポーネントフォルダのパス、etc…）を取得するには、以下のメソッドを使用します。

   ```
   HashMap<String, String> getEnviromentSettings()
   ```

次に、トリガーからフローを起動するために call()メソッドを呼び出します。開発者は以下の call()メソッドの実装内容に従って、フローを起動する際の処理を実装してください。

call()

ランタイムの実行結果をフローに渡して起動し、フローの実行結果を戻り値として受け取るメソッドです。このメソッドでは、以下の２つの処理を実装します。

1. **ランタイムの実行結果をフローに渡して起動する。**
   ランタイムの実行結果を UI で設定した変数に保存し、フローを起動します。フローの起動には FlowLauncher クラスの以下のメソッドを使用します。

   ```
   Response invoke(HashMap<String, Object>)
   ```

   引数の HashMap には、データクラスに定義されているプロパティ名をキーとし、ランタイムの実行結果を設定します。なお、値を設定できるのは [TriggerIn] 属性を付与した Variable クラスのプロパティのみです。プロパティのデータ型に合わせて、以下のクラスを使用します。

   - Alpha　　：String クラス。
   - Numeric　：Double クラス。
   - Date　　：Date クラス。
   - Time　　：String クラス。
   - Logical　：Boolean クラス。
   - Blob　　：Byte クラスの配列。

2. **フローから戻り値を受け取る。**
   invoke(HashMap<String, Object>)メソッドの戻り値として Response クラスのオブジェクトを受け取ります。フローの実行結果がデータクラスのプロパティに保存されているので、以下のメソッドを使用して取得します。

   ```
   UserProperty getData(“<プロパティ名”>)
   ```

   なお、戻り値が設定されるのは [TriggerReturn] 属性を付与した Variable または Expression クラスのプロパティのみです。

disable()

トリガーを配置したフローが Enable Flow コンポーネントにより無効化された際に呼ばれるメソッドです。フローが無効化された際に実行する処理を実装します。

enable()

トリガーを配置したフローが Enable Flow コンポーネントにより有効化された際に呼ばれるメソッドです。フローが有効化された際に実行する処理を実装します。

unload()

Magic xpi のエンジンが停止した際に呼ばれるメソッドです。例えば、Magic xpi サーバが停止した際に実行する処理を実装します。

◆ Polling トリガー

続いて、トリガーの動作タイプを Polling とした場合の実装方法を説明します。Polling トリガーはテンプレートプロジェクトが生成されないため、以下で示すフローチャートおよびメソッドの概要に従って実装します。ここでは「IPollingTrigger」クラスで定義されているすべてのメソッドを実装してください。

- **トリガーの初期設定**

トリガー起動時の初期設定を行うプロセスを実装します。

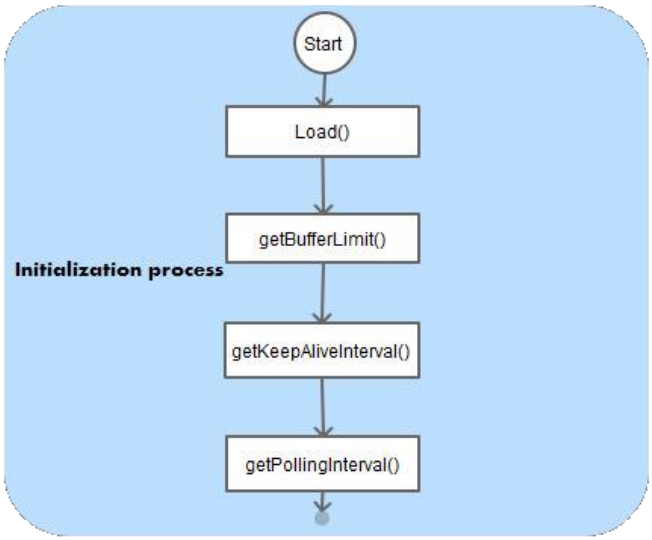


図 39：フローチャート：トリガーの初期設定

メソッドの概要：トリガーの初期設定

void load(TriggerGeneralParams param)	
実行条件	トリガーの起動時。
実装内容	このメソッドでは、トリガーの実行に必要な初期設定を行います。データクラスや各領域に設定された値を取得します。

引数	TriggerGeneralParams param　： 各領域の設定値を取得するためのパラメータオブジェクト。
戻り値	なし
int getBufferLimit()	
実行条件	load()メソッドが実行された後。
実装内容	バッファサイズの上限を返す。
引数	なし
戻り値	Int　：バッファサイズの上限（byte）。
int getKeepAliveInterval()	
実行条件	getBufferLimit()メソッドが実行された後。
実装内容	監視先とのキープアライブの通信インターバルを返す。
引数	なし
戻り値	int　：キープアライブの通信インターバル（秒）。
int getPollingInterval()	
実行条件	getKeepAliveInterval()メソッドが実行された後。
実装内容	監視先への問い合わせを行う際の監視インターバルを返す。
引数	なし
戻り値	int　：問い合わせを行う際の監視インターバル（秒）。

- **ポーリング処理**

監視先に問い合わせを行い、フロー呼び出しを行うプロセスを実装します。

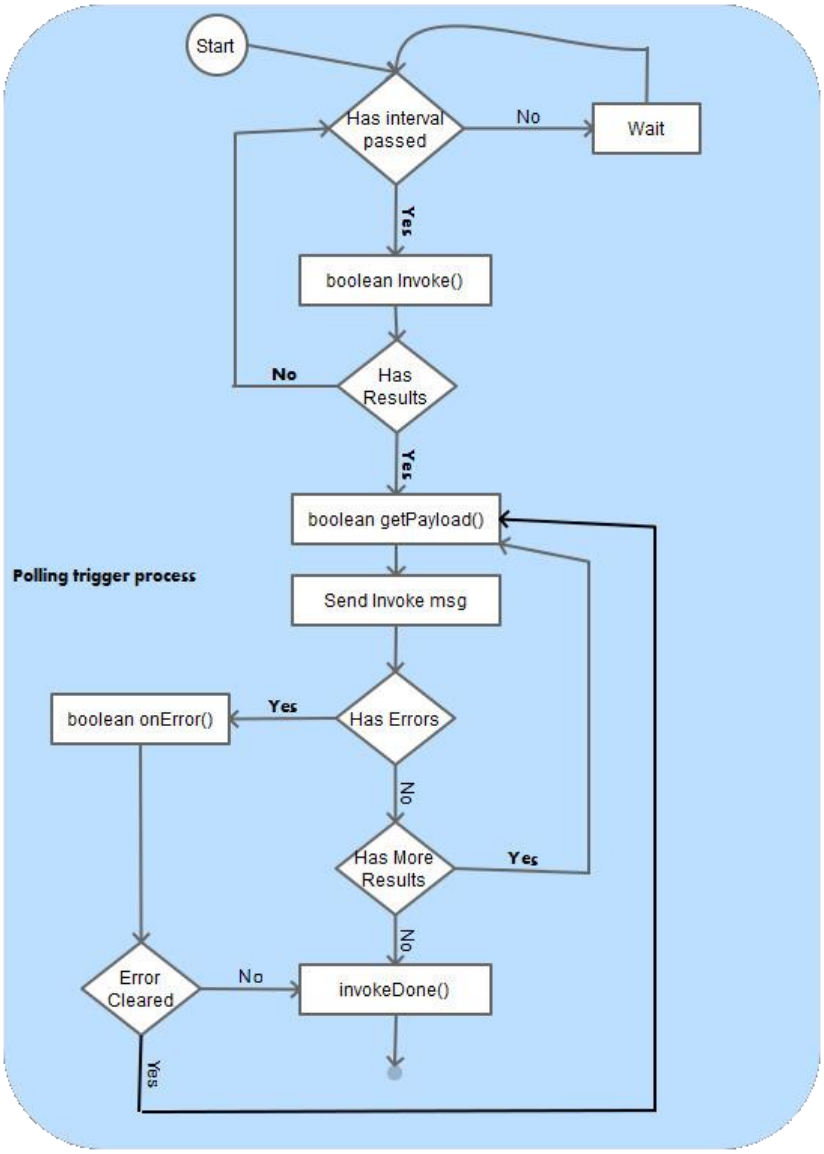


図 40：フローチャート：ポーリング処理

メソッドの概要：ポーリング処理

boolean invoke(TriggerGeneralParams param)	
実行条件	前回の問い合わせ実行時から、指定した監視インターバルが過ぎた場合。
実装内容	監視先に定期的に問い合わせを行い、応答を確認します。監視先からの応答に対して、フロー呼び出しを行うかどうかを戻り値に設定します。
引数	TriggerGeneralParams param　： 各領域の設定値を取得するためのパラメータオブジェクト。
戻り値	True　：フロー呼び出しを行う場合。 False　：フロー呼び出しを行わない場合。

boolean getPayload(HashMap<String, Object> payload)	
実行条件	Invoke()メソッドの戻り値が True の場合。 または、onError()メソッドの戻り値が True の場合。 または、自分自身の戻り値が True の場合。
実装内容	フロー呼び出しを行うメソッドです。 引数の HashMap に、データクラスに定義されているプロパティ名をキーとし、ランタイムの実行結果を設定します。値を設定できるのは [TriggerIn] 属性を付与した Variable クラスのプロパティのみです。 必要に応じて監視先から取得したデータを複数に分割し、それぞれに対してフロー呼び出しを行います。その場合、フロー呼び出しを再度行うかどうかを戻り値に設定します。
引数	HashMap<String, Object> : データクラスに定義されている[TriggerIn] 属性のプロパティ。
戻り値	True : 自分自身を呼び出し、フロー呼び出しを再度行う場合。 False : invokeDone()メソッドを実行し、フロー呼び出しを終了する場合。
void invokeDone()	
実行条件	getPayload()メソッドの戻り値が False の場合。 または、onError()メソッドの戻り値が False の場合。
説明	フロー呼び出し終了時の処理を実装します。
引数	なし
戻り値	なし
boolean onError()	
実行条件	getPayload()メソッドで例外が発生した場合。
説明	エラー発生時の処理を実装します。
引数	なし
戻り値	True : getPayload()メソッドを呼び出し、フロー呼び出しを再度行う場合。 False : invokeDone()メソッドを実行し、フロー呼び出しを終了する場合。

-Directory Scanner- の実装

Connector Builder でインターフェースタイプ[DataMapper]（以下、マッパーと示す）を使用したコンポーネント「Directory Scanner」を作成します。**※本章で作成するコンポーネントはサンプルとして作成するものです。Magic xpi に付属している「Directory Scanner」コンポーネントの作成方法を再現するものではありません。**

本書で実装する「Directory Scanner」（以下、カスタムコンポーネントと示す）の概要は以下の通りです。

ステップ：

1. マッパーの送り元から、移動前ディレクトリ、ファイル名、移動先ディレクトリの値を受け取る。
2. 移動前ディレクトリからファイルを検索（ワイルドカード可）し、該当するファイルを全て取得する。
3. 取得したファイルを全て、移動先ディレクトリに移動する。
4. 移動したファイルのファイル名やパスを戻り値とし、Magic xpi の変数に CSV 形式で出力する。

トリガー：

1. UI に入力された、移動前ディレクトリ、ファイル名、移動先ディレクトリの値を受け取る。
2. 移動前ディレクトリからファイルを検索（ワイルドカード可）し、該当するファイルを全て取得する。
3. 取得したファイルを全て、移動先ディレクトリに移動する。
4. 移動対象となったファイルの内容を戻り値とし、フローを開始する（移動したファイルの数だけスレッド数が増える）。

カスタムコンポーネントの定義

「カスタムコンポーネントの作成」章を参照し、「Connector Builder」を起動します。

「General settings」ウィンドウ表示後、[General details]セクションに対し、下記の表に従って定義を行います。

Name	MyFirstConnector
Description	任意の説明文を入力
Icon file name	任意のアイコンを選択
Toolbox group	User Components
Step default interface	Data Mapper

設定内容を下記に示します。

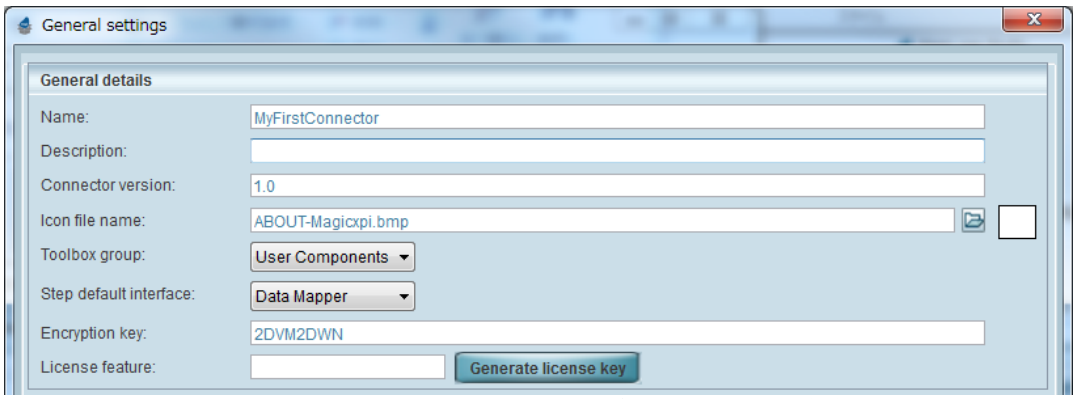


図 41：Connector Builder コンポーネントの共通設定内容

ステップの定義内容

[Step]タブを開き、下記の表に従って定義を行います。

Include Step	チェックを有効
UI Type	Dynamic
UI implementing class	SDK_TEST_1.MyFirstAdapterStep
Runtime technology	Java
Runtime implementing class	com.magicsoftware.sdk.MyFirstStep

設定内容を下記に示します。

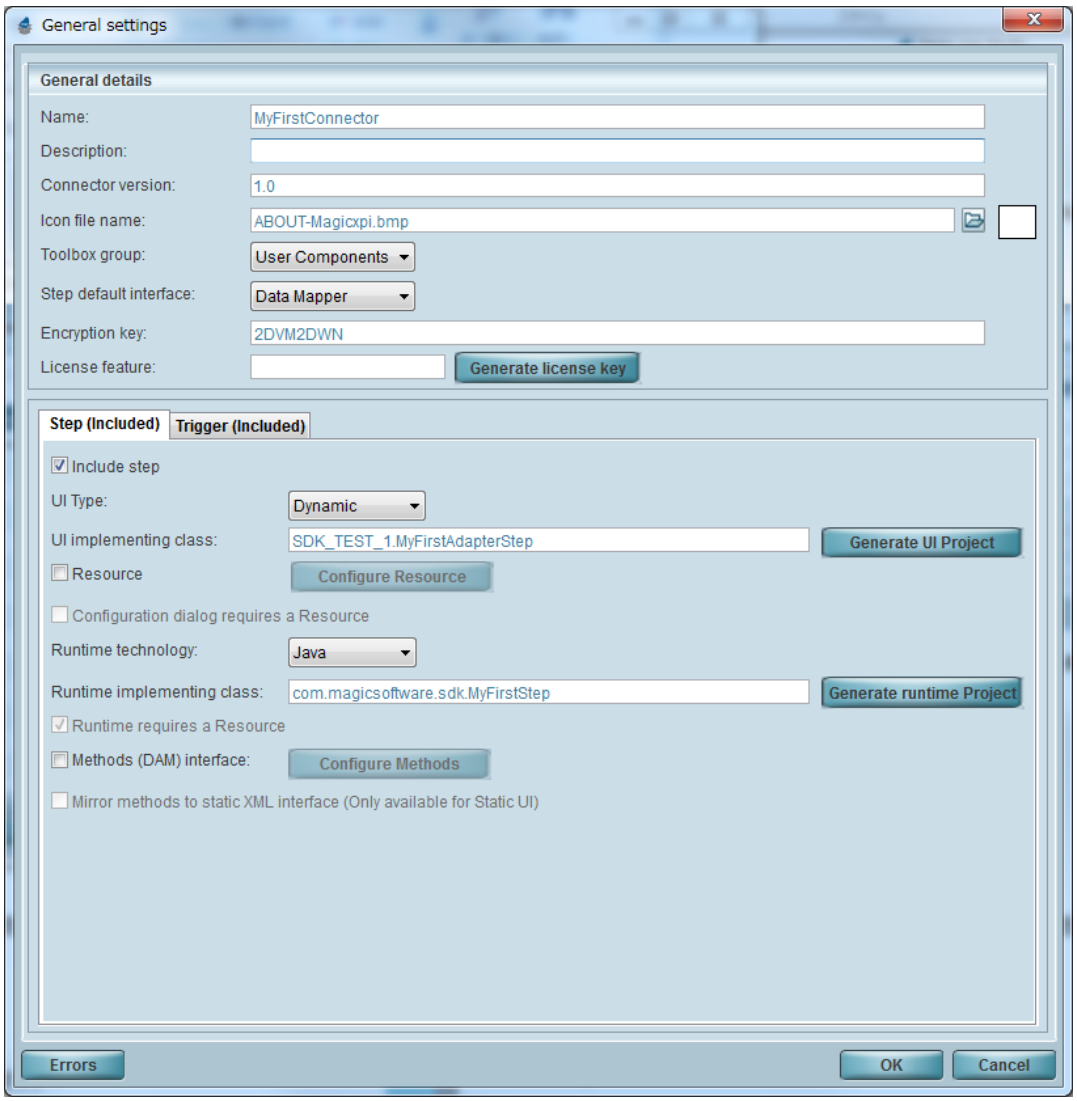


図 42：Connector Builder –ステップ–設定内容

トリガーの定義内容

[Trigger]タブを開き、下記の表に従って定義を行います。

Include Trigger	チェックを有効
UI Type	Dynamic
UI implementing class	SDK_TEST_1.MyFirstAdapterTrigger
Runtime technology	Java
Runtime implementing class	com.magicsoftware.sdk.MyFirstTrigger
Trigger invocation type	External
Flow Invocation behavior	Async

設定内容を下記に示します。

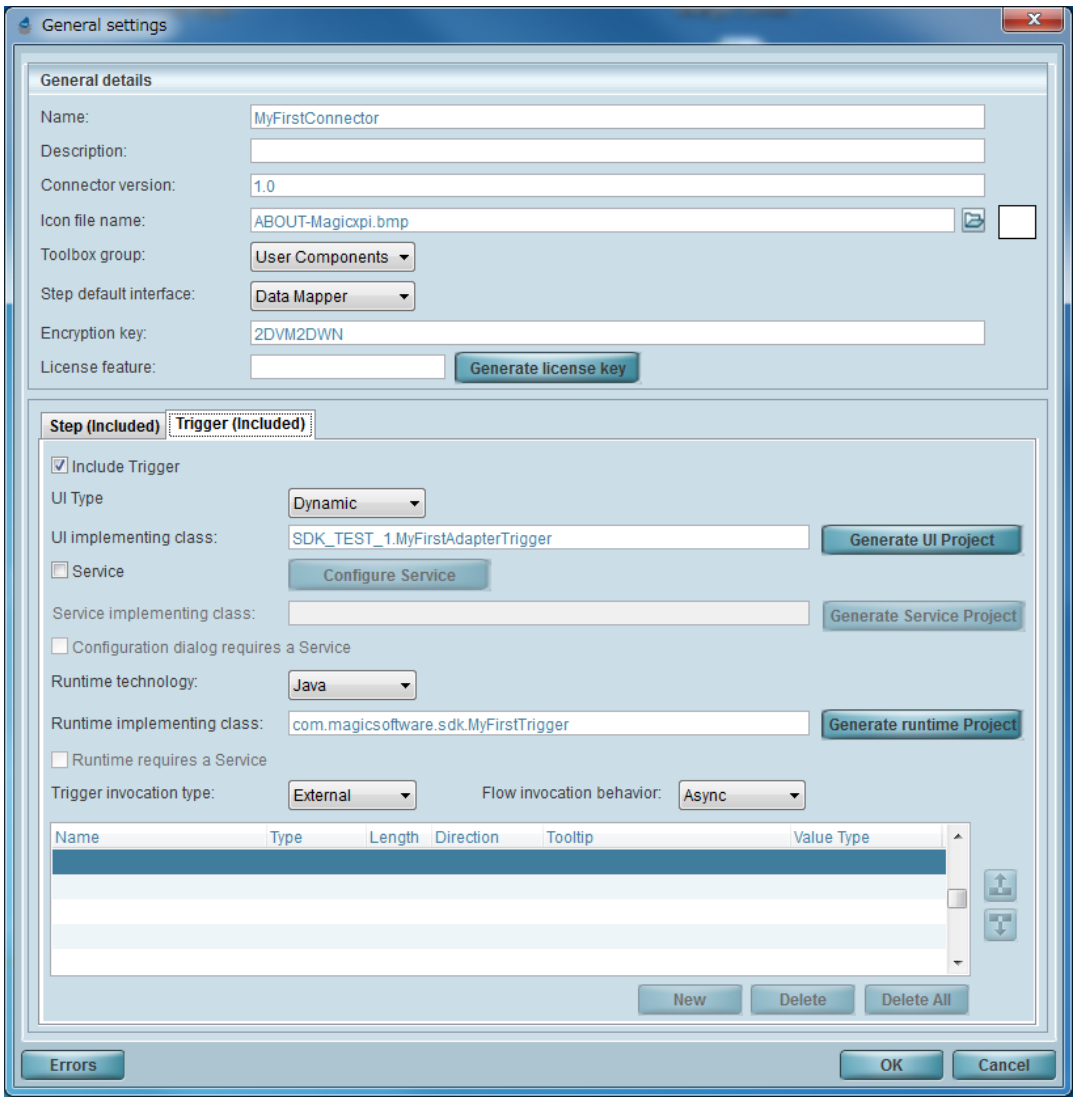


図 43：Connector Builder –トリガー–設定内容

テンプレートプロジェクトの生成

ステップテンプレートプロジェクトの生成

1. ステップ UI テンプレートプロジェクト生成
   [Step]タブを開き、[Generate UI Project]ボタンをクリックします。「Generate UI template project」表示後、以下のように入力します。

   Project name　：MyFirstAdapterStep
   Location　：C:\Users\<ユーザ名>\Documents\Visual Studio 2015\Projects

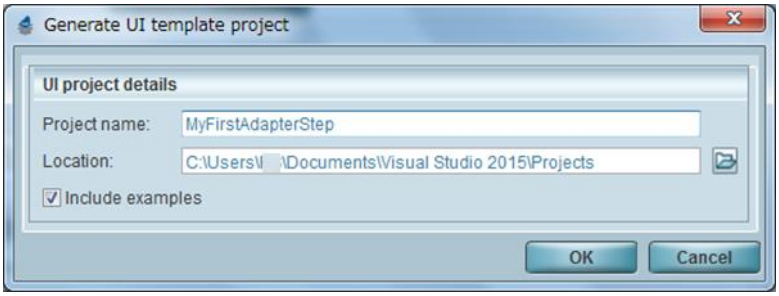


図 44：Connector Builder ステップ UI テンプレートプロジェクト生成

入力完了後、[OK]ボタンをクリックし、指定したフォルダにプロジェクトが生成されていることを確認してください。

2. ステップランタイムテンプレートプロジェクト生成
続いて、[Generate runtime Project]ボタンをクリックします。「Generate runtime template project」表示後、以下のように入力します。

Project name : MyFirstStep
Location : C:\pleiades\<workspace 名>

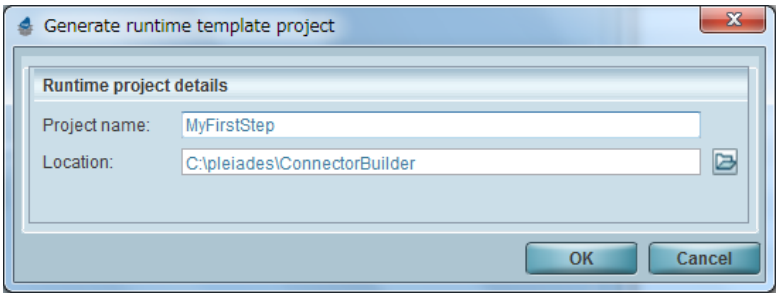


図 45：Connector Builder ステップランタイムテンプレートプロジェクト生成

入力完了後、[OK]ボタンをクリックし、指定したフォルダにプロジェクトが生成されていることを確認してください。

トリガーテンプレートプロジェクトの生成

1. トリガーUI テンプレートプロジェクト生成
[Trigger]タブを開き、[Generate UI Project]ボタンをクリックします。「Generate UI template project」表示後、以下のように入力します。

Project name : MyFirstAdapterTrigger
Location : C:\Users\<ユーザ名>\Documents\Visual Studio 2015\Projects

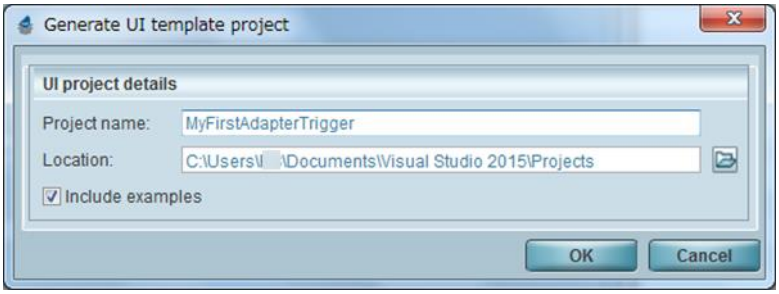


図 46：Connector Builder トリガーUI テンプレートプロジェクト生成

入力完了後、[OK]ボタンをクリックし、指定したフォルダにプロジェクトが生成されていることを確認してください。

2. トリガーランタイムテンプレートプロジェクト生成
続いて、[Generate runtime Project]ボタンをクリックします。「Generate runtime template project」表示後、以下のように入力します。

Project name ：MyFirstTrigger
Location ：C:\pleiades\<workspace 名>

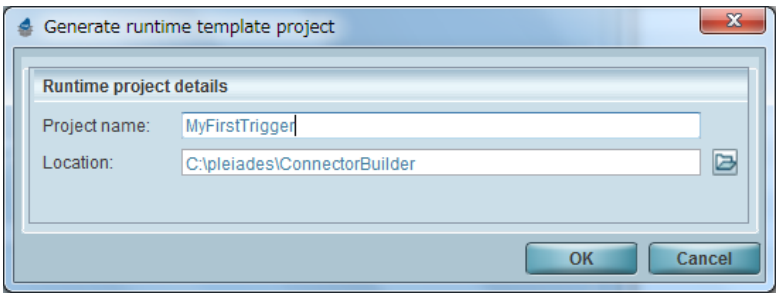


図 47：Connector Builder トリガーランタイムテンプレートプロジェクト生成

入力完了後、[OK]ボタンをクリックし、指定したフォルダにプロジェクトが生成されていることを確認してください。

コンポーネントの配置先

作成したコンポーネントは以下のフォルダに配置されています。

配置先パス：
<Magic xpi インストールフォルダ>\Runtime\addon_connectors\<作成したコンポーネントの名称>

カスタムコンポーネントの実装

実装するカスタムコンポーネントには、ファイルを移動して情報を得るための「ステップ」、ファイルを移動したことをフラグとし、フローを開始するための「トリガー」の２つを作成します。**※生成したテンプレート内にある処理の中で、本カスタムコンポーネントの実装する際に、編集不要なメソッドや変数は記載していません。**

実装するステップ UI の内容

ステップ UI カスタムコンポーネントを実装するには、以下のような手順で行います。完成した UI のイメージは下記の**図 48** を参照してください。

1. **事前準備**
2. **コンポーネントの構成を設定する UI の作成**
3. **UI の呼び出し**
4. **UI 内処理の実装**
5. **UI への入力内容の確認**
6. **UI への入力内容の再表示**
7. **実装したソース**
8. **プロジェクトのビルド**

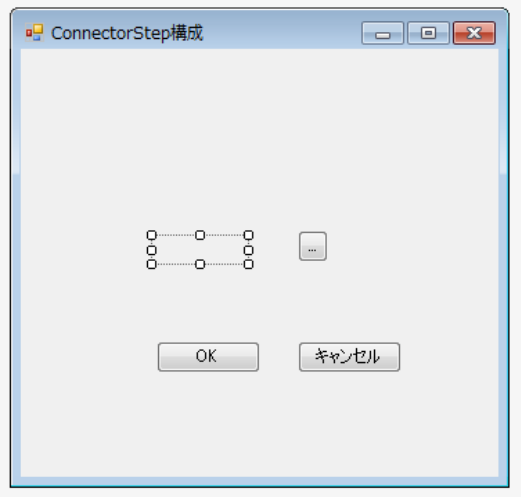


図 48：完成 UI イメージ

ステップ UI の実装

UI の実装には、ステップ UI テンプレートプロジェクト使用します。Visual Studio Express 2015 for Desktop（以下、Visual Studio Express 2015 と示す）で、生成したステップ UI テンプレートプロジェクトを開きます。

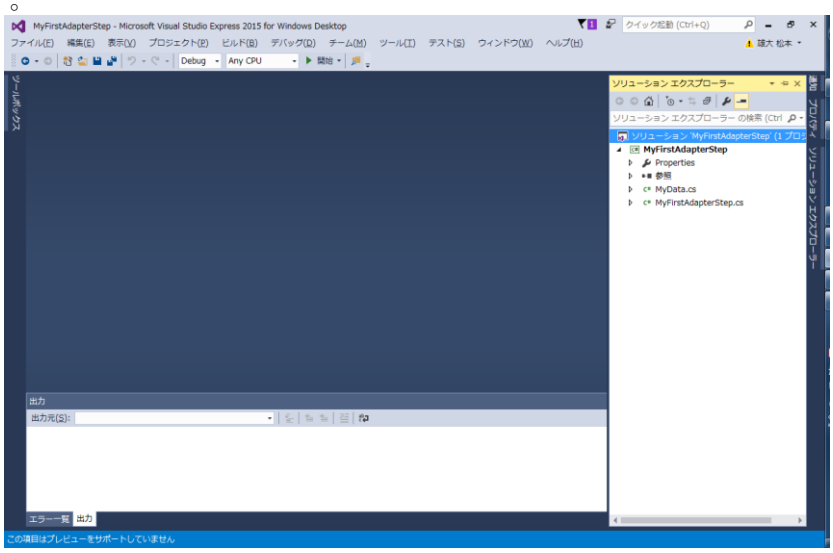


図 49：UI の作成：ステップ UI テンプレートプロジェクトを開く

1. 事前準備

1-1. マッピング項目のテンプレート作成

Magic xpiでマッパーを使用するコンポーネントを作成する場合は、送り先の定義情報を事前に準備する必要があります。送り先にXMLを使用する場合は、XSDファイルを事前に準備する必要があります。**※XSDファイルは自動生成されません。手動で作成してください。**

準備したXSDファイルは下記のフォルダに配置します。

XSDファイル配置先：

```
<Magic xpiインストールフォルダ>\Runtime\addon_connectors\<コンポーネントフォルダ>\xsd\SDK.xsd";
```

準備したXSDファイルは、MyFirstAdapterStep.csのGetXMLSchemaConfigurationメソッド内にある下記の箇所で参照しています。GetXMLSchemaConfigurationメソッドは、生成したステップUIテンプレートプロジェクト内に含まれています。**※XSDファイル配置先に合わせて参照先を修正してください。**

- XSDファイル参照箇所：

```
xmlSchemaInfoLocal.XSDSchemaFilePath = utils.GetSystemProperty("ConnectorPath")+ "\\xsd\\SDK.xsd";
```

- GetXMLSchemaConfigurationメソッド

```
public SchemaInfo GetXMLSchemaConfiguration()
{
   XMLSchemaInfo xmlSchemaInfoLocal = new XMLSchemaInfo();
   xmlSchemaInfoLocal.SchemaName = "XML_Schema_Name";
   xmlSchemaInfoLocal.AlwayCreateNodes = true;
   xmlSchemaInfoLocal.AppendData = false;
   xmlSchemaInfoLocal.DataDestinationType = 0;
   xmlSchemaInfoLocal.Description = "Description...";
   xmlSchemaInfoLocal.RecursionDepth = 3;
   xmlSchemaInfoLocal.XMLEncoding = XMLSchemaInfo.UTF_8;
   xmlSchemaInfoLocal.XMLValidation = false;
   xmlSchemaInfoLocal.XSDSchemaFilePath
      = utils.GetSystemProperty("ConnectorPath")+ "\\xsd\\SDK.xsd";
   return xmlSchemaInfoLocal;
}
```

今回使用するXSDファイルを下記のように定義します。

- XSD.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2008 sp1 (http://www.altova.com) by Bhagyashree (magic) -->
<!-- edited with XMLSPY v2004 rel. 4 U (http://www.xmlspy.com) by XMLSPY 2004 Enterprise Ed. Release 4 (MAGIC) -->

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="DirectoryScanner">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Methods">
     <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
       <xs:element name="LANToLAN" minOccurs="0" maxOccurs="unbounded">
        <xs:annotation>
         <xs:documentation>ファイルを移動します。</xs:documentation>
        </xs:annotation>
        <xs:complexType>
         <xs:sequence>
```

```
        <xs:element name="Input">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="ScannerDirectory" type="xs:string">
            <xs:annotation>
             <xs:documentation>監視したいディレクトリパスを指定します。</xs:documentation>
            </xs:annotation>
           </xs:element>
           <xs:element name="Fillter" type="xs:string">
            <xs:annotation>
             <xs:documentation>ファイル名を指定します。</xs:documentation>
            </xs:annotation>
           </xs:element>
           <xs:element name="DistinationDirectory" type="xs:string">
            <xs:annotation>
             <xs:documentation>ファイルの移動先ディレクトリパスを指定します。</xs:documentation>
            </xs:annotation>
           </xs:element>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:choice>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
</xs:element>
 </xs:schema>
```

1-2. GetXMLSchemaConfigurationメソッドの呼び出し

後述するConfigureメソッドの戻り値とconfigurationChangedの値が[True]であった場合に、自動でGetSchemaメソッドが呼ばれます。このGetSchemaメソッドから、GetXMLSchemaConfigurationメソッドの呼び出し処理を実装する必要があります。**※今回は未使用のため表記していませんが、JSONやフラットファイルを使用したい場合は、GetSchemaメソッドからGetJSonSchemaConfiguration()またはGetFFSchemaConfiguration()を呼び出す必要があります。**

下記のように、GetSchemaメソッドを修正します。

- 修正後のGetSchemaメソッド

```
public SchemaInfo GetSchema()
{
        return GetXMLSchemaConfiguration();
}
```

1-3. 保持する値の定義

MyData.csで保持する値とランタイム側の値を共有するために必要な定義を行います。
MyData.csを開き、MyDataクラスに下記のプロパティを定義します。

定義するメンバ：

```
[Id(1)]
[PrimitiveDataTypes(DataType.Blob)]
[DisplayPropertyName("selectVariable")]
[Out]
public Variable selectVariable { get; set; }
```

MyDataクラスのコンストラクタに下記のコードを追加します。

追加コード：

```
selectVariable = new Variable();
selectVariable.SetValue("");
```

実装例を以下に示します。

- MyData.cs

```
public class MyData
{
        [Id(1)]
        [PrimitiveDataTypes(DataType.Blob)]
        [DisplayPropertyName("selectVariable")]
        [Out]
        public Variable selectVariable { get; set; }

        public MyData()
        {
                selectVariable = new Variable();
                selectVariable.SetValue("");
        }
}
```

2. コンポーネントの構成を設定する UI の作成

2-1. フォームの作成

コンポーネントの構成を設定する UI を作成します。UI は Visual C#の Winodws フォームを使用して作成します。プロジェクトを右クリックし、コンテキストメニューから[追加]—[新しい項目]を選択します。

図 50：UI の作成：フォームの追加

「新しい項目の追加」ウィンドウが表示されるので、「Windows Forms」アイテムの中から「Winodws フォーム」を選択します。名前は「CharacteristicView.cs」と入力し、「追加」をクリックします。

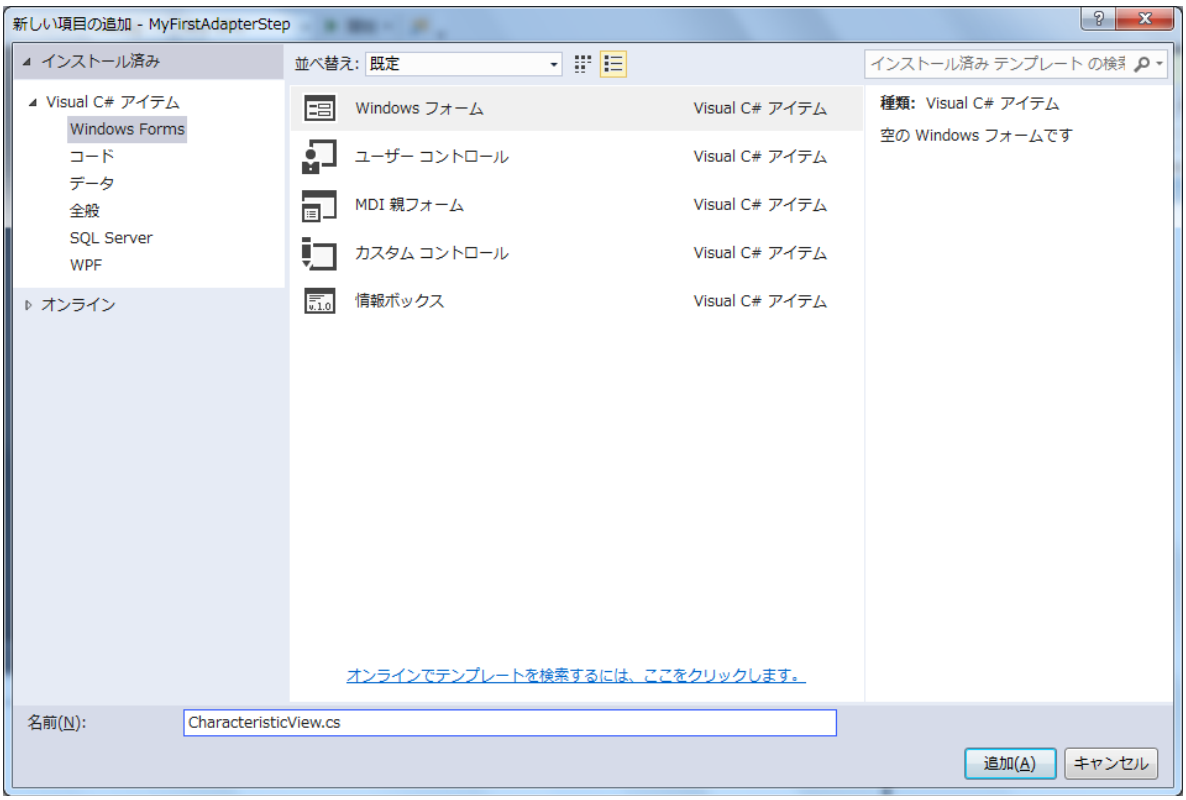
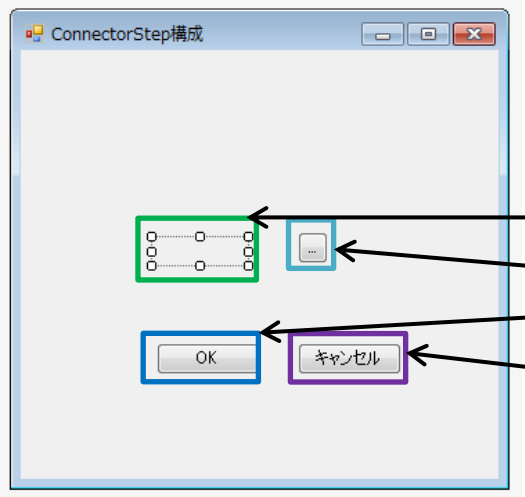


図 51：UI の作成：フォーム名

ソリューションエクスプローラに追加された CharacteristicView.cs をダブルクリックし、フォームを開きます。フォーム内にオブジェクトを配置する必要があるので、ツールボックスから「Button」を３つ、「Label」を１つ、下記のように配置します。

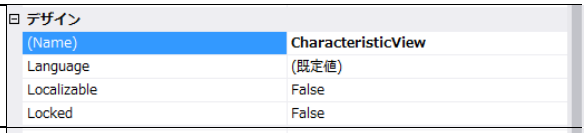
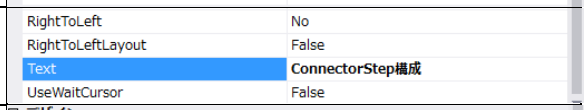
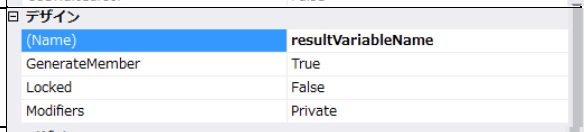
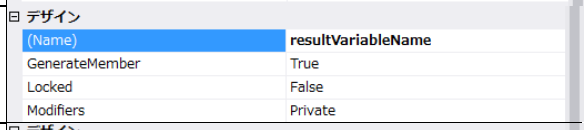
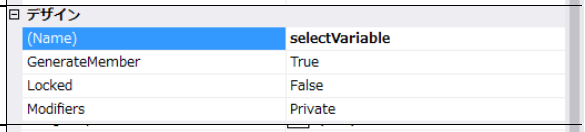
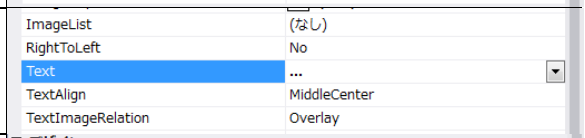
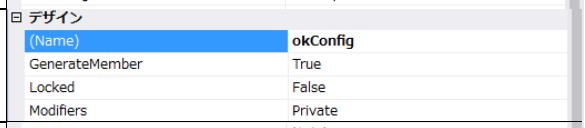
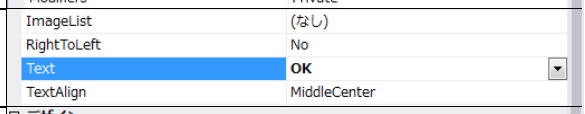
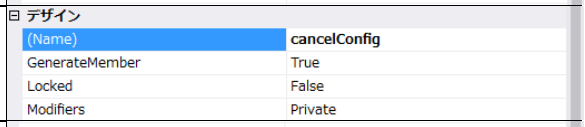
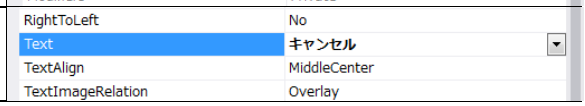


	オブジェクト名	名前
①	Form	CharacteristicView フォーム
②	Label	resultVariableName ラベル
③	Button	selectVariable ボタン
④	Button	okConfig ボタン
⑤	Button	cancelConfig ボタン

図 52：UI の作成：オブジェクトの配置

各オブジェクトのプロパティを、以下の表に従って編集します。

	オブジェクト名	設定値	参考画像

①	Form	デザイン-Name[CharacteristicView]	
		表示-Text[ConnectorStep 構成]	
②	Label	デザイン-Name[resultVariableName]	
		配置-AutoSize[True]	
③	Button	デザイン-Name[selectVariable]	
		表示-Text[…]	
④	Button	デザイン-Name[okConfig]	
		表示-Text[OK]	
⑤	Button	デザイン-Name[cancelConfig]	
		表示-Text[キャンセル]	

次に、プロパティにある「イベントボタン」をクリックし、イベントハンドラの設定を行います。

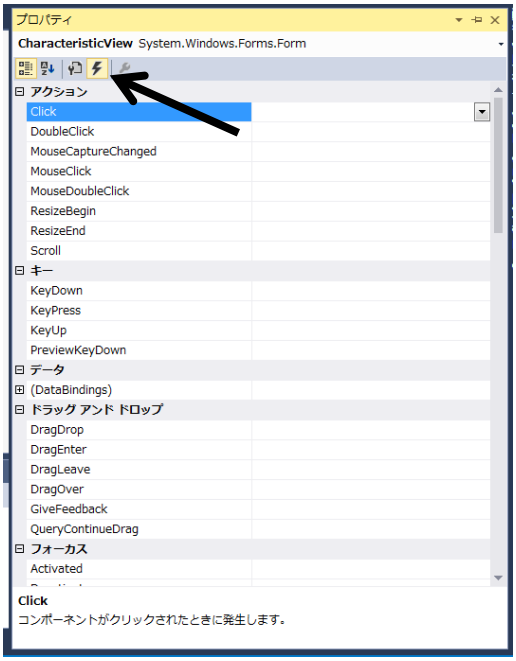
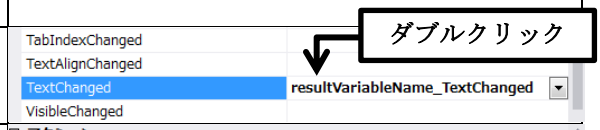
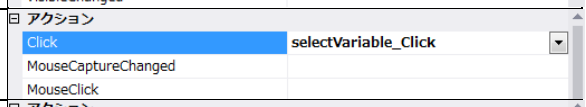
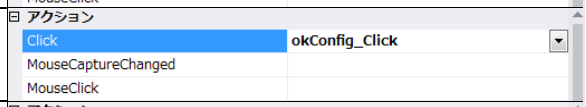
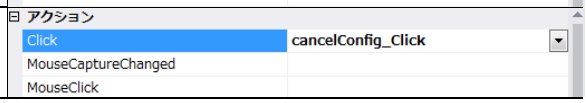


図 53：UI の作成：オブジェクトへのイベントハンドラ設定

オブジェクトへのイベントハンドラを、以下の表に従って追加します。また、追加したイベントハンドラをそれぞれダブルクリックし、イベントハンドラメソッドを生成します。

	オブジェクト名	設定値	参考画像
①	Form	変更なし	
②	Label	プロパティ変更-TextChanged [resultVariableName_TextChanged]	
③	Button	アクション-Click[selectVariable_Click]	
④	Button	アクション-Click[selectVariable_Click]	
⑤	Button	アクション-Click[selectVariable_Click]	

作成したフォームのソースコードを開きます。CharacteristicView.cs を右クリックし、「コードの表示」をクリックします。

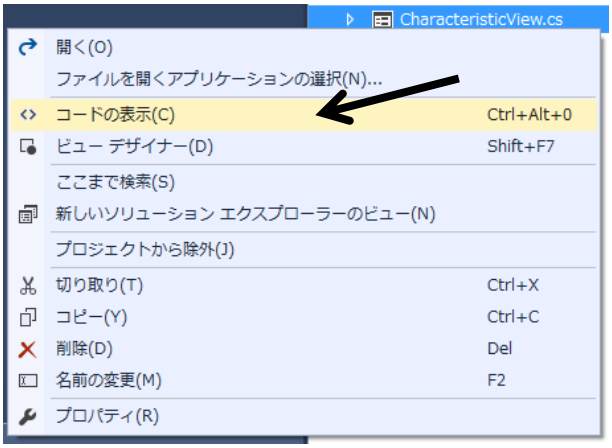


図 54：UI の作成：フォームのソースコード表示

後述する処理で、ISDKStudioUtils 型の値と MyData 型の値を受け取る必要があるため、CharacteristicView.cs の既存のコンストラクタに下記の引数を追加します。**※UI での変数リストの表示、式エディタの表示、値の保持を行う場合は必要になります。**

追加する引数：

```
ISDKStudioUtils utils
MyData storeMyData
```

受け取った引数をメンバ変数へ格納するため、CharacteristicView クラスのメンバを定義し、既存のコンストラクタに下記のコードを追加します。

定義するメンバ：

```
ISDKStudioUtils studioUtils;
MyData myData;
```

追加コード：

```
this.studioUtils = utils;
this.myData = storeMyData;
```

これまでの実装例を以下に示します。

- CharacteristicView.cs

```
public partial class CharacteristicView : Form
{
        ISDKStudioUtils studioUtils;
        MyData myData;

        public CharacteristicView(ISDKStudioUtils utils, MyData storeMyData)
        {
                InitializeComponent();

                this.studioUtils = utils;
                this.myData = storeMyData;
        }

        private void selectVariable_Click(object sender, EventArgs e)
        {

        }

        private void resultVariableName_TextChanged(object sender, EventArgs e)
        {
```

```
        }

        private void okConfig_Click(object sender, EventArgs e)
        {

        }

        private void cancelConfig_Click(object sender, EventArgs e)
        {

        }
}
```

3. UI の呼び出し

3-1. 作成したフォームをコンポーネントの UI として呼び出し

作成した CharacteristicView フォームを呼び出す処理を追加します。

また、フォームで Magic xpi の変数リストの呼び出しと、フォーム上でのデータ保持を行う必要があるため、フォームを呼び出す際に、引数として ISDKStudioUtils 型変数と MyData 型変数を渡します。MyFirstAdapterStep.cs を開き、Configure メソッドに下記のコードを追記します。

追記コード：

```
var characteristic = new CharacteristicView(utils, adaptorData);
characteristic.ShowDialog();
```

実装例を以下に示します。

- MyFirstAdapterStep.cs

```
public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
            IReadOnlyResourceConfiguration resourceData,
            object navigateTo, out bool configurationChanged)
{
        this.utils = utils;

        MyData adaptorData = new MyData();
        if (dataObject != null && dataObject is MyData)
        {
                adaptorData = (dataObject as MyData);
        }
        else
        {
                dataObject = adaptorData;

                var characteristic = new CharacteristicView(utils, adaptorData);
                characteristic.ShowDialog();
        }

        return true;
}
```

以上で作成した CharacteristicView フォームがステップカスタムコンポーネントの UI として表示されるようになります。

4. UI 内処理の実装

4-1. selectVariable ボタン

Magic xpi の変数リストを表示して変数を選択します。選択した変数には、CSV 形式のデータを格納するため、Blob 変数を選択できるようします。

また、選択した変数を myData.selectVariable に保持し、変数名を resultVariableName ラベルに表示します。

CharacteristicView フォームのソースコードを開き、selectVariable_Click メソッドに下記のコードを追記します。

追加コード：

```
myData.selectVariable = studioUtils.OpenVariablePicklist
(myData.selectVariable, VariableFilter.ALLVariables, DataType.Blob);

resultVariableName.Text = myData.selectVariable.GetValue();
```

実装例を以下に示します。

- CharacteristicView.cs

```
private void selectVariable_Click(object sender, EventArgs e)
{
        myData.selectVariable = studioUtils.OpenVariablePicklist
         (myData.selectVariable, VariableFilter.ALLVariables, DataType.Blob);

        resultVariableName.Text = myData.selectVariable.GetValue();
}
```

4-2. okConfig ボタン

構成内容への入力内容の確認をします。

構成内容が正しいか判定した値を保存するメンバ変数を、下記のメソッドを追加します。
定義するメンバ：

```
private bool configurationSuccess= false;
```

okConfig_Click メソッドに下記のコードを追加します。
追加コード：

```
if (resultVariableName.Text.Length != 0)
{
      this.configurationSuccess = true;
}

Dispose();
```

判定値を別クラスが受け取れるように、下記のメソッドを追加します。
追加メソッド：

```
public bool isConfigurationSuccess
{
      get
      {
            return this.configurationSuccess;
      }
}
```

これまでの実装例を下記に示します。

- CharacteristicView.cs

```
public partial class CharacteristicView : Form
{
        ISDKStudioUtils studioUtils;
        MyData myData;
```

```
        private bool configurationChanged = false;
        private bool configurationSuccess = false;

. . . . . .

        private void okConfig_Click(object sender, EventArgs e)
        {
                if (resultVariableName.Text.Length != 0)
                {
                        this.configurationSuccess = true;
                }

                this.Close();
        }

        public bool isConfigurationChanged
        {
                get
                {
                        return this.configurationChanged;
                }
        }

        public bool isConfigurationSuccess
        {
                get
                {
                        return this.configurationSuccess;
                }
        }

. . . . . .

}
```

4-3. resultVariableName ラベル
構成内容に変更が加えられたことを保存します。

CharacteristicView.cs を開き、CharacteristicView クラスに下記のメンバを定義します。
定義するメンバ：

```
private bool configurationChanged = false;
```

resultVariableName_TextChanged メソッドに下記のコードを追加します。
追加コード：

```
this.configurationChanged = true;
```

確認した判定値を別クラスが受け取れるように、下記のメソッドを追加します。
追加メソッド：

```
public bool isConfigurationChanged
{
      get
      {
            return this.configurationChanged;
      }
}
```

これまでの実装例を下記に示します。

- CharacteristicView.cs

```
public partial class CharacteristicView : Form
{
        ISDKStudioUtils studioUtils;
        MyData myData;
```

```
        private bool configurationChanged = false;

        private void resultVariableName_TextChanged(object sender, EventArgs e)
        {
                this.configurationChanged = true;
        }

        public bool isConfigurationChanged
        {
                get
                {
                        return this.configurationChanged;
                }
        }

}
```

4-4. cancelConfig ボタン

構成内容への変更を破棄します。CharacteristicView.cs のソースコードを開きます。

cancelConfig_Click メソッドに下記のコードを追加します。

追加コード：

```
this.configurationChanged = false;
Dispose();
```

実装例を下記に示します。

- CharacteristicView.cs

```
private void cancelConfig_Click(object sender, EventArgs e)
{
        this.configurationChanged = false;
        Dispose();
}
```

5. UI への入力内容の確認

5-1. 入力内容の確認

MyFirstAdapterStep.cs 内の Configure メソッドの戻り値が[False]だった場合は、直前に変更した内容は保持されず、マッパーも開かれません。逆に[True]だった場合は、値が保持されてマッパーが開かれます。

CharacteristicView フォームから入力内容の判定した値を MyFirstAdapterStep.cs で取得するため、下記のコードを記述します。

記述コード：

```
return characteristic.isConfigurationSuccess;
```

実装例を下記に示します。

- MyFirstAdapterStep.cs

```
public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
        IReadOnlyResourceConfiguration resourceData,
        object navigateTo, out bool configurationChanged)
{

        var characteristic = new CharacteristicView(utils, adaptorData);
        characteristic.ShowDialog();

        configurationChanged = true;

        return characteristic.isConfigurationSuccess;
```

```
}
```

5-2. 入力内容に変更があるかを確認

Magic xpi では、フローの内容に変更が加えられた場合、タブの末尾に「*」が表示されるようになっています。カスタムコンポーネントの構成内容に変更があった場合も、その「*」を表示する必要があります。

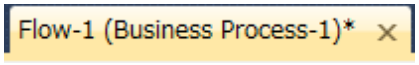


図 55：UI の作成：フロー内容変更時に表示される「*」

MyFirstAdapterStep.cs の Configure メソッドにある、bool 型の configurationChanged 変数が「*」を表示するフラグの役割を果たしています。

- configurationChanged = True の場合は「*」が付く
- configurationChanged = False の場合は「*」が付かない

CharacteristicView フォームから変更の有無を判定した値を MyFirstAdapterStep.cs で取得するため、下記のコードを記述します。

記述コード：

```
configurationChanged = characteristic.isConfigurationChanged;
```

実装例を下記に示します。

- MyFirstAdapterStep.cs

```
public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
        IReadOnlyResourceConfiguration resourceData,
        object navigateTo, out bool configurationChanged)
{
.....

        var characteristic = new CharacteristicView(utils, adaptorData);
        characteristic.ShowDialog();

        configurationChanged = characteristic.isConfigurationChanged;

.....

}
```

6. UI への入力内容の再表示

6-1. 前回 UI に保持した値を再度表示

UI を開く際に、保持してきた値を再度 CharacteristicView フォームのオブジェクトへ設定する必要があります。

CharacteristicView.cs のソースコードを開き、CharacteristicView_Load メソッド内に下記のコードを追加します。

追加コード：

```
resultVariableName.Text = myData.selectVariable.GetValue();
```

実装例を下記に示します。

- CharacteristicView.cs

```
private void CharacteristicView_Load(object sender, EventArgs e)
```

```
{
        resultVariableName.Text = myData.selectVariable.GetValue();
}
```

7. 実装したソース

これまでに実装した MyData.cs、CharacteristicView.cs、MyFirstAdapterStep.cs のソースコード全体を以下に示します。

- MyData.cs

```
using System;
using MagicSoftware.Integration.UserComponents;

namespace SDK_TEST_1
{

   public class MyData
   {
      [Id(1)]
      [PrimitiveDataTypes(DataType.Blob)]
      [DisplayPropertyName("selectVariable")]
      [AllowEmptyExpression]
      [Out]
      public Variable selectVariable { get; set; }

      public MyData()
      {
         selectVariable = new Variable();
         selectVariable.SetValue("");

      }
   }
}
```

- CharacteristicView.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using MagicSoftware.Integration.UserComponents.Interfaces;
using MagicSoftware.Integration.UserComponents;

namespace SDK_TEST_1
{
   public partial class CharacteristicView : Form
   {
      ISDKStudioUtils studioUtils;
      MyData myData;
      private bool configurationChanged = false;
      private bool configurationSuccess = false;

      public CharacteristicView(ISDKStudioUtils utils, MyData storeMyData)
      {
         InitializeComponent();

         this.studioUtils = utils;
         this.myData = storeMyData;
      }

      private void CharacteristicView_Load(object sender, EventArgs e)
      {
         resultVariableName.Text = myData.selectVariable.GetValue();
      }
```

```
    private void selectVariable_Click(object sender, EventArgs e)
    {
      myData.selectVariable = studioUtils.OpenVariablePicklist
        (myData.selectVariable, VariableFilter.ALLVariables, DataType.Blob);
      resultVariableName.Text = myData.selectVariable.GetValue();
    }

    public bool isConfigurationChanged
    {
      get
      {
        return this.configurationChanged;
      }
    }

    public bool isConfigurationSuccess
    {
      get
      {
        return this.configurationSuccess;
      }
    }

    private void resultVariableName_TextChanged(object sender, EventArgs e)
    {
      this.configurationChanged = true;
    }

    private void okConfig_Click(object sender, EventArgs e)
    {

      if (resultVariableName.Text.Length != 0)
      {
        this.configurationSuccess = true;
      }

      Dispose();
    }

    private void cancelConfig_Click(object sender, EventArgs e)
    {
      this.configurationChanged = false;

      Dispose();
    }

  }
}
```

- MyFirstAdapterStep.cs

```
using System;
using System.Collections.Generic;
using MagicSoftware.Integration.UserComponents.Interfaces;
using MagicSoftware.Integration.UserComponents;
using System.ComponentModel.Composition;
using System.Windows.Forms;

namespace SDK_TEST_1
{

  [Export(typeof(IUserComponent))]
  public class MyFirstAdapterStep : IUserComponent
  {
    private ISDKStudioUtils utils;
    public MyFirstAdapterStep()
    {
    }
```

```
#region IUserComponent implementation

public object CreateDataObject()
{
   return new MyData();
}

public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
   IReadOnlyResourceConfiguration resourceData,
   object navigateTo, out bool configurationChanged)
{

   this.utils = utils;

   MyData adaptorData = new MyData();
   if (dataObject != null && dataObject is MyData)
      adaptorData = (dataObject as MyData);
   else
      dataObject = adaptorData;

   var characteristic = new CharacteristicView(utils, adaptorData);
   characteristic.ShowDialog();

   configurationChanged = characteristic.isConfigurationChanged;

   return characteristic.isConfigurationSuccess;
}

public SchemaInfo GetSchema()
{
   return GetXMLSchemaConfiguration();
}

public ICheckerResult Check(ref object data,
   IReadOnlyResourceConfiguration resourceData)
{
   return null;
}

public bool ValidateResource
   (IReadOnlyResourceConfiguration resouceData, out string errorMsg)
{
   errorMsg = null;

   return true;
}

public void InvokeResourceHelper(string helperID,
   IResourceConfiguration resouceData)
{

}

#endregion

public SchemaInfo GetXMLSchemaConfiguration()
{
   XMLSchemaInfo xmlSchemaInfoLocal = new XMLSchemaInfo();
   xmlSchemaInfoLocal.SchemaName = "XML_Schema_Name";
   xmlSchemaInfoLocal.AlwayCreateNodes = true;
   xmlSchemaInfoLocal.AppendData = false;
   xmlSchemaInfoLocal.DataDestinationType = 0;
   xmlSchemaInfoLocal.Description = "Description...";
   xmlSchemaInfoLocal.RecursionDepth = 3;
   xmlSchemaInfoLocal.XMLEncoding = XMLSchemaInfo.UTF_8;
   xmlSchemaInfoLocal.XMLValidation = false;
   xmlSchemaInfoLocal.XSDSchemaFilePath
      = utils.GetSystemProperty("ConnectorPath")+ "\\xsd\\SDK.xsd";
   return xmlSchemaInfoLocal;
```

```
    }

  }
}
```

8. **プロジェクトのビルド**
   以上で UI の作成は終了です。Visual Studio Express 2015 のメニューバーから[ビルド]—[ソリューションのリビルド]を選択し、ビルドを行います。

図 56：ステップ UI の作成：UI クラスのビルド

下記のコピー元・コピー先を参照し、ビルド後に生成されるファイルをコピーしてください。

コピー元：
・[UI クラスのプロジェクトフォルダ]\MyFirstAdapterStep\bin\Debug\MyFirstAdapterStep.dll
・[UI クラスのプロジェクトフォルダ]\MyFirstAdapterStep\bin\Debug\MyFirstAdapterStep.pdb

コピー先：
・[コンポーネントフォルダ] \ui\lib\MyFirstAdapterStep.dll
・[コンポーネントフォルダ] \ui\lib\MyFirstAdapterStep.pdb

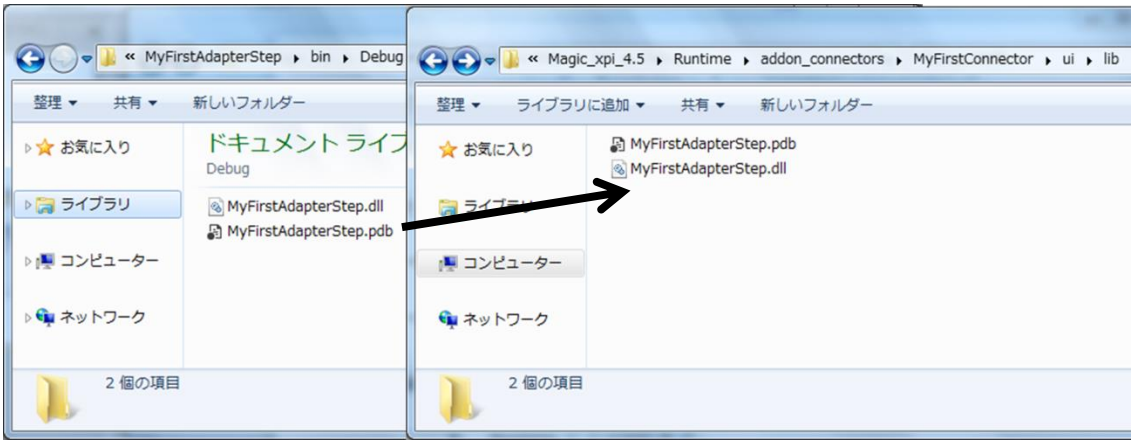


図 57：ステップ UI の作成：UI クラスのビルド生成物のコピー

実装するステップランタイムの内容

ステップのランタイムを実装するには、以下のような手順で行います。

1. **ログ出力の設定**
2. **マッピング内容の取得**
3. **マッピング内容を解析し、対象ファイルを取得および移動**
4. **取得した値から、CSV 形式の戻り値を作成**

5. **ランタイムで生成した戻り値を Magic xpi の変数へ格納**
6. **実装したソース**
7. **ランタイムクラスの Jar ファイル生成**

ランタイムの実装には、ステップランタイムテンプレートプロジェクトを使用します。Eclipse 4.6 Neon Pleiades All in One（以下、Eclipse 4.6 と示す）で、生成したステップランタイムテンプレートプロジェクトを開きます。

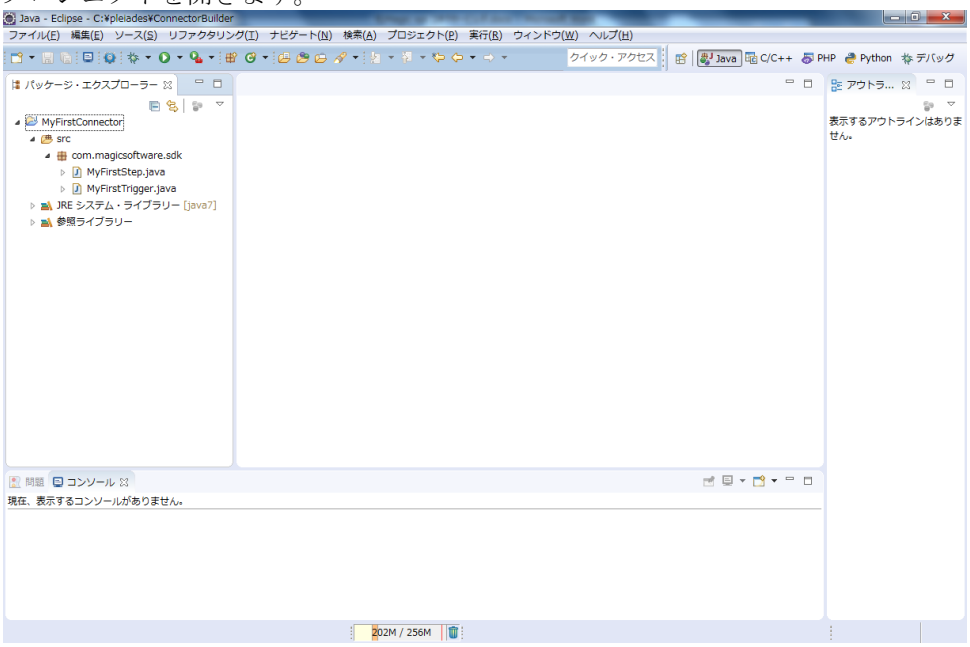


図 58：ステップランタイムの作成：ステップランタイムテンプレートプロジェクトを開く

ステップランタイムの実装

1. ログ出力の設定

1-1. ログ出力の設定

実装するコンポーネントごとにログ出力をするための設定を行います。Magic xpi をインストールした際に下記のフォルダに「log4j.xml」が生成されているので、そのファイルを編集します。

log4j.xml 生成先：
<Magic xpi インストールディレクトリ>\Runtime\java\classes\log4j.xml

1-2. log4j.xml の編集

log4j.xml の START MAGIC_INFO APPENDERS セクションの末尾に下記のように追記します。

追記内容：

```
<appender name="MyFirstAdapter-Step-appender" class="org.apache.log4j.RollingFileAppender">
        <param name="Threshold" value="&fileDefThreshold;"/>
        <param name="File" value="C:/Magic_xpi_4.5/Runtime/logs/java/MyFirstAdapter_${pid}.log"/>
        <param name="MaxFileSize" value="&fileSize;"/>
        <param name="MaxBackupIndex" value="&fileBackups;"/>
        <layout class="org.apache.log4j.PatternLayout">
                <param name="ConversionPattern" value="&XpiPattern;"/>
        </layout>
 </appender>
```

続いて、START MAGIC_INFO LOGGERS セクションの末尾に下記のように追記します。

```
<logger name="com.magicsoftware.sdk.MyFirstStep" additivity="false">
        <level value="debug"/>
        <appender-ref ref="MyFirstAdapter-Step-appender"/>
</logger>
```

1-3. Log4j の定義

下記のように編集し、コンポーネントごとにログ出力するようにします。

編集前：

```
private final static Logger log = Logger.getLogger("MyStepLoggerName");
```

編集後：

```
private final static Logger log = Logger.getLogger("com.magicsoftware.sdk.MyFirstStep");
```

2. マッピング内容の取得

2-1. マッピング内容の取得

Magic xpi でマッピングされた値を受け取る処理を実装します。マッピングされた値は XML でランタイムに渡されます。マッピングされた値は params.getPayloadOBject()で受け取ることができます。次章で XML の解析を行うため、InputSource 型の変数へ格納しています。

MyFirstStep.java を開き、Invoke メソッドに下記のコードを追加します。

追加コード：

```
InputSource inputSource = new InputSource(new ByteArrayInputStream(params.getPayloadOBject()));
```

実装例を下記に示します。

- MyFirstStep.java

```
public void invoke(StepGeneralParams params) throws SDKException{

        try {
                InputSource inputSource = new InputSource( new ByteArrayInputStream(params.getPayloadOBject()));

        } catch (Exception e) {

        }
}
```

3. マッピング内容を解析し、対象ファイルを取得および移動

3-1. マッピング内容の解析

[2.マッピング内容の取得]でも記述した通り、マッピングされた値は XML でランタイムに渡されます。XML を解析するメソッドを新たに作成します。下記の定数とメソッドを追加します。

追加定数：

```
private final static String MAPPING_VALUE_DIRECTORY = "ScannerDirectory";
private final static String MAPPING_VALUE_FILLTER = "Fillter";
private final static String MAPPING_VALUE_DISTINATION_DIRECTORY = "DistinationDirectory";
```

新規追加メソッドの処理概要：

1. マッピングされた値（XML）から Input 要素ごとに反復処理を実施。
2. Input 要素から子要素を抽出し、子要素名をキーとした要素値を連想配列で保持。
3. マッピングされた値を参照し、移動対象となるファイルを特定し、ファイルを移動。
4. ファイル移動が成功した場合は、移動したファイルのファイル名やパスを新たな連想配列で保持。

5. メソッドの戻り値として、Input 要素ごとに[4.]で保持していた連想配列内のファイル名やパスを返す。

新規追加メソッド：

```
public ArrayList<HashMap<String, String>> extractionMappingValue(InputSource mappingValue)
        throws SDKException {

                ArrayList<HashMap<String, String>> targetFileInfoList =
                        new ArrayList<HashMap<String, String>>();

                try {
                        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

                        DocumentBuilder builder;
                        builder = factory.newDocumentBuilder();

                        Document document = builder.parse(mappingValue);

                        NodeList inputNodeList = document.getElementsByTagName("Input");

                        for (int nodeCount=0; nodeCount<inputNodeList.getLength(); nodeCount++) {

                                NodeList inputChildList = inputNodeList.item(nodeCount).getChildNodes();

                                final HashMap<String, String> mappingDirectoryScanner = new HashMap<String, String>();

                                for (int inputChildCnt = 0;  inputChildCnt < inputChildList.getLength(); inputChildCnt++) {

                                        Node inputChild = inputChildList.item(inputChildCnt);

                                        if (inputChild.getNodeType() == Node.ELEMENT_NODE) {

                                                mappingDirectoryScanner.put(inputChild
                                                        .getNodeName(),inputChild.getTextContent());

                                        }
                                }

                                final FilenameFilter filterFilePath = new FilenameFilter() {
                                        @Override
                                        public boolean accept(File dir, String name) {

                                                if(name.matches(mappingDirectoryScanner
                                                  .get(MAPPING_VALUE_FILLTER)
                                                  .replace("*", ".*").replace("..", "."))){
                                                        return true;
                                                  } else {
                                                        return false;
                                                  }
                                        }
                                };

                                File directory = new
File(mappingDirectoryScanner.get(MAPPING_VALUE_DIRECTORY));

                                File[] filterFilePathList = directory.listFiles(filterFilePath);

                                for (File moveFilePath : filterFilePathList) {

                                File distinationDirectory = new File(mappingDirectoryScanner
                                        .get(MAPPING_VALUE_DISTINATION_DIRECTORY)
                                             + "\\" + moveFilePath.getName());

                                 if (distinationDirectory.isFile()) {
                                        distinationDirectory.delete();
                                 }

                                 if (moveFilePath.isFile()) {
                                        if (moveFilePath.renameTo(distinationDirectory)) {
```

```
                                        HashMap<String, String> targetFileInfo  = new HashMap<String, String>();

                                        targetFileInfo.put(MOVE_BEFORE_FILE_PATH,moveFilePath.getPath());
targetFileInfo.put(MOVE_AFTER_FILE_PATH,distinationDirectory.getPath());
                                        targetFileInfo.put(TARGET_FILE_NAME,moveFilePath.getName());

                                        targetFileInfoList.add(targetFileInfo);

                                        log.debug("Step success ");
                                } else {
                                        log.debug("Step failure ");
                                }
                        }
                }
        }
        } catch (Exception e) {
                throw new SDKException(100,"Error:"+e.toString());
        }
        return targetFileInfoList;
}
```

3-2. 解析結果の受け取り

[3-1. マッピング内容の解析]で実装したメソッドの戻り値を受け取るため、invoke メソッドに下記のコードを追加します。

追加コード：

```
ArrayList<HashMap<String, String>> mappingValueList = extractionMappingValue(inputSource);
```

invoke メソッドの実装例を下記に示します。

- invoke メソッド

```
public void invoke(StepGeneralParams params) throws SDKException{

        try {
                InputSource inputSource = new InputSource( new ByteArrayInputStream(params.getPayloadOBject()));

                ArrayList<HashMap<String, String>> mappingValueList = extractionMappingValue(inputSource);

                } catch (Exception e) {

                }
}
```

4. 取得した値から、CSV 形式の戻り値を作成

4-1. CSV の生成

移動したファイルのファイル名やパスを参照して CSV を生成します。新たに下記の定数とメソッドを追加します。

追加定数：

```
private final static String MOVE_BEFORE_FILE_PATH = "MoveBeforeFilePath";
private final static String MOVE_AFTER_FILE_PATH = "MoveAfterFilePath";
private final static String TARGET_FILE_NAME = "TargetFileName";
```

新規追加メソッド処理概要：

1. 移動したファイルの移動した情報を参照し、CSV 作成。
2. CSV の作成に成功した場合は、その作成した CSV を返す。

新規追加メソッド：

```
public String createOutputCsv(ArrayList<HashMap<String, String>> mappingValueList) {

        StringBuilder sb = new StringBuilder();
```

```
        for (HashMap<String, String> valueList : mappingValueList) {

                sb.append(valueList.get(MOVE_BEFORE_FILE_PATH));
                sb.append(",");
                sb.append(valueList.get(MOVE_AFTER_FILE_PATH));
                sb.append(",");
                sb.append(valueList.get(TARGET_FILE_NAME));
                sb.append("\n");

        }

        if (sb.toString().isEmpty()){
                return "";
        }

        return sb.toString();

}
```

4-2. 生成した CSV の受け渡し

[4-1. CSV の生成]で実装したメソッドの戻り値（CSV）を受け取るため、invoke メソッドに下記のコードを追加します。

追加コード：

```
String stringResult = createOutputCsv(mappingValueList);
```

invoke メソッドの実装例を下記に示します。

- invoke メソッド

```
public void invoke(StepGeneralParams params) throws SDKException{

        try {
                InputSource inputSource = new InputSource( new ByteArrayInputStream(params.getPayloadOBject()));

                ArrayList<HashMap<String, String>> mappingValueList = extractionMappingValue(inputSource);

                String stringResult = createOutputCsv(mappingValueList);

                } catch (Exception e) {

                }
}
```

5. ランタイムで生成した戻り値を Magic xpi の変数へ格納

5-1. Magic xpi の変数へ格納

ランタイムで生成した値を Magic xpi の変数へ渡すためには、ステップ UI 作成時に定義したデータクラス内にある Variable 型のプロパティに設定する必要があります。MyFirstStep.java を開き、Invoke メソッドに下記のコードを追加します。

追加コード：

```
UserProperty selectVariable = params.getUserProperty("selectVariable");
selectVariable.setBlob(stringResult.getBytes());
```

これまでの Invoke メソッドの実装例を下記に示します。

- MyFirstStep.java

```
public void invoke(StepGeneralParams params) throws SDKException{

        try {
                InputSource inputSource = new InputSource(new ByteArrayInputStream(params.getPayloadOBject()));

                ArrayList<HashMap<String, String>> mappingValueList = extractionMappingValue(inputSource);
```

```
                String stringResult = createOutputCsv(mappingValueList);

                UserProperty selectVariable = params.getUserProperty("selectVariable");
                selectVariable.setBlob(stringResult.getBytes());

        } catch (Exception e) {

        }
}
```

6. 実装したソース

これまでに実装した MyFirstStep.java のソースコード全体を以下に示します。

- MyFirstStep.java

```
package com.magicsoftware.sdk;

import java.io.BufferedWriter;
import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.FilenameFilter;
import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.ArrayList;
import java.util.HashMap;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

import com.magicsoftware.xpi.sdk.SDKException;
import com.magicsoftware.xpi.sdk.UserProperty;
import com.magicsoftware.xpi.sdk.step.IStep;
import com.magicsoftware.xpi.sdk.step.StepGeneralParams;

public class MyFirstStep implements IStep {

        private final static Logger log = Logger.getLogger("com.magicsoftware.sdk.MyFirstStep");

        private final static String MAPPING_VALUE_DIRECTORY = "ScannerDirectory";
        private final static String MAPPING_VALUE_FILLTER = "Fillter";
        private final static String MAPPING_VALUE_DISTINATION_DIRECTORY = "DistinationDirectory";

        private final static String MOVE_BEFORE_FILE_PATH = "MoveBeforeFilePath";
        private final static String MOVE_AFTER_FILE_PATH = "MoveAfterFilePath";
        private final static String TARGET_FILE_NAME = "TargetFileName";

        @Override
        public void invoke(StepGeneralParams params) throws SDKException{

                try {

                        InputSource inputSource = new InputSource(
                                        new ByteArrayInputStream(params.getPayloadOBject()));

                        ArrayList<HashMap<String, String>> mappingValueList =
extractionMappingValue(inputSource);

                        String stringResult = createOutputCsv(mappingValueList);

                        UserProperty selectVariable = params.getUserProperty("selectVariable");
                        selectVariable.setBlob(stringResult.getBytes());
```

```
		} catch (Exception e) {

		}

	}

	public ArrayList<HashMap<String, String>> extractionMappingValue(InputSource mappingValue)
			throws SDKException {

		ArrayList<HashMap<String, String>> targetFileInfoList =
				new ArrayList<HashMap<String, String>>();

		try {
			DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
			DocumentBuilder builder;

			builder = factory.newDocumentBuilder();
			Document document = builder.parse(mappingValue);

			NodeList inputNodeList = document.getElementsByTagName("Input");

			for (int nodeCount=0; nodeCount<inputNodeList.getLength(); nodeCount++) {
				NodeList inputChildList = inputNodeList.item(nodeCount).getChildNodes();

				final HashMap<String, String> mappingDirectoryScanner
					= new HashMap<String, String>();

				for (int inputChildCnt = 0;
					inputChildCnt < inputChildList.getLength(); inputChildCnt++) {

					Node inputChild = inputChildList.item(inputChildCnt);

					if (inputChild.getNodeType() == Node.ELEMENT_NODE) {

						mappingDirectoryScanner
						.put(inputChild.getNodeName(),inputChild.getTextContent());

					}
			}

				final FilenameFilter filterFilePath = new FilenameFilter() {
					@Override
					public boolean accept(File dir, String name) {
						if(name.matches(mappingDirectoryScanner
									.get(MAPPING_VALUE_FILLTER)
									.replace("*", ".*").replace("..", "."))){

							return true;
						} else {
							return false;
						}
					}
				};

				File directory
				= new File(mappingDirectoryScanner.get(MAPPING_VALUE_DIRECTORY));

				File[] filterFilePathList = directory.listFiles(filterFilePath);

				for (File moveFilePath : filterFilePathList) {

					File distinationDirectory
						= new File(mappingDirectoryScanner
						.get(MAPPING_VALUE_DISTINATION_DIRECTORY)
						+ "\\" + moveFilePath.getName());

					if (distinationDirectory.isFile()) {
						distinationDirectory.delete();
					}
```

```
                                        if (moveFilePath.isFile()) {
                                                if (moveFilePath.renameTo(distinationDirectory)) {

                                                        HashMap<String, String> targetFileInfo
                                                                = new HashMap<String, String>();

                                                        targetFileInfo
                                                        .put(MOVE_BEFORE_FILE_PATH
                                                        ,moveFilePath.getPath());

                                                        targetFileInfo
                                                        .put(MOVE_AFTER_FILE_PATH
                                                        ,distinationDirectory.getPath());

                                                        targetFileInfo
                                                        .put(TARGET_FILE_NAME
                                                        ,moveFilePath.getName());

                                                        targetFileInfoList.add(targetFileInfo);

                                                        log.info("Step moving success");
                                                } else {
                                                        log.info("Step moving failure");
                                                }
                                        }
                                }
                        }
                } catch (Exception e) {

                }

                return targetFileInfoList;
        }

        public String createOutputCsv(ArrayList<HashMap<String, String>> mappingValueList) {

                StringBuilder sb = new StringBuilder();

                for (HashMap<String, String> valueList : mappingValueList) {

                        sb.append(valueList.get(MOVE_BEFORE_FILE_PATH));
                        sb.append(",");
                        sb.append(valueList.get(MOVE_AFTER_FILE_PATH));
                        sb.append(",");
                        sb.append(valueList.get(TARGET_FILE_NAME));
                        sb.append("\n");

                }

                if (sb.toString().isEmpty()){
                        return "";
                }

                return sb.toString();
        }
}
```

7. **ランタイムクラスの Jar ファイル生成**
   以上でランタイムの作成は終了です。ステップランタイムテンプレートプロジェクトを右クリックし、コンテキストメニューからエクスポートを選択します。

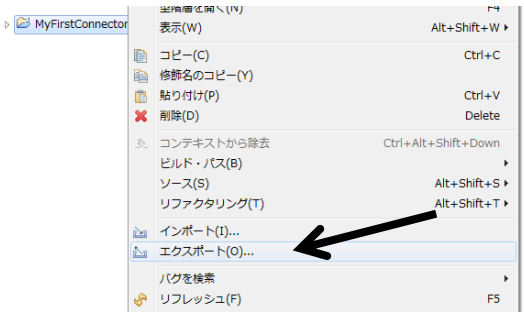


図 59：ステップランタイムの作成：JAR ファイルエクスポート 1

エクスポートウィンドウ表示後、[Java]—[JAR ファイル]と選択し、[次へ]ボタンをクリックします。

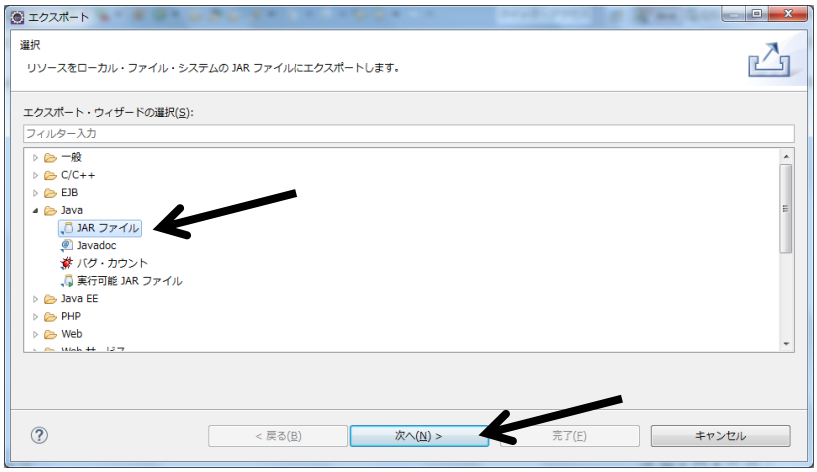


図 60：ステップランタイムの作成：JAR ファイルエクスポート 2

JAR エクスポートウィンドウ表示後、下記を参照してエクスポート先を指定し、 [完了]ボタンをクリックします。

エクスポート先：
[コンポーネントフォルダ] \runtime\java\lib\MyFirstConnector.jar

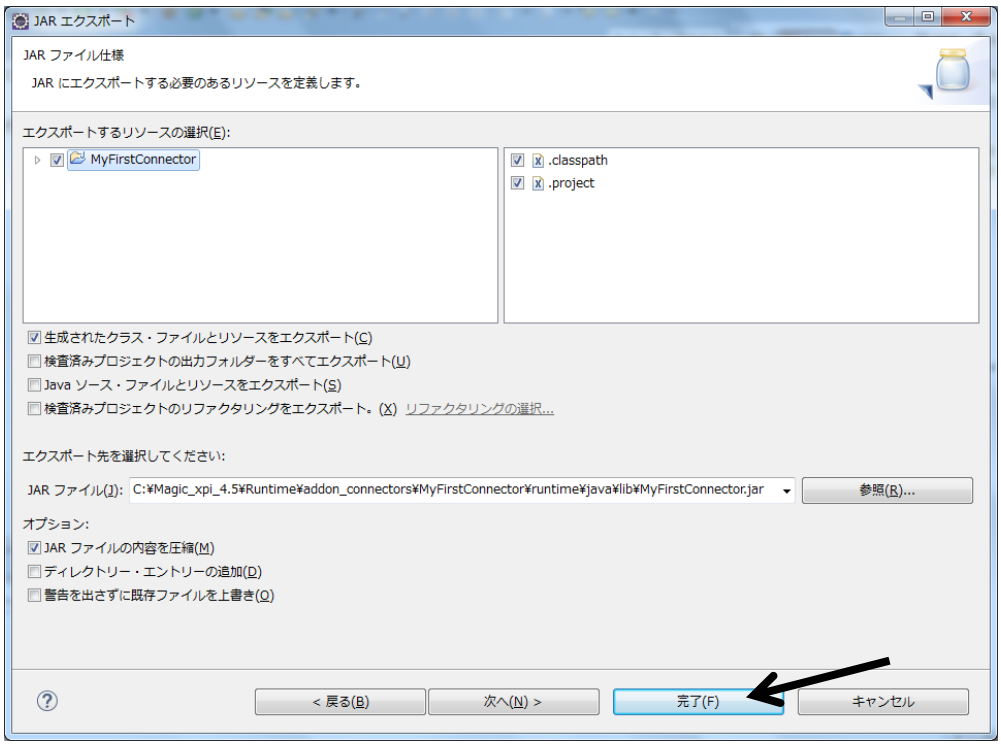


図 61：ステップランタイムの作成：JAR ファイルエクスポート 3

実装したステップの動作確認

1. 動作確認に必要な事前準備
   動作確認の事前準備として下記の表に従い、任意のディレクトリ、任意のファイルを作成します。

   動作確認で使用する例：

種類	パス	使用用途
ディレクトリ	C:\temp	移動前ディレクトリ
ディレクトリ	C:\temp\Destination	移動先ディレクトリ
ファイル	C:\temp\Step.txt	移動対象ファイル

   上記表内の「移動対象ファイル」は、空でも問題ありません。

2. 動作確認の実施
   これまで実装したコンポーネントを、実際に Magic xpi で実行して動作確認を行います。Magic xpi を起動し、新しいプロジェクトを作成します。その後、「Flow-1」というフローが自動で追加されます。フローが自動起動するように、プロパティから Auto Start プロパティを「Yes」に設定しておきます。

   作成した「MyFirstConnector」コンポーネントがツールボックス内の「User Conponents」グループに表示されています。「MyFirstConnector」コンポーネントをフローエリアへドラッグ＆ドロップします。

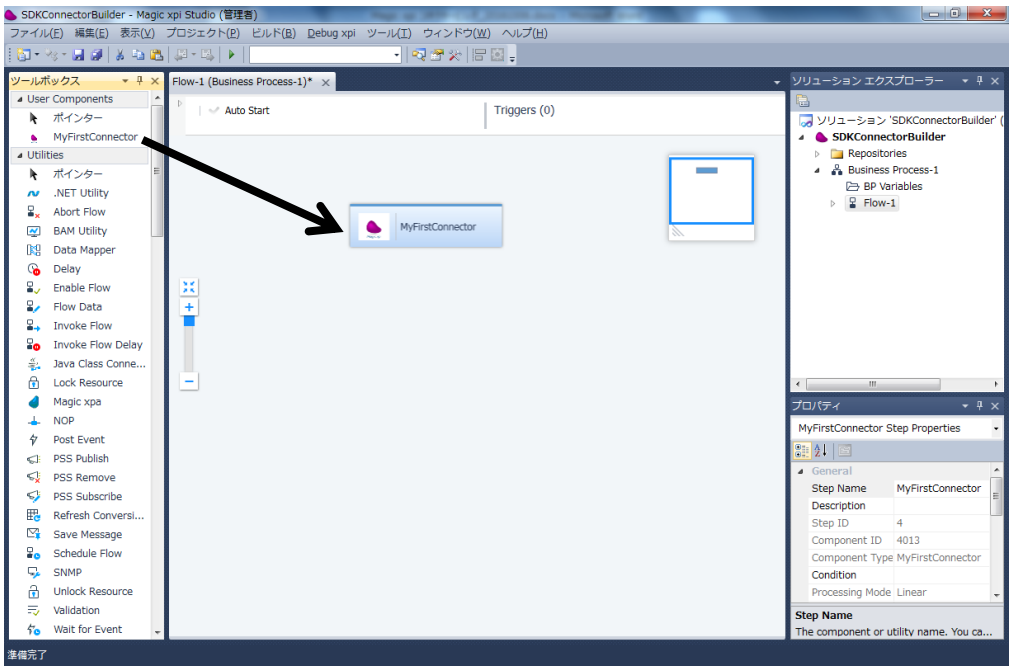


図 62：ステップ動作確認：コンポーネントの配置

配置したコンポーネントをダブルクリックして構成画面を表示します。[…]ボタンをクリックし、ランタイムの実行結果を格納する Blob 変数を選択します。例では「C.UserBlob」を選択します。その後、[OK]ボタンをクリックしマッパー画面を開きます。

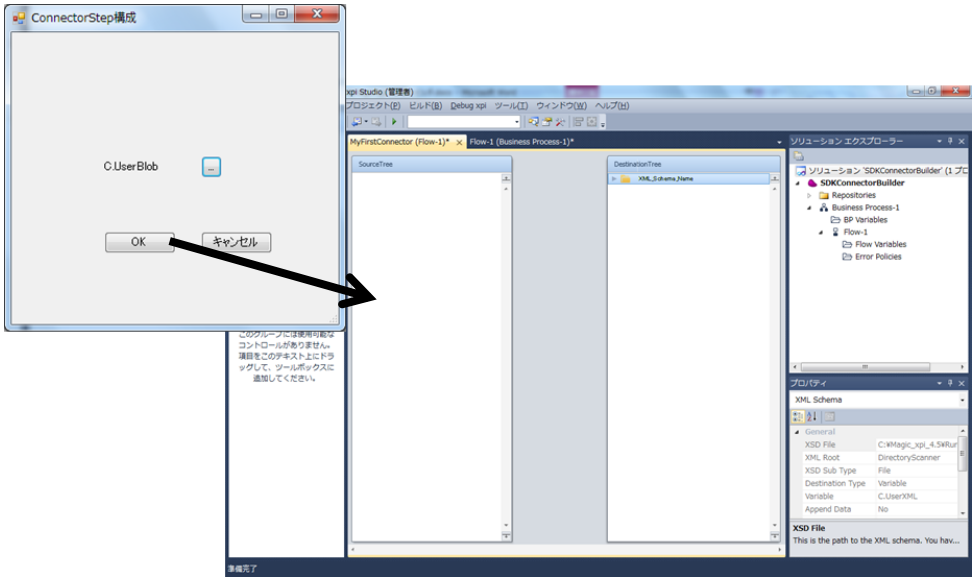


図 63：ステップ動作確認：構成画面の設定とマッパー画面の表示

マッパーの送り元となるマッパースキーマを選択します。本セクションでは FlatFile のマッパースキーマを使用します。「Mapper Schemas」グループから「Flat File」を選択し、マッパー画面左の「SourceTree」にドラッグ＆ドロップします。

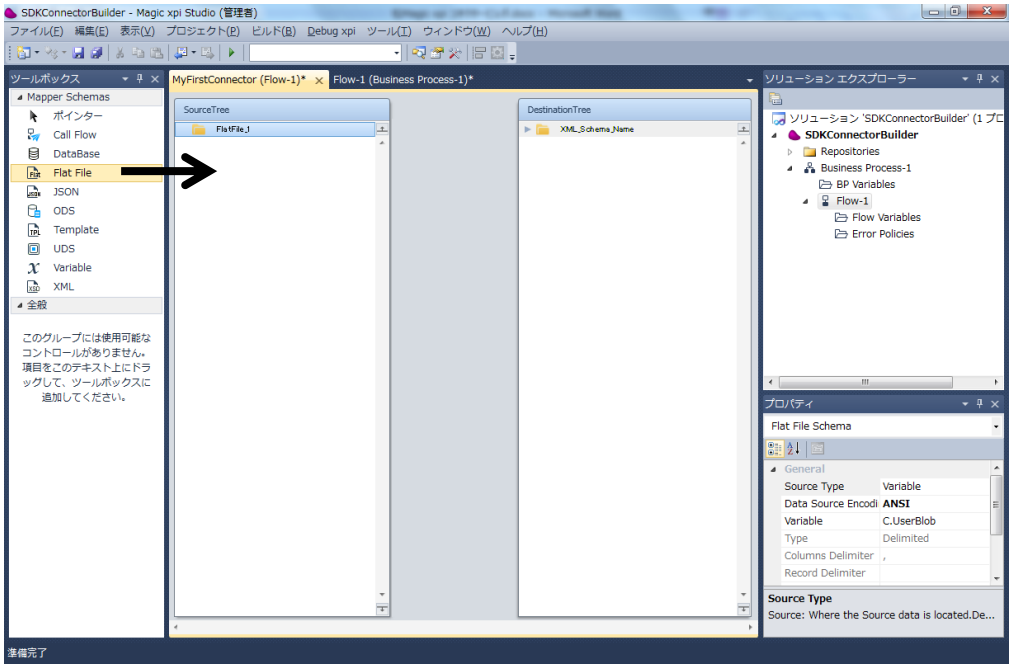


図 64：ステップ動作確認：送り元の配置

FlatFile は外部のファイルや Blob 型の変数を参照して送り元を構成します。今回は下記の外部ファイルを使用します。

参照する外部ファイル名	SDK_FF.txt
参照する外部ファイルパス	C:\temp\SDK_FF.txt
参照する外部ファイル内容	C:\temp,Step.txt,C:\temp\Destination

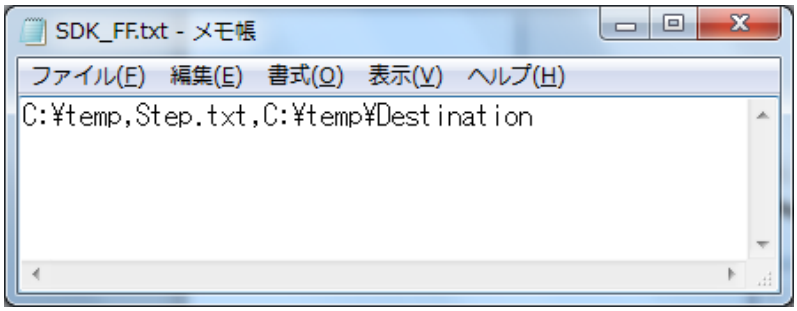


図 65：ステップ動作確認：FlatFile 参照先外部ファイルの例

外部ファイルを参照するために、FlatFile のプロパティを下記の表に従って設定します。

プロパティ名	設定値		参考画像					
Source Type	File		Source Type File					
Data Source Encoding	UTF-8		Data Source Encodi UTF-8					
File Path	'C:\temp\SDK_FF.txt' ※式エディタから入力		'C:¥temp¥SDK_FF.txt'					
Lines	右記を参照 ※Flat File エディタから定義	#	Name	Data Type	Format	From	Length	
		1	移動前ディレクトリ	Alpha	255		255	
		2	ファイル名	Alpha	255		255	

		3	移動先ディレクトリ	Alpha	255		255

上記プロパティ設定後、下記のようにマッピングを行います。

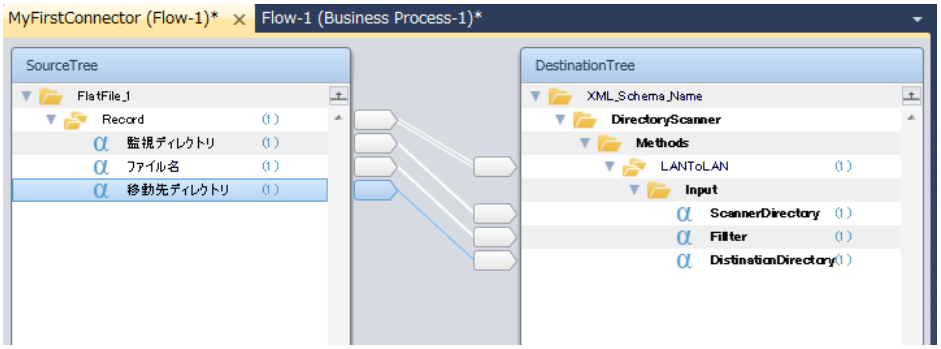


図 66：ステップ動作確認：値のマッピング

続いて、Magic xpi の実行中の変数内容を確認するため、フロー内にブレイクポイントを指定します。

画面左のツールボックスから NOP コンポーネントをドラッグ＆ドロップし、フローエリアへ配置します。配置した NOP コンポーネントを右クリックしてコンテキストメニューから「Breakpoint」をクリックします。

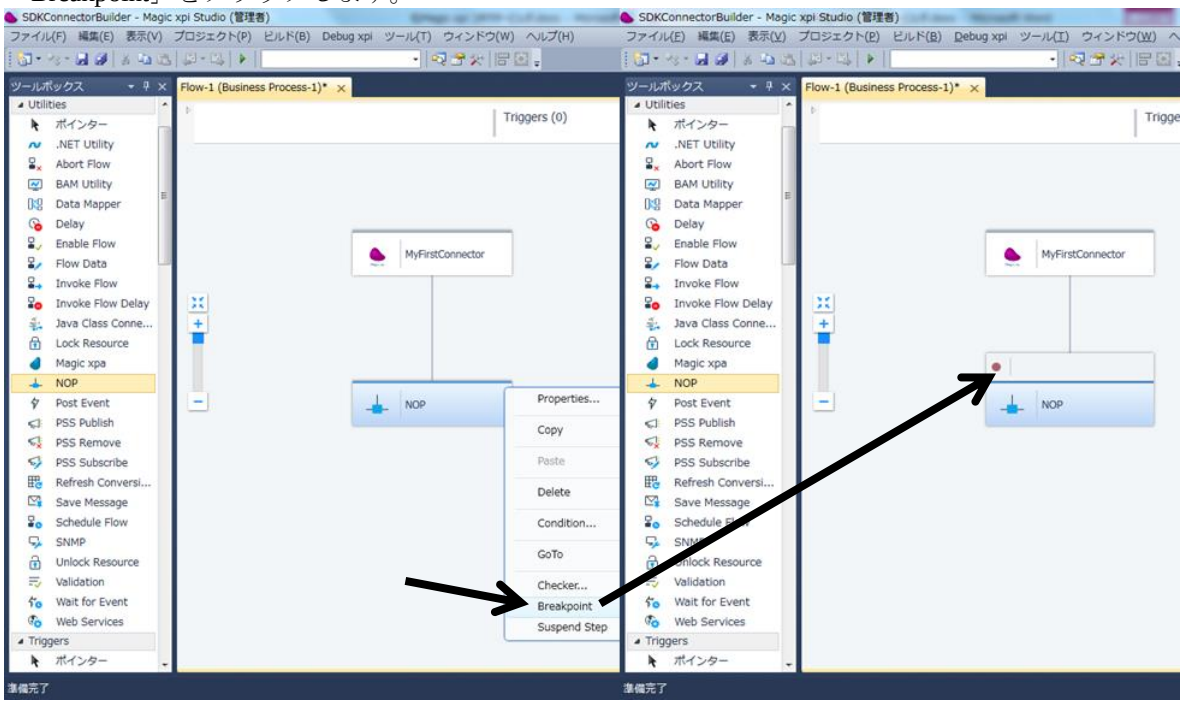


図 67：ステップ動作確認：ブレイクポイントの指定

以上で、動作確認の準備が完了しました。プロジェクトを保存後、ツールバーの実行ボタンをクリックして動作確認を開始します。

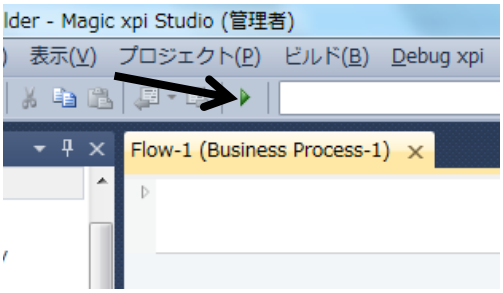


図 68：ステップ動作確認：動作確認実行

実行後、「C:\temp\Step.txt」が「C:\temp\Destination\Step.txt」へ移動していることが確認できます。

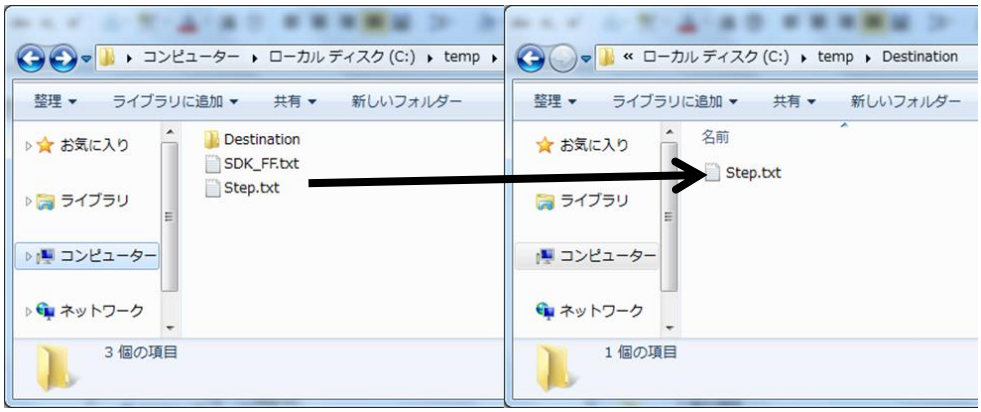


図 69：ステップ動作確認：動作確認実行後

ファイルの移動後、指定したブレイクポイントで Magic xpi の変数を確認します。

Magic xpi の画面左にある Context Tree から、ブレイクポイントを指定した「NOP」を右クリックし、コンテキストメニューから「Context View」をクリックします。

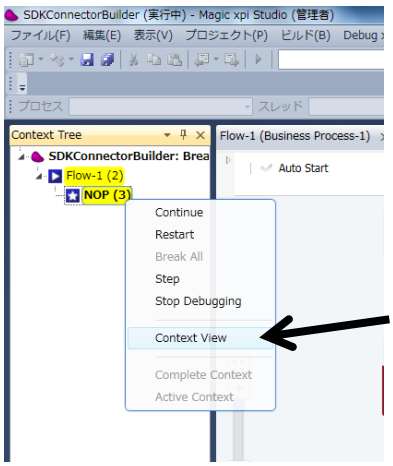


図 70：ステップ動作確認：Context View の選択

「Context View」ウィンドウを開き、ステップの戻り値を格納する変数として指定した、「C.UserBlob」の内容を確認します。変数リストから「C.UserBlob」を探し、右側の[…]ボタンをクリックします。

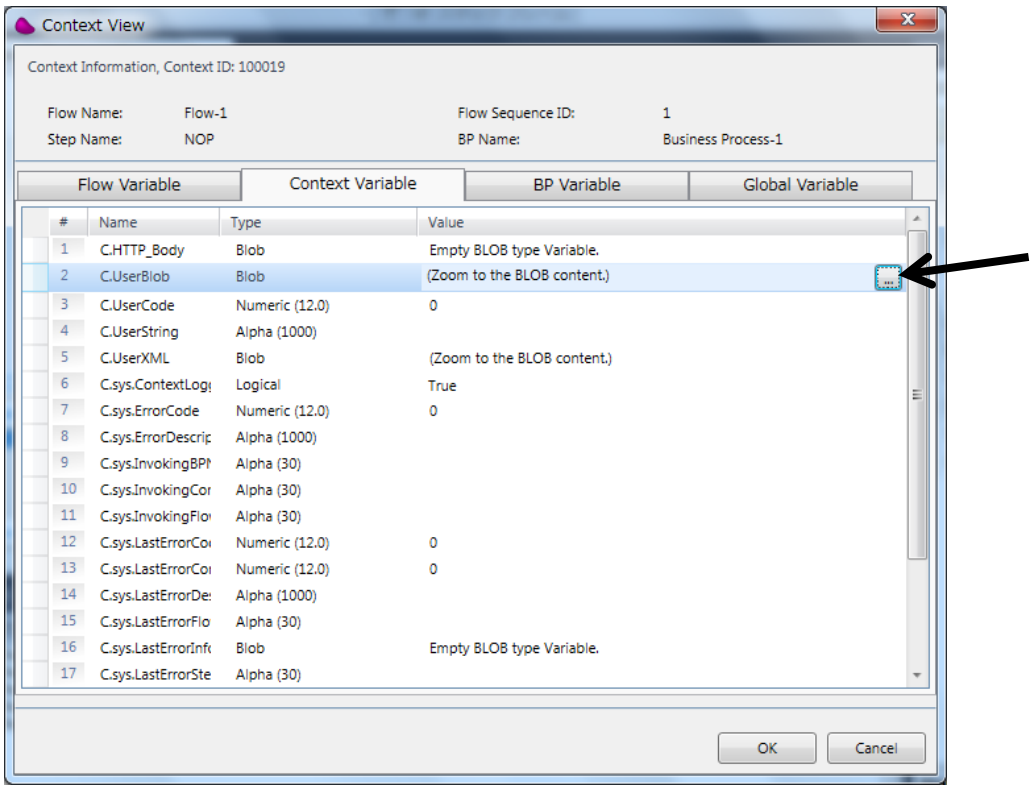


図 71：ステップ動作確認：Context View

「View Variable: C.UserBlob」ウィンドウ表示後、[Open]ボタンをクリックします。

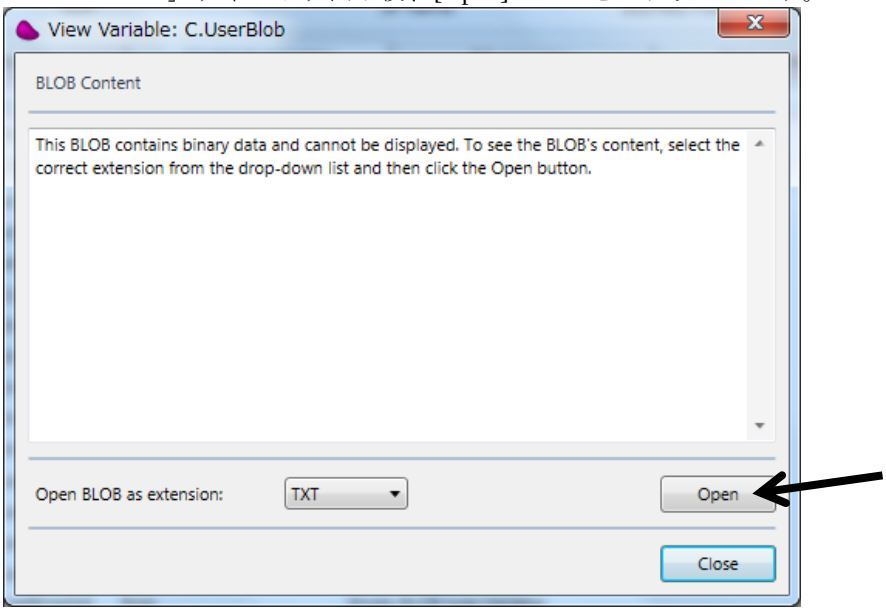


図 72：ステップ動作確認：C.UserBlob

CSV 形式で「移動前のパス」・「移動後のパス」・「移動したファイル名」が出力されていることが確認できます。

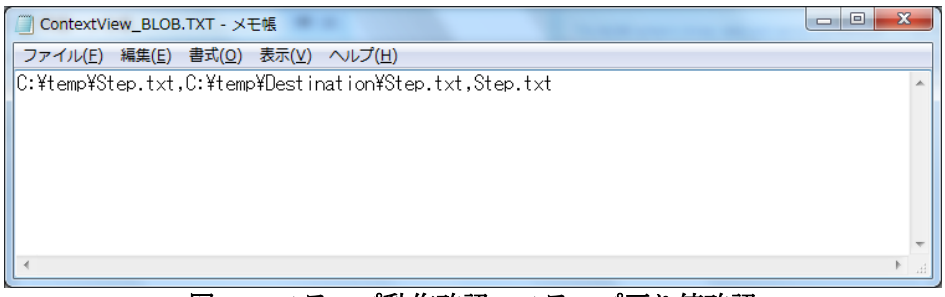


図 73：ステップ動作確認：ステップ戻り値確認

実装するトリガーUI の内容

トリガーUI カスタムコンポーネントを実装するには、以下のような手順で行います。完成した UI のイメージは下記の**図 74** を参照してください。

1. **事前準備**
2. **コンポーネントの構成を設定する UI の作成**
3. **UI の呼び出し**
4. **UI 内処理の実装**
5. **UI への入力内容の確認**
6. **UI への入力内容の再表示**
7. **実装したソース**
8. **プロジェクトのビルド**

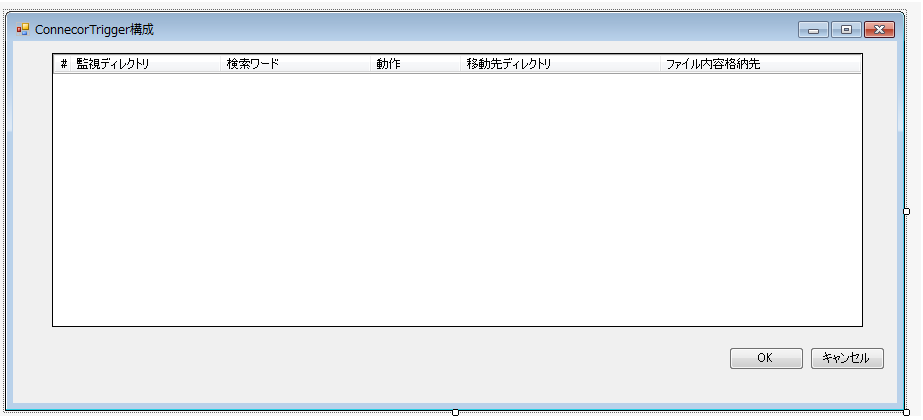


図 74：完成 UI イメージ

トリガーUI の実装

UI の実装には、トリガーUI テンプレートプロジェクトを使用します。Visual Studio Express 2015 for Desktop（以下、Visual Studio Express 2015 と示す）で、生成したトリガーUI テンプレートプロジェクト生成を開きます。

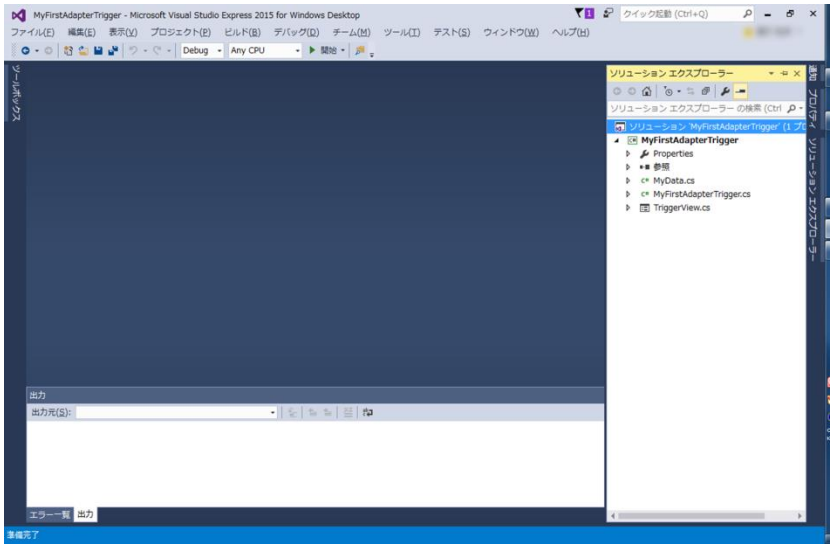


図 75：トリガー UI の作成：トリガーUI テンプレートプロジェクト生成を開く

1. 事前準備

1-1. 保持する値の定義

MyData.csで保持する値とランタイム側の値を共有するために必要な定義を行います。
MyData.csを開き、MyDataクラスに下記のプロパティを定義します。

定義するプロパティ：

```
[Id(1)]
[PrimitiveDataTypes(DataType.Blob)]
[DisplayPropertyName("inputTriggerResult")]
[TriggerIn]
public Variable inputTriggerResult { get; set; }

[Id(2)]
[PrimitiveDataTypes(DataType.Alpha)]
[DisplayPropertyName("inputDirectory")]
[UseForConfiguration]
public Alpha storeDirectory { get; set; }

[Id(3)]
[PrimitiveDataTypes(DataType.Alpha)]
[DisplayPropertyName("inputFilter")]
[UseForConfiguration]
public Alpha storeFilter { get; set; }

[Id(4)]
[PrimitiveDataTypes(DataType.Alpha)]
[DisplayPropertyName("inputDistinationDirectory")]
[UseForConfiguration]
public Alpha storeDistinationDirectory { get; set; }

[Id(5)]
[PrimitiveDataTypes(DataType.Alpha)]
[DisplayPropertyName("selectAction")]
[UseForConfiguration]
public Alpha storeAction { get; set; }
```

MyDataクラスのコンストラクタに下記のコードを追加します。

追加コード：

```
inputTriggerResult = new Variable();
inputTriggerResult.SetValue("");

MyAlphaConfig = new Alpha();
MyAlphaConfig.SetAlpha("Alpha Default Value");

storeDirectory = new Alpha();
storeDirectory.SetAlpha("");

storeFilter = new Alpha();
storeFilter.SetAlpha("");

storeDistinationDirectory = new Alpha();
storeDistinationDirectory.SetAlpha("");

storeAction = new Alpha();
storeAction.SetAlpha("");
```

実装例を以下に示します。

- MyData.cs

```
using System;
using MagicSoftware.Integration.UserComponents;

namespace SDK_TEST_1
{

   public class MyData
   {

      [Id(1)]
      [PrimitiveDataTypes(DataType.Blob)]
      [DisplayPropertyName("inputTriggerResult")]
      [TriggerIn]
      public Variable inputTriggerResult { get; set; }

      [Id(2)]
      [PrimitiveDataTypes(DataType.Alpha)]
      [DisplayPropertyName("inputDirectory")]
      [UseForConfiguration]
      public Alpha storeDirectory { get; set; }

      [Id(3)]
      [PrimitiveDataTypes(DataType.Alpha)]
      [DisplayPropertyName("inputFilter")]
      [UseForConfiguration]
      public Alpha storeFilter { get; set; }

      [Id(4)]
      [PrimitiveDataTypes(DataType.Alpha)]
      [DisplayPropertyName("inputDistinationDirectory")]
      [UseForConfiguration]
      public Alpha storeDistinationDirectory { get; set; }

      [Id(5)]
      [PrimitiveDataTypes(DataType.Alpha)]
      [DisplayPropertyName("selectAction")]
      [UseForConfiguration]
      public Alpha storeAction { get; set; }

      public MyData()
      {
```

```
            inputTriggerResult = new Variable();
            inputTriggerResult.SetValue("");

            MyAlphaConfig = new Alpha();
            MyAlphaConfig.SetAlpha("Alpha Default Value");

            storeDirectory = new Alpha();
            storeDirectory.SetAlpha("");

            storeFilter = new Alpha();
            storeFilter.SetAlpha("");

            storeDistinationDirectory = new Alpha();
            storeDistinationDirectory.SetAlpha("");

            storeAction = new Alpha();
            storeAction.SetAlpha("");

        }
    }
}
```

2. コンポーネントの構成を設定する UI の作成

2-1. フォームの作成

トリガーコンポーネントを構成する UI を作成します。UI は Visual C#の Winodws フォームを使用して作成します。プロジェクトを右クリックし、コンテキストメニューから[追加]—[新しい項目]と選択します。

図 76：トリガーUI の作成：フォームの追加

「新しい項目の追加」ウィンドウが表示されるので、「Windows Forms」アイテムの中から「Winodws フォーム」を選択します。名前は「TriggerView.cs」と入力し「追加」をクリックします。

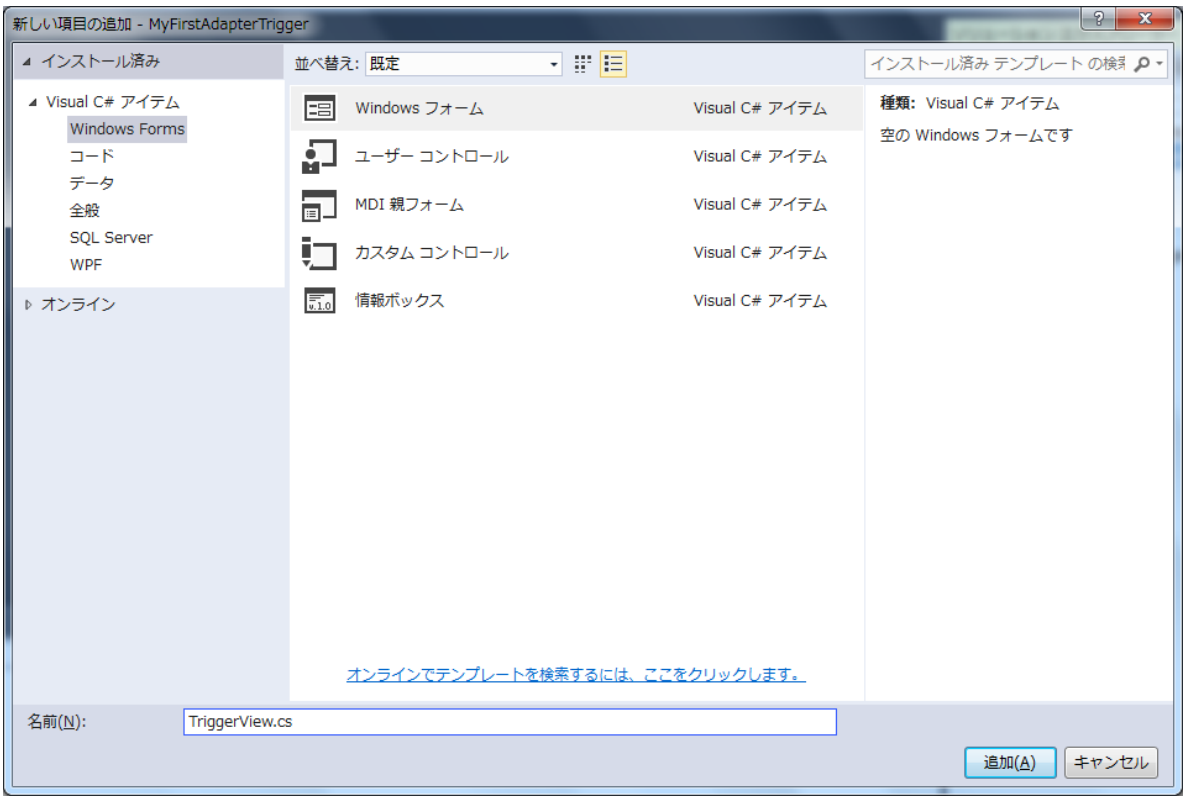
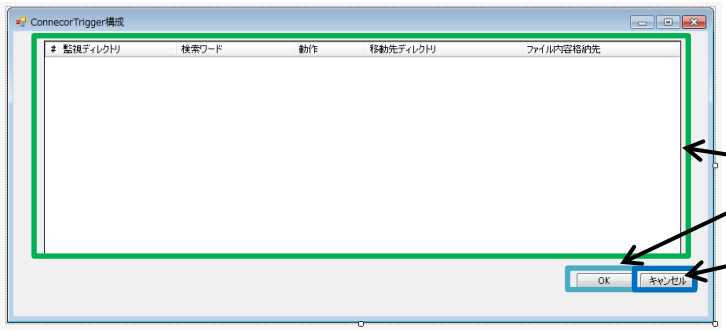


図 77：トリガーUI の作成：フォーム名

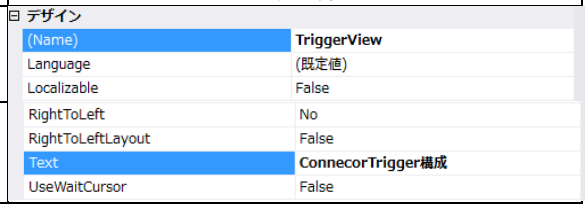
ソリューションエクスプローラに追加された TriggerView.cs をダブルクリックし、フォームを開きます。ツールボックスから「DataGridView」を１つ、「Button」を２つ、下記のように配置します。

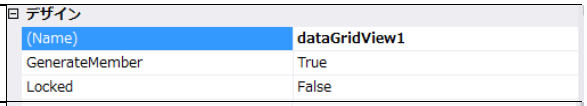
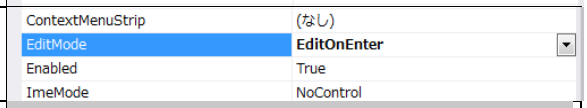
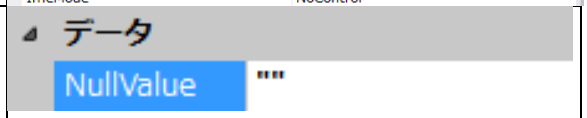
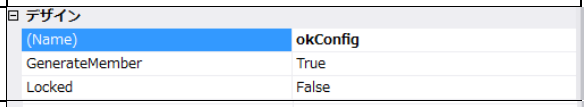
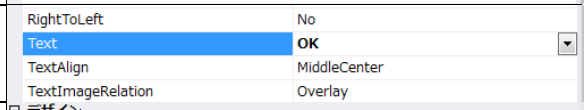
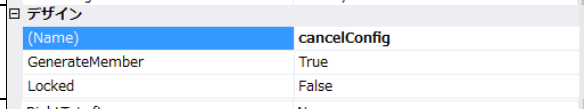
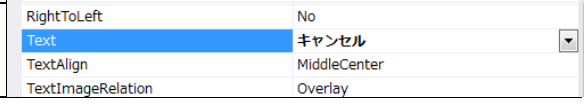


	オブジェクト名	名前
①	Form	TriggerView フォーム
②	DataGridView	dataGridView1 ビュー
③	Button	okConfig ボタン
④	Button	cancelConfig ボタン

図 78：トリガーUI の作成：オブジェクト配置

各オブジェクトに対し、以下の表のようにプロパティを編集します。

	オブジェクト名	設定値	参考画像
①	Form	デザイン-Name[TriggerView]	
		表示-Text[ConnecorTrigger 構成]	

②	DataGridView	デザイン-Name[dataGridView1]	
		動作-Text[EditOnEnter]	
		表示-RowTemplate を展開-DefaultCellStype-「…」ボタンをクリック-データ-NullValue[“”]	
③	Button	デザイン-Name[okConfig]	
		表示-Text[OK]	
④	Button	デザイン-Name[cancelConfig]	
		表示-Text[キャンセル]	

dataGridView1 ビューに、列を追加します。dataGridView1 ビューを右クリックし、コンテキストメニューから「列の編集」をクリックします。

図 79：トリガーUI の作成：dataGridView1 ビューへ列を追加（列の編集）

「列の編集」ウィンドウ表示後、画面左下にある「追加」ボタンをクリックし、下記の列名を追加します。

	列名
①	#
②	監視ディレクトリ
③	検索ワード

④	動作
⑤	移動先ディレクトリ
⑥	ファイル内容格納先

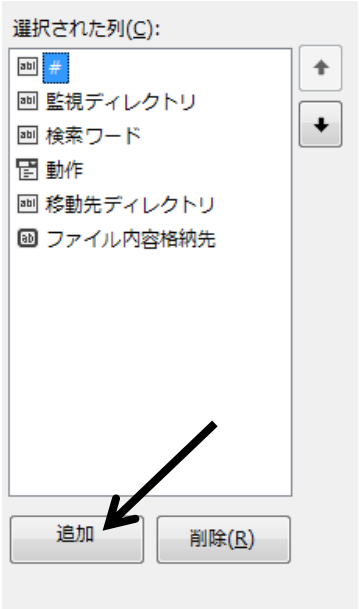
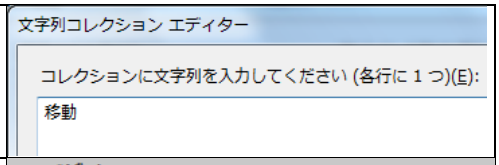
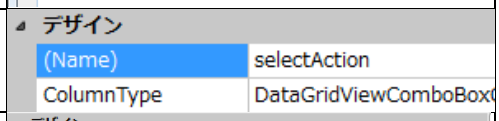
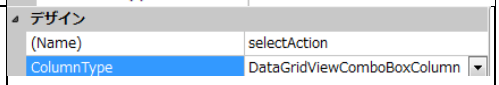
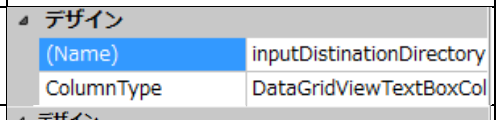
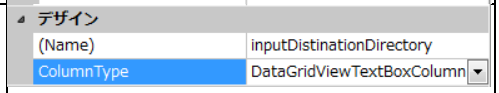
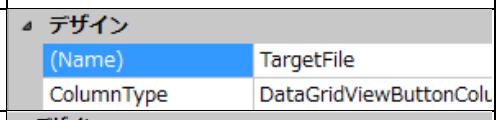
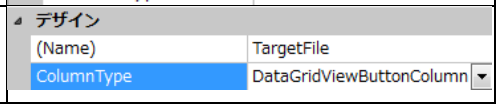


図 80：トリガーUI の作成：dataGridView1 ビューへ列を追加

各列のプロパティは、下記の表に従って編集します。

	列名	設定値	参考画像
①	#	デザイン-Name[rcdNumber]	デザイン (Name) rcdNumber ColumnType DataGridViewTextBoxColu
		デザイン-ColumnType[DataGridViewTextBoxColumn]	デザイン (Name) rcdNumber ColumnType DataGridViewTextBoxColumn
		動作-ReadOnly[True]	ContextMenuStrip (なし) MaxInputLength 32767 ReadOnly True
②	監視ディレクトリ	デザイン-Name[inputDirectory]	デザイン (Name) inputDirectory ColumnType DataGridViewTextBoxCol
		デザイン-ColumnType[DataGridViewTextBoxColumn]	デザイン (Name) inputDirectory ColumnType DataGridViewTextBoxColumn
③	検索ワード	デザイン-Name[inputFilter]	デザイン (Name) inputFilter ColumnType DataGridViewTextBoxCol
		デザイン-ColumnType[DataGridViewTextBoxColumn]	デザイン (Name) inputFilter ColumnType DataGridViewTextBoxColumn

④	動作	データ-Items[コレクション] ”移動”を追加	
		デザイン-Name[selectAction]	
		デザイン-ColumnType[DataGridViewComboBoxColumn]	
⑤	移動先ディレクトリ	デザイン-Name[inputDistinationDirectory]	
		デザイン-ColumnType[DataGridViewTextBoxColumn]	
⑥	ファイル内容格納先	デザイン-Name[TargetFile]	
		デザイン-ColumnType[DataGridViewButtonColumn]	

オブジェクトへのイベントハンドラを、以下の表に従って追加します。また、追加したイベントハンドラをそれぞれダブルクリックし、イベントハンドラメソッドを生成します。

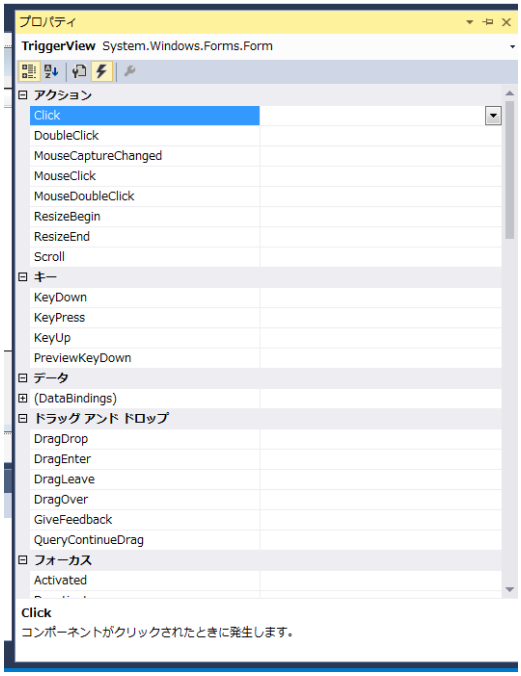
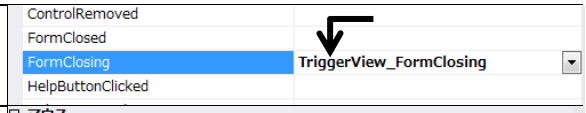
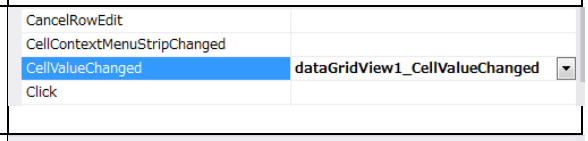
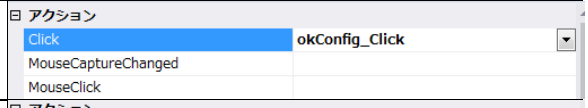
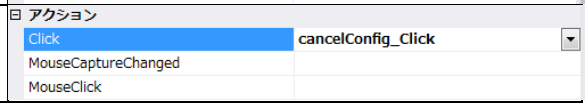


図 81：トリガーUI の作成：オブジェクトへのイベントハンドラ設定

対象オブジェクトのイベントハンドラを、以下の表に従って追加します。また、追加したイベントハンドラをそれぞれダブルクリックし、イベントハンドラメソッドを生成します。

	オブジェクト名	設定値	参考画像

ダブルクリック

①	Form	動作-FormClosing[TriggerView_FormClosing]	
②	DataGridView	マウス-CellContentClick[dataGridView1_CellContentClick]	
③	DataGridView	アクション-CellValueChanged[dataGridView1_CellValueChanged]	
④	Button	アクション-Click[okConfig_Click]	
⑤	Button	アクション-Click[cancelConfig]	

作成したフォームのソースコードを開きます。TriggerView.cs を右クリックし「コードの表示」をクリックします。

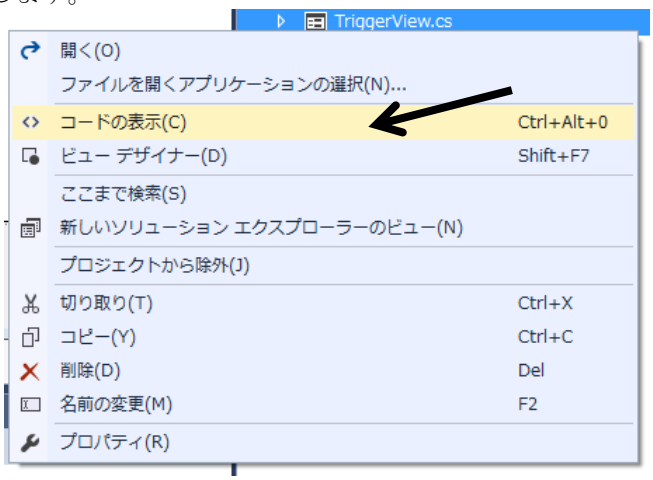


図 82：トリガーUI の作成：フォームのソースコード表示

後述する処理で、ISDKStudioUtils 型の値と MyData 型の値を受け取る必要がありため、TriggerView.cs の既存のコンストラクタに下記の引数を追加します。**※UI での変数リストの表示、式エディタの表示、値の保持を行う場合は必要になります。**

追加する引数：

```
ISDKStudioUtils utils
MyData storeMyData
```

受け取った引数をメンバ変数へ格納するため、TriggerView クラスのメンバを定義し、既存のコンストラクタに下記のコードを追加します

定義するメンバ：

```
ISDKStudioUtils studioUtils;
MyData myData;
```

追加コード：

```
this.studioUtils = utils;
this.myData = storeMyData;
```

これまでの実装例を以下に示します。

- TriggerView.cs

```
public partial class TriggerView : Form
{
        private MyData myData;
        private ISDKStudioUtils studioUtils;

        public TriggerView(ISDKStudioUtils utils,MyData storeMyData)
        {
                InitializeComponent();

                this.myData = storeMyData;
                this.studioUtils = utils;
        }

      private void TriggerView_Load(object sender, EventArgs e)
      {

      }

      private void TriggerView_FormClosing(object sender, FormClosingEventArgs e)
      {

      }

      private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
      {

      }

      private void dataGridView1_CellValueChanged(object sender, DataGridViewCellEventArgs e)
      {

      }

      private void okConfig_Click(object sender, EventArgs e)
      {

      }

      private void cancelConfig_Click(object sender, EventArgs e)
      {

      }

}
```

3. UI の呼び出し

3-1. 作成したフォームをコンポーネントの UI として呼び出し

作成した TriggerView フォームを呼び出す処理を追加します。

また、フォームで MagicXpi4.5 の変数リストの呼び出しと、フォーム上でのデータ保持を行う必要があるため、フォームを呼び出す際に、引数として ISDKStudioUtils 型変数と MyData 型変数を渡します。MyFirstAdapterTrigger.cs を開き、Configure メソッドに下記のコードを追記します。

追記コード：

```
var characteristic = new CharacteristicView(utils, adaptorData);
characteristic.ShowDialog();
```

実装例を以下に示します。

- MyFirstAdapterTrigger.cs

```
public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
 IReadOnlyServiceConfiguration serviceData, object navigateTo, out bool configurationChanged)
{

        MyData adaptorData = new MyData();
        if (dataObject != null && dataObject is MyData){
        {
                adaptorData = (dataObject as MyData);
        }
        else
        {
                dataObject = adaptorData;
        }

        var triggerView = new TriggerView(utils, adaptorData);
        triggerView.ShowDialog();
｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀｀

}
```

以上で、作成した TriggerView フォームがトリガーカスタムコンポーネントの UI として表示されるようになりました。

4. UI 内処理の実装

4-1. dataGridView1 ビュー

[ファイル内容格納先]列をクリックし、変数リストを表示して変数を選択します。
dataGridView1_CellContentClick メソッドに下記のコードを追記します。

追加コード：

```
DataGridView dgv = (DataGridView)sender;

if (dgv.Columns[e.ColumnIndex].Name == "TargetFile")
{
        myData.inputTriggerResult = studioUtils.OpenVariablePicklist
                (myData.inputTriggerResult
                , VariableFilter.ALLVariables, DataType.Blob);

        dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex].Value
        = myData.inputTriggerResult.GetValue();
}
```

実装例を以下に示します。

```
private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
{
        DataGridView dgv = (DataGridView)sender;

        if (dgv.Columns[e.ColumnIndex].Name == "TargetFile")
        {
                myData.inputTriggerResult  = studioUtils.OpenVariablePicklist
                        (myData.inputTriggerResult, VariableFilter.ALLVariables, DataType.Blob);

                dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex].Value = myData.inputTriggerResult.GetValue();
        }
}
```

続いて、構成内容に変更が加えられたことを保存します。

TriggerView.cs を開き、TriggerView クラスに下記のメンバを定義します。

定義するメンバ：

```
private bool configurationChanged = false;
```

dataGridView1_CellValueChanged メソッドに下記のコードを追加します。
追加コード：

```
this.configurationChanged = true;
```

確認した判定値を別クラスが受け取れるように、下記のメソッドを追加します。
追加メソッド：

```
public bool isConfigurationChanged
{
    get
    {
        return this.configurationChanged;
    }
}
```

これまでの実装例を下記に示します。

- TriggerView.cs

```
public partial class TriggerView : Form
{
    MyData myData;
    ISDKStudioUtils studioUtils;
    private bool configurationChanged = false;

    private void dataGridView1_CellValueChanged(object sender, DataGridViewCellEventArgs e)
    {
        this.configurationChanged = true;
    }

    public bool isConfigurationChanged
    {
        get
        {
            return this.configurationChanged;
        }
    }

}
```

4-2. okConfig ボタン
dataGridView1 ビューへの入力値を保持します。

okConfig ボタンをクリックした際に、DataGridView1 ビュー内の[監視ディレクトリ]列、[検索ワード] 列、[動作] 列、[移動先ディレクトリ] 列への入力値を保持します。

okConfig_Click メソッドに下記のコードを追記します。
追加コード：

```
myData.storeDirectory.SetAlpha(dataGridView1.Rows[0].Cells[1].Value.ToString());
myData.storeFilter.SetAlpha(dataGridView1.Rows[0].Cells[2].Value.ToString());
myData.storeAction.SetAlpha(dataGridView1.Rows[0].Cells[3].Value.ToString());
myData.storeDistinationDirectory.SetAlpha(dataGridView1.Rows[0].Cells[4].Value.ToString());

Dispose();
```

実装例を以下に示します。

- TriggerView.cs

```
private void okConfig_Click(object sender, EventArgs e)
{
        myData.storeDirectory.SetAlpha(dataGridView1.Rows[0].Cells[1].Value.ToString());
        myData.storeFilter.SetAlpha(dataGridView1.Rows[0].Cells[2].Value.ToString());
        myData.storeAction.SetAlpha(dataGridView1.Rows[0].Cells[3].Value.ToString());
        myData.storeDistinationDirectory.SetAlpha(dataGridView1.Rows[0].Cells[4].Value.ToString());

        Dispose();
}
```

4-3. cancelConfig ボタン

構成内容への変更を破棄します。TriggerView.cs のソースコードを開きます。

cancelConfig_Click メソッドに下記のコードを追加します。

追加コード：

```
this.configurationChanged = false;

Dispose();
```

実装例を下記に示します。

- TriggerView.cs

```
private void cancelConfig_Click(object sender, EventArgs e)
{
this.configurationChanged = false;

Dispose();
}
```

4-4. TriggerView フォーム

TriggerView.cs を開き、TriggerView クラスに下記のメンバを定義します。

定義するメンバ：

```
private bool configurationSuccess= false;
```

TriggerView_FormClosing メソッドに下記のコードを追加します。

追加コード：

```
if (myData.inputTriggerResult.GetValue().Length == 0)
{
   MessageBox.Show("ファイルの格納先が不正です。",
   "エラー",
   MessageBoxButtons.OK,
   MessageBoxIcon.Error);
}

this.configurationSuccess = true;
```

続いて判定値を別クラスが受け取れるように、下記のメソッドを追加します。

追加メソッド：

```
public bool isConfigurationSuccess
{
      get
      {
           return this.configurationSuccess;
```

```
    }
}
```

これまでの実装例を下記に示します。

- CharacteristicView.cs

```
public partial class TriggerView : Form
{
        MyData myData;
        ISDKStudioUtils studioUtils;
        private bool configurationChanged = false;
        private bool configurationSuccess = false;

        private void TriggerView_FormClosing(object sender, FormClosingEventArgs e)
        {

        if (myData.inputTriggerResult.GetValue().Length == 0)
        {
                MessageBox.Show("ファイルの格納先が不正です。",
                "エラー",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }

        public bool isConfigurationSuccess
        {
                get
                {
                        return this.configurationSuccess;
                }
        }

        this.configurationSuccess = true;
}
```

5. UI への入力内容の確認

5-1. 入力内容の確認

MyFirstAdapterTrigger.cs 内の Configure メソッド戻り値が[False]だった場合は、直前に変更した内容は保持されません。逆に[True]だった場合は、値が保持されません。

なお、本カスタムコンポーネントでは、TriggerView が閉じる際に、列名[ファイル内容格納先]が正しく指定されているか判定を行います。正しくなかった場合は、エラーメッセージを表示します。また、別列は値を保持したいので、必ず[True]を戻り値としています。

MyFirstAdapterTrigger.cs の Configure メソッドの戻り値に対して、下記のように記述して構成内容が正しいか判定した値の判定結果を受け取ります。

記述コード：

```
return characteristic.isConfigurationSuccess;
```

実装例を下記に示します。

- MyFirstAdapterTrigger.cs

```
public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
        IReadOnlyServiceConfiguration serviceData,
        object navigateTo, out bool configurationChanged)
{
```

```
        var triggerView = new TriggerView(utils, adaptorData);
        triggerView.ShowDialog();

        configurationChanged =true;

        return triggerView.isConfigurationSuccess;

    }
```

5-2. 入力内容に変更があるかを確認

Magic xpi では、フロー内容に変更が加えられた場合、フロータブの末尾に「*」が表示されるようになっています。構成内容に変更があった場合も、その「*」を表示する必要があります。

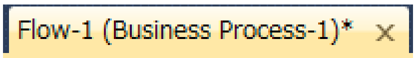


図 83：トリガーUI の作成：フロー内容変更時に表示される「*」

MyFirstAdapterTrigger.cs の Configure メソッドにある、bool 型の configurationChanged 変数が「*」を表示するフラグの役割を果たしています。

・configurationChanged = True の場合は「*」が付く
・configurationChanged = False の場合は「*」が付かない

TriggerView フォームから変更の有無を判定した値を MyFirstAdapterTrigger.cs で取得するため、下記のコードを記述します。

記述コード：

```
configurationChanged = characteristic.isConfigurationChanged;
```

実装例を下記に示します。

●MyFirstAdapterTrigger.cs

```
public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
    IReadOnlyServiceConfiguration serviceData,
    object navigateTo, out bool configurationChanged)
{
、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、
        var triggerView = new TriggerView(utils, adaptorData);
        triggerView.ShowDialog();

        configurationChanged = triggerView.isConfigurationChanged;

        return triggerView.isConfigurationSuccess;
、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、、
}
```

6. UI への入力内容の再表示

6-1. 前回 UI に保持した値を、再度表示

UI を開く際に、保持してきた値を再度 TriggerView フォームのオブジェクトへ設定する必要があります。

TriggerView.cs のソースコードを開き、TriggerView_Load メソッド内に下記のコードを追加します。

追加コード：

```
dataGridView1.Rows.Insert(0, "1");

if (dataGridView1.RowCount > 0)
{
        dataGridView1.Rows[0].Cells[1].Value = myData.storeDirectory.GetAlpha();
        dataGridView1.Rows[0].Cells[2].Value = myData.storeFilter.GetAlpha();
        dataGridView1.Rows[0].Cells[3].Value = myData.storeAction.GetAlpha();
        dataGridView1.Rows[0].Cells[4].Value = myData.storeDistinationDirectory.GetAlpha();
        dataGridView1.Rows[0].Cells[5].Value = myData.inputTriggerResult.GetValue();
}
```

実装例を下記に表示します。

- TriggerView.cs

```
private void TriggerView_Load(object sender, EventArgs e)
{
        dataGridView1.Rows.Insert(0, "1");

        if (dataGridView1.RowCount > 0)
        {
                dataGridView1.Rows[0].Cells[1].Value = myData.storeDirectory.GetAlpha();
                dataGridView1.Rows[0].Cells[2].Value = myData.storeFilter.GetAlpha();
                dataGridView1.Rows[0].Cells[3].Value = myData.storeAction.GetAlpha();
                dataGridView1.Rows[0].Cells[4].Value = myData.storeDistinationDirectory.GetAlpha();
                dataGridView1.Rows[0].Cells[5].Value = myData.inputTriggerResult.GetValue();
        }
}
```

7. 実装したソース

これまでに実装した MyData.cs、TriggerView.cs、MyFirstAdapterTrigger.cs のソースコード全体を以下に示します。

- MyData.cs

```
using System;
using MagicSoftware.Integration.UserComponents;

namespace SDK_TEST_1
{

   public class MyData
   {

      [Id(1)]
      [PrimitiveDataTypes(DataType.Blob)]
      [DisplayPropertyName("inputTriggerResult")]
      [TriggerIn]
      public Variable inputTriggerResult { get; set; }

      [Id(2)]
      [PrimitiveDataTypes(DataType.Alpha)]
      [DisplayPropertyName("inputDirectory")]
      [UseForConfiguration]
      public Alpha storeDirectory { get; set; }

      [Id(3)]
      [PrimitiveDataTypes(DataType.Alpha)]
      [DisplayPropertyName("inputFilter")]
      [UseForConfiguration]
      public Alpha storeFilter { get; set; }

      [Id(4)]
      [PrimitiveDataTypes(DataType.Alpha)]
      [DisplayPropertyName("inputDistinationDirectory")]
```

```
        [UseForConfiguration]
        public Alpha storeDistinationDirectory { get; set; }

        [Id(5)]
        [PrimitiveDataTypes(DataType.Alpha)]
        [DisplayPropertyName("selectAction")]
        [UseForConfiguration]
        public Alpha storeAction { get; set; }

        public MyData()
        {

            inputTriggerResult = new Variable();
            inputTriggerResult.SetValue("");

            storeDirectory = new Alpha();
            storeDirectory.SetAlpha("");

            storeFilter = new Alpha();
            storeFilter.SetAlpha("");

            storeDistinationDirectory = new Alpha();
            storeDistinationDirectory.SetAlpha("");

            storeAction = new Alpha();
            storeAction.SetAlpha("");

        }
    }
}
```

- TriggerView.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using MagicSoftware.Integration.UserComponents.Interfaces;
using MagicSoftware.Integration.UserComponents;

namespace SDK_TEST_1
{
    public partial class TriggerView : Form
    {
        MyData myData;
        ISDKStudioUtils studioUtils;
        private bool configurationChanged = false;
        private bool configurationSuccess = false;

        public TriggerView(ISDKStudioUtils utils,MyData storeMyData)
        {
            InitializeComponent();

            this.myData = storeMyData;
            this.studioUtils = utils;


        }

        private void TriggerView_Load(object sender, EventArgs e)
        {
            dataGridView1.Rows.Insert(0, "1");]

            if (dataGridView1.RowCount > 0)
```

```
    {
        dataGridView1.Rows[0].Cells[1].Value = myData.storeDirectory.GetAlpha();
        dataGridView1.Rows[0].Cells[2].Value = myData.storeFilter.GetAlpha();
        dataGridView1.Rows[0].Cells[3].Value = myData.storeAction.GetAlpha();
        dataGridView1.Rows[0].Cells[4].Value = myData.storeDistinationDirectory.GetAlpha();
        dataGridView1.Rows[0].Cells[5].Value = myData.inputTriggerResult.GetValue();
    }
}

private void TriggerView_FormClosing(object sender, FormClosingEventArgs e)
{

    if (myData.inputTriggerResult.GetValue().Length == 0)
    {
        MessageBox.Show("ファイルの格納先が不正です。",
        "エラー",
        MessageBoxButtons.OK,
        MessageBoxIcon.Error);
    }

    this.configurationSuccess = true;
}

private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
{

    DataGridView dgv = (DataGridView)sender;

    if (dgv.Columns[e.ColumnIndex].Name == "TargetFile")
    {
        myData.inputTriggerResult
            = studioUtils.OpenVariablePicklist
            (myData.inputTriggerResult, VariableFilter.ALLVariables, DataType.Blob);

        dataGridView1.Rows[e.RowIndex].Cells[e.ColumnIndex].Value
            = myData.inputTriggerResult.GetValue();
    }
}

private void dataGridView1_CellValueChanged(object sender, DataGridViewCellEventArgs e)
{
    this.configurationChanged = true;
}

private void okConfig_Click(object sender, EventArgs e)
{

    myData.storeDirectory.SetAlpha(dataGridView1.Rows[0].Cells[1].Value.ToString());
    myData.storeFilter.SetAlpha(dataGridView1.Rows[0].Cells[2].Value.ToString());
    myData.storeAction.SetAlpha(dataGridView1.Rows[0].Cells[3].Value.ToString());
    myData.storeDistinationDirectory.SetAlpha(dataGridView1.Rows[0].Cells[4].Value.ToString());

    Dispose();
}

private void cancelConfig_Click(object sender, EventArgs e)
{
    this.configurationChanged = false;

    Dispose();
}

public bool isConfigurationChanged
{
    get
    {
        return this.configurationChanged;
    }
}
```

```
        public bool isConfigurationSuccess
        {
            get
            {
                return this.configurationSuccess;
            }
        }
    }
}
```

- MyFirstAdapterTrigger.cs

```
using System;
using System.Collections.Generic;
using MagicSoftware.Integration.UserComponents.Interfaces;
using MagicSoftware.Integration.UserComponents;
using System.ComponentModel.Composition;
using System.Windows.Forms;

namespace SDK_TEST_1
{

    [Export(typeof(IUserTriggerComponent))]
    public class MyFirstAdapterTrigger : IUserTriggerComponent
    {
        public MyFirstAdapterTrigger()
        {
        }

        #region IUserTriggerComponent implementation

        public object CreateDataObject()
        {
            return new MyData();
        }

        public bool? Configure(ref object dataObject, ISDKStudioUtils utils,
            IReadOnlyServiceConfiguration serviceData,
            object navigateTo, out bool configurationChanged)
        {

            MyData adaptorData = new MyData();
            if (dataObject != null && dataObject is MyData)
            {
                adaptorData = (dataObject as MyData);
            }
            else
            {
                dataObject = adaptorData;
            }

            var triggerView = new TriggerView(utils, adaptorData);
            triggerView.ShowDialog();

            configurationChanged = triggerView.isConfigurationChanged;

            return triggerView.isConfigurationSuccess;

        }


        #endregion

        public ICheckerResult Check(ref object data, IReadOnlyServiceConfiguration serviceData)
        {
            return null;
        }
```

```
        public bool ValidateService(IReadOnlyServiceConfiguration serviceData, out string errorMsg)
        {

            errorMsg = null;

            return true;

        }
        public void InvokeServiceHelper(string helperID, IServiceConfiguration serviceData)
        {

        }
    }
}
```

8. **プロジェクトのビルド**

以上で UI の作成は終了です。Visual Studio Express 2015 のメニューバーから[ビルド]—[ソリューションのリビルド]を選択し、ビルドを行います。

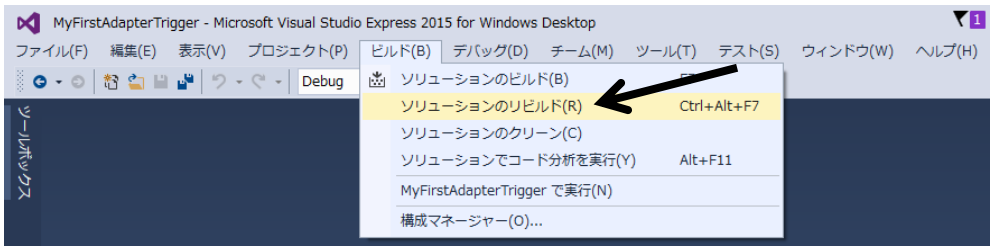


図 84：トリガーUI の作成：UI クラスのビルド

下記のコピー元・コピー先を参照し、ビルド後に生成されるファイルをコピーしてください。

コピー元：
・[UI クラスのプロジェクトフォルダ]\ MyFirstAdapterTrigger\bin\Debug\MyFirstAdapterTrigger.dll
・[UI クラスのプロジェクトフォルダ]\ MyFirstAdapterTrigger\bin\Debug\MyFirstAdapterTrigger.pdb

コピー先：
・[コンポーネントフォルダ] \ui\lib\MyFirstAdapterTrigger.dll
・[コンポーネントフォルダ] \ui\lib\MyFirstAdapterTrigger.pdb

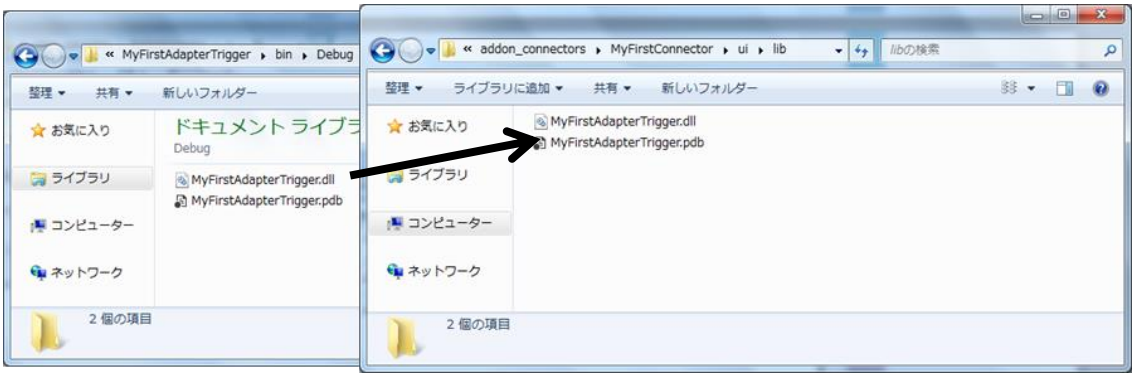


図 85：トリガーUI の作成：UI クラスのビルド生成物のコピー

実装するトリガーランタイムの内容

1. **ログ出力の設定**
2. **UI に入力された内容から、移動するファイルを選定・移動**
3. **移動したファイル内容を Magic xpi の変数へ格納**
4. **実装したソース**
5. **ランタイムクラスの Jar ファイル生成**

ランタイムの実装には、トリガーランタイムテンプレートプロジェクトを使用します。Eclipse 4.6 Neon Pleiades All in One（以下、Eclipse 4.6 と示す）で、生成したトリガーランタイムテンプレートプロジェクトを開きます。

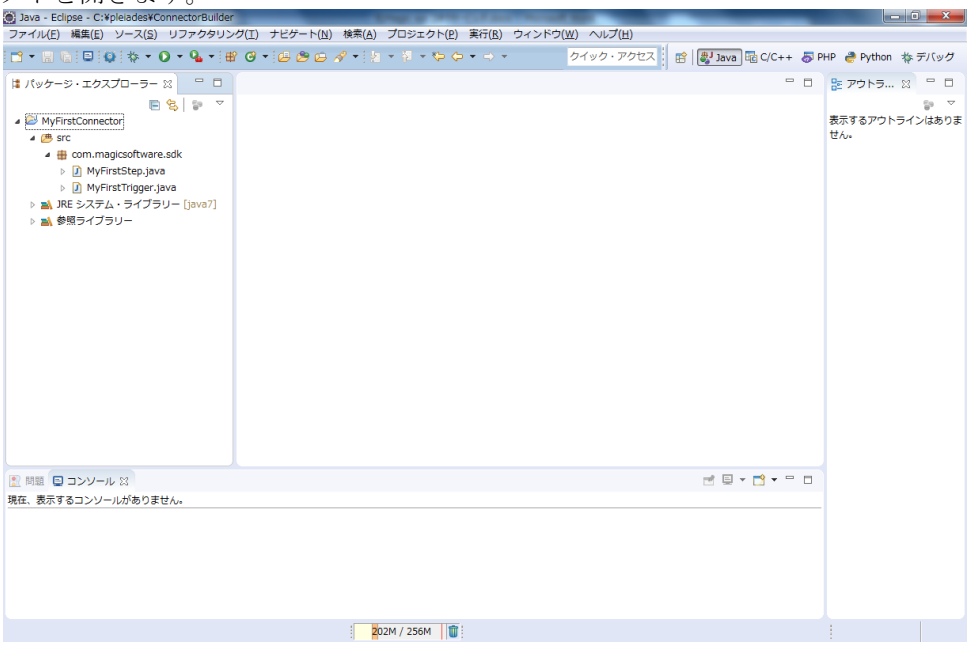


図 86：トリガーランタイムの作成：トリガーランタイムテンプレートプロジェクトを開く

トリガーランタイムの実装

1. **ログ出力の設定**

1-1. ログ出力の設定

実装するコンポーネントごとにログ出力をするための設定を行います。Magic xpi をインストールした際に下記のフォルダに「log4j.xml」が生成されているので、そのファイルを編集します。

log4j.xml 生成先：
< Magic xpi インストールディレクトリ>\Runtime\java\classes\log4j.xml

1-2. log4j.xml の編集

log4j.xml の START MAGIC_INFO APPENDERS セクションの末尾に下記のように追記します

追記内容：

```
<appender name="MyFirstAdapter-Trigger-appender" class="org.apache.log4j.RollingFileAppender">
        <param name="Threshold" value="&fileDefThreshold;"/>
        <param name="File" value="${com.magicsoftware.ibolt.home}/logs/java/mytriggerlog_${pid}.log"/>
        <param name="MaxFileSize" value="&fileSize;"/>
        <param name="MaxBackupIndex" value="&fileBackups;"/>
        <layout class="org.apache.log4j.PatternLayout">
```

```
                <param name="ConversionPattern" value="&XpiPattern;"/>
        </layout>
</appender>
```

続いて、START MAGIC_INFO LOGGERS セクションの末尾に下記のように追記します。

```
<logger name="com.magicsoftware.sdk.MyFirstTrigger" additivity="false">
        <level value="debug"/>
        <appender-ref ref="MyFirstAdapter-Trigger-appender"/>
</logger>
```

1-3. Log4j の定義

下記のように編集し、コンポーネントごとにログ出力するようにします。

編集前：

```
private final static Logger log = Logger.getLogger("MyTriggerLoggerName");
```

編集後：

```
private final static Logger log = Logger.getLogger("com.magicsoftware.sdk.MyFirstTrigger");
```

2. UI に入力された内容から、移動するファイルを選定・移動

2-1. TriggerView フォームに入力された内容の取得

フォームに入力した値を、ランタイムで受け取るためには、UI 作成時に定義した、データクラスである MyData.cs 内のメンバ名が関係してきます。

例：MyData.cs 内のメンバ名

```
public class MyData
{
        [Id(番号)]
        [属性名]
        public メンバ型名 メンバ名 { get; set; }
}
```

実装例を以下に示します。

- MyFirstTrigger.java

```
public class MyFirstTrigger implements IExternalTrigger{
`````````````````````````````````````````````````````````````````````
     @Override
     public void load(TriggerGeneralParams generalParams, FlowLauncher fl) throws SDKException {

          UserProperty upDirectory = generalParams.getConfigurations().get("storeDirectory");
          UserProperty upFilter = generalParams.getConfigurations().get("storeFilter");
          final UserProperty upDistinationDirectory = generalParams.
                    getConfigurations().get("storeDistinationDirectory");
          UserProperty upAction = generalParams.getConfigurations().get("storeAction");
     }
`````````````````````````````````````````````````````````````````````
}
```

2-2. TriggerView フォームから取得した値でファイルの移動

MyFirstTrigger.java の Load メソッドに、下記の定数とファイルを移動する処理を実装します。

追加定数：

```
private final static String MOVE_SELECT_ACTION = "移動";
```

実装する処理の概要：

1. TriggerView フォームから取得した値から、移動対象となるファイルを特定し、ファイルを移動。
2. 3 秒ごとに[1. ]を実行。

実装する処理：

```
@Override
public void load(TriggerGeneralParams generalParams, FlowLauncher fl) throws SDKException {
	try{
		launcher=fl;

		UserProperty upDirectory = generalParams.getConfigurations().get("storeDirectory");
		UserProperty upFilter = generalParams.getConfigurations().get("storeFilter");
		final UserProperty upDistinationDirectory = generalParams.getConfigurations()
				.get("storeDistinationDirectory");
		UserProperty upAction = generalParams.getConfigurations().get("storeAction");

/**************************↓コードを追加*********************************************/
		final String filter = upFilter.getValue().toString();

		final FilenameFilter filterFilePath = new FilenameFilter() {
			@Override
			public boolean accept(File dir, String name) {
				if(name.matches(filter.replace("*", ".*").replace("..", "."))){
					return true;
				} else {
					return false;
				}
			}
		};

		final File directory = new File(upDirectory.getValue().toString());
		final String action = upAction.getValue().toString();

		timer = new Timer();

		timer.scheduleAtFixedRate(new TimerTask() {
			@Override
			public void run() {

				File[] filterFilePathList = directory.listFiles(filterFilePath);

				if(MOVE_SELECT_ACTION.equals(action)){

					for(File moveFilePath : filterFilePathList) {

						File distinationDirectory= new File(upDistinationDirectory.getValue().toString()
							+ "\\" + moveFilePath.getName());

						if (moveFilePath.isFile()) {

							if(moveFilePath.renameTo(distinationDirectory)){

							}
						}
					}
				}
			}
		}, 3*1000, 3*1000);
	} catch(Exception e) {
		throw new SDKException(100, "Error:"+e.toString());
	}
/*****************************************************************************************/
}
```

### 3. 移動したファイル内容を Magic xpi の変数へ格納

トリガー機能としてフローを起動する処理を実装します。call メソッドを呼び出すことでフローを起動できます。起動と同時に、フローへ値を渡したい場合には、call メソッドに渡したい引数を指定するなどの方法で、値を共有する必要があります。

ランタイムから Magic xpi の変数へ値を渡すには、UI 作成時に定義した MyData.cs 内のメンバ名が関係してきます。Magic xpi の変数へ渡すには、Variable 型のメンバ名を使用します。

例：MyData.cs 内のメンバ名

```
public class MyData
{
        [Id(番号)]
        [属性名]
        public Variable メンバ名 { get; set; }
}
```

本トリガーコンポーネントでは、移動したファイル内容を Blob 変数としてフローに渡します。Blob 変数には byte[]型で渡す必要があるため、call メソッドの引数に byte[]型の引数を指定します。
MyFirstTrigger.java を開き、call メソッドを下記のように編集、実装します。

編集後の call メソッドの処理概要：

1. 引数として受け取ったファイル内容を、ファイル内容格納先である inputTriggerResult にセット
2. フロー呼び出しとともに、1. でセットした値を Magic xpi の変数へ渡す

- MyFirstTrigger.java

```
private void call(byte[] fileContent){

        HashMap<String, Object> args = new HashMap<String, Object>();
        args.put("inputTriggerResult", fileContent);

        try {
                Response result = launcher.invoke(args);

                if(result!=null)
                {
                        UserProperty userP=result.getData("inputTriggerResult");
                        String s=new String(((byte [] )userP.getValue()));

                        log.info("フローへ渡した値："+s);
                }

        } catch (Exception e) {
                        log.error("フローの呼び出しに失敗しました。");
        }
}
```

次に、load メソッドでファイル移動に成功した直後に、フローを呼び出す必要があります。下記の「フロー呼び出し処理追加」を参照し、load メソッドにコードを追加します。

フロー呼び出し処理追加：

```
timer.scheduleAtFixedRate(new TimerTask() {
        @Override
        public void run() {
```

```
            File[] filterFilePathList = directory.listFiles(filterFilePath);

                if(MOVE_SELECT_ACTION.equals(action)){

                    for(File moveFilePath : filterFilePathList) {

                        File distinationDirectory= new File(upDistinationDirectory.getValue().toString()
                                + "\\" + moveFilePath.getName());

                        if (moveFilePath.isFile()) {

/*************************↓コードを追加*********************************************/
                            byte[] fileContentBytes = null;
                            try {
                                fileContentBytes = Files. readAllBytes(Paths.get(moveFilePath.getPath()));
                            } catch (Exception e) {

                            };
/************************************************************************************/

                            if(moveFilePath.renameTo(distinationDirectory)){
/*******************************↓コードを追加*************************************/
                                call(fileContentBytes);
/************************************************************************************/

                            }
                        }
                    }
                }
            }
        }, 3*1000, 3*1000);
```

## 4. 実装したソース

これまでに実装した MyFirstStep.java のソースコード全体を以下に示します。

- MyFirstStep.java

```
package com.magicsoftware.sdk;

import java.io.File;
import java.io.FilenameFilter;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.HashMap;
import java.util.Timer;
import java.util.TimerTask;

import org.apache.log4j.Logger;

import com.magicsoftware.xpi.sdk.SDKException;
import com.magicsoftware.xpi.sdk.UserProperty;
import com.magicsoftware.xpi.sdk.trigger.TriggerGeneralParams;
import com.magicsoftware.xpi.sdk.trigger.external.FlowLauncher;
import com.magicsoftware.xpi.sdk.trigger.external.IExternalTrigger;
import com.magicsoftware.xpi.sdk.trigger.external.Response;

public class MyFirstTrigger implements IExternalTrigger{

        private final static Logger log = Logger.getLogger("com.magicsoftware.sdk.MyFirstTrigger");

        private final static String MOVE_SELECT_ACTION = "移動";

        private FlowLauncher launcher;
        private Timer timer;

        @Override
        public void disable() {
        }

        @Override
        public void enable() {
```

```
}
@Override
public void load(TriggerGeneralParams generalParams, FlowLauncher fl) throws SDKException {

	try{
		launcher=fl;

		UserProperty upDirectory = generalParams.getConfigurations().get("storeDirectory");
		UserProperty upFilter = generalParams.getConfigurations().get("storeFilter");
		final UserProperty upDistinationDirectory =
			generalParams.getConfigurations().get("storeDistinationDirectory");
		UserProperty upAction = generalParams.getConfigurations().get("storeAction");
		final String filter = upFilter.getValue().toString();

		final FilenameFilter filterFilePath = new FilenameFilter() {
			@Override
			public boolean accept(File dir, String name) {
				if(name.matches(filter.replace("*", ".*").replace("..", "."))){
					return true;
				} else {
					return false;
				}
			}
		};

		final File directory = new File(upDirectory.getValue().toString());
		final String action = upAction.getValue().toString();

		timer = new Timer();

		timer.scheduleAtFixedRate(new TimerTask() {
			@Override
			public void run() {

				File[] filterFilePathList = directory.listFiles(filterFilePath);

				if(MOVE_SELECT_ACTION.equals(action)){

					for(File moveFilePath : filterFilePathList) {

						File distinationDirectory
						= new File(upDistinationDirectory
						.getValue().toString()
						+ "\\" + moveFilePath.getName());

						if (moveFilePath.isFile()) {

							byte[] fileContentBytes = null;
							try {
								fileContentBytes = Files
								.readAllBytes(Paths.get
								(moveFilePath.getPath()));
							} catch (Exception e) {

							};

							if(moveFilePath.renameTo
								(distinationDirectory)){
								call(fileContentBytes);
							}
						}
					}
				}
			}, 3*1000, 3*1000);

	} catch(Exception e) {
		throw new SDKException(100, "Error:"+e.toString());
	}
}

@Override
public void unload() {

}

private void call(byte[] fileContent){

	HashMap<String, Object> args = new HashMap<String, Object>();
```

```
                    args.put("inputTriggerResult", fileContent);
                    try {
                            Response result = launcher.invoke(args);

                            if(result!=null)
                            {
                                    UserProperty userP=result.getData("inputTriggerResult");

                                    String s=new String(((byte [] )userP.getValue()));

                                    log.info("フローへ渡した値："+s);

                            }
                    } catch (Exception e) {
                            log.error("フローの呼び出しに失敗しました。");
                    }
            }
}
```

5. **ランタイムクラスの Jar ファイル生成**

以上でランタイムの作成は終了です。トリガーランタイムテンプレートプロジェクトを右クリックし、コンテキストメニューからエクスポートを選択します。

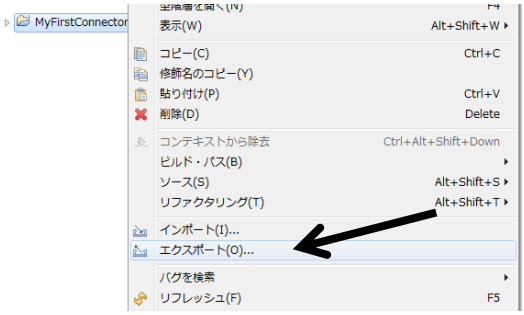



**図 87：トリガーランタイムの作成：JAR ファイルエクスポート 1**

エクスポートウィンドウ表示後、[Java]—[JAR ファイル]と選択し[次へ]ボタンをクリックします。

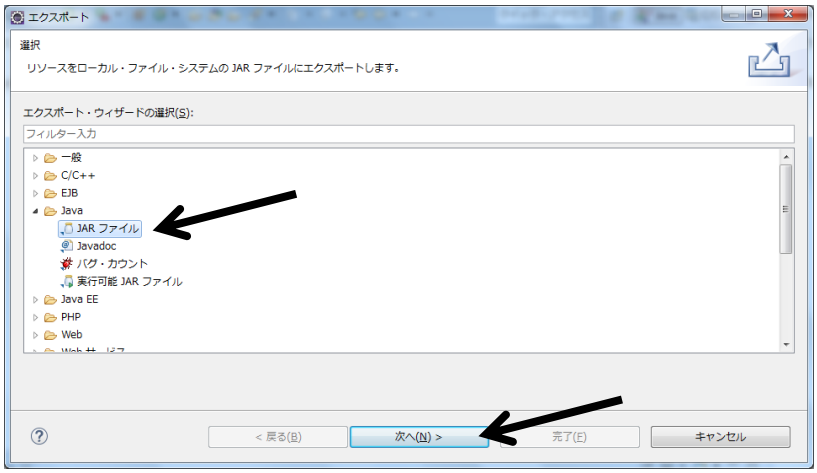



**図 88：トリガーランタイムの作成：JAR ファイルエクスポート 2**

JAR エクスポートウィンドウが表示後、下記を参照しエクスポート先指定します。指定完了後[完了]ボタンをクリックします。

エクスポート先：
[コンポーネントフォルダ] \runtime\java\lib\MyFirstConnector.jar

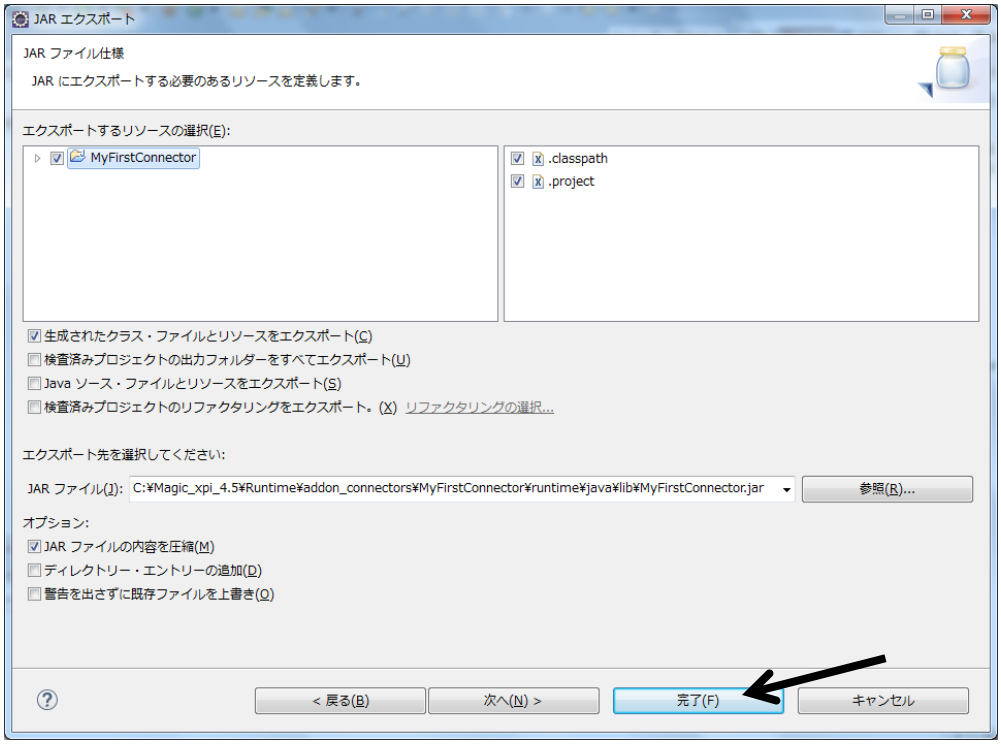



**図 89：トリガーランタイムの作成：JAR ファイルエクスポート 3**

## 実装したトリガーの動作確認

1. 動作確認に必要な事前準備
   動作確認の事前準備として下記の表に沿い、任意のディレクトリ、任意のファイルを作成します。

   動作確認で使用する例：

| 種類 | パス | 使用用途 |
|---|---|---|
| ディレクトリ | C:\temp | 移動前ディレクトリ |
| ディレクトリ | C:\temp\Destination | 移動先ディレクトリ |
| ファイル | C:\temp\Trigger.txt | 移動対象ファイル |

   上記表内の「移動対象ファイル」のファイル内容を、任意に編集します。例では、下記のようにファイル内容を編集します。

   ファイル内容：
   Trigger 動作確認用ファイル

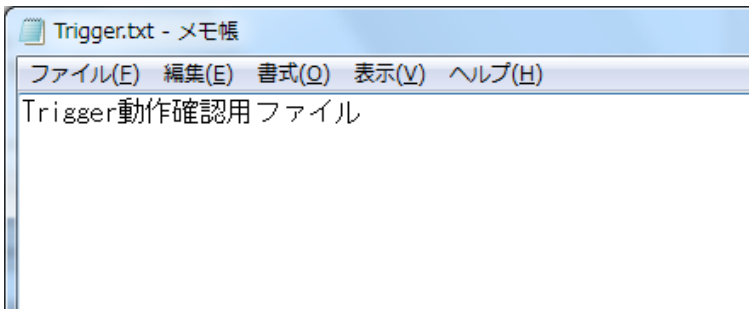



**図 90：トリガーランタイムの作成：移動対象ファイル内容例**

2. 動作確認の実施

これまで実装したコンポーネントを実際に Magic xpi で実行して動作確認を行います。Magic xpi を起動し、新しいプロジェクトを作成します。その後、「Flow-1」というフローが自動で追加されます。

作成した「MyFirstConnector」コンポーネントが画面左ツールボックス内の「User Conponents」グループに表示されています。「MyFirstConnector」コンポーネントをトリガーエリアへドラッグ＆ドロップします。

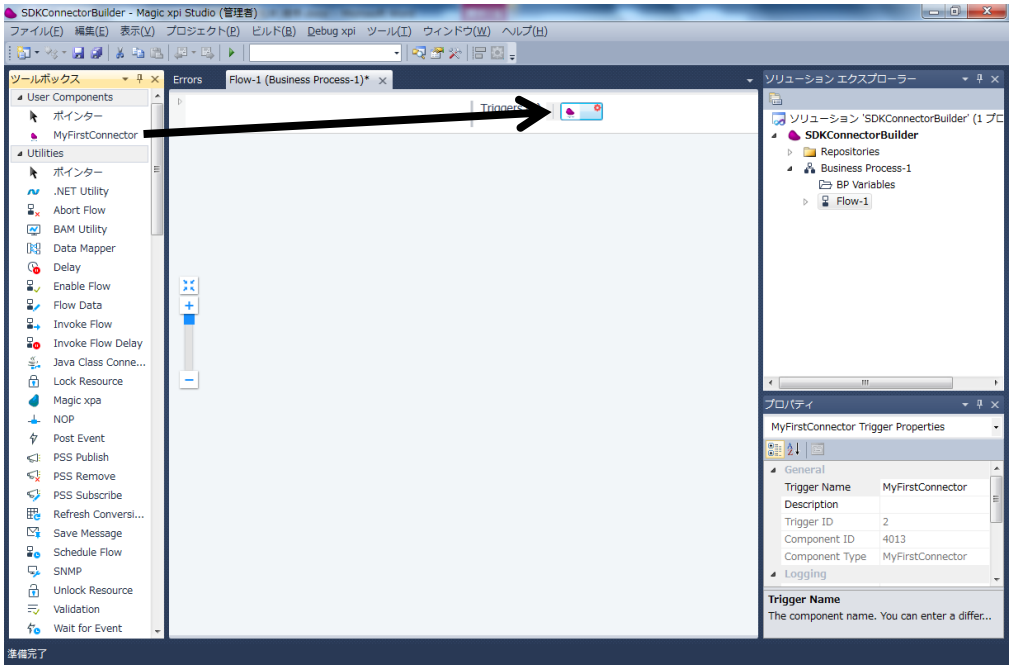



**図 91：トリガー動作確認 コンポーネントの配置**

配置したコンポーネントをダブルクリックして構成画面を表示します。下記の表に従って、各項目に入力します。入力後、「OK」ボタンをクリックし、構成画面を閉じます。

構成画面入力内容：

| | |
|---|---|
| 監視ディレクトリ | 任意のディレクトリ（例では「C:\temp」を入力） |
| 検索ワード | 任意のファイル名（例では「Trigger.txt」を入力） |
| 動作 | 移動 |
| 移動先ディレクトリ | 任意のディレクトリ（例では「C:\temp\Destination」を入力） |

| | |
|---|---|
| ファイル内容格納先 | C.UserBlob |

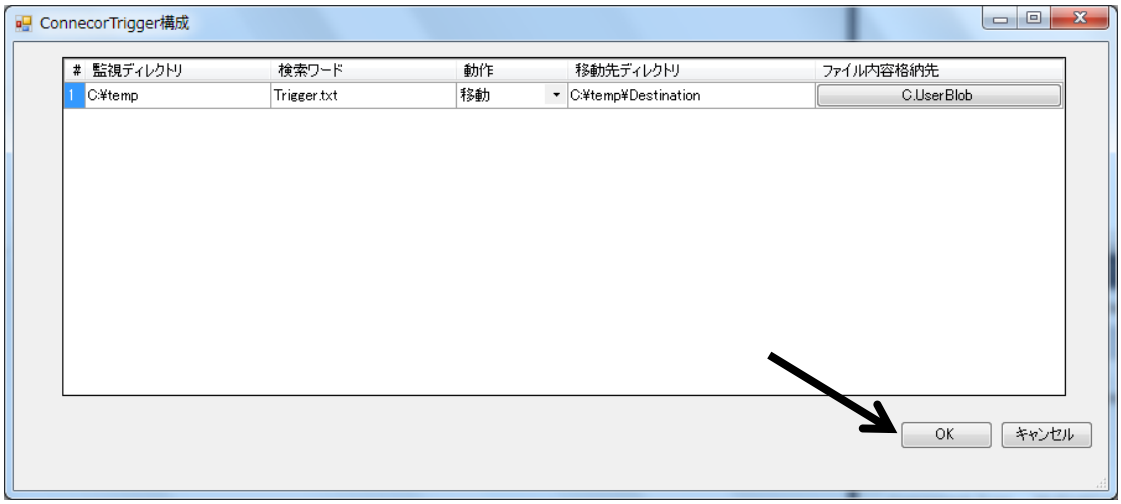



**図 92：トリガー動作確認：コンポーネントの構成画面の入力例**

続いて、Magic xpi の変数内容を確認するため、フロー内にブレイクポイントを指定します。

画面左のツールボックスから NOP コンポーネントをドラッグ＆ドロップし、フローエリアへ配置します。配置した NOP コンポーネントを右クリックし、コンテキストメニューから「Breakpoint」をクリックします。

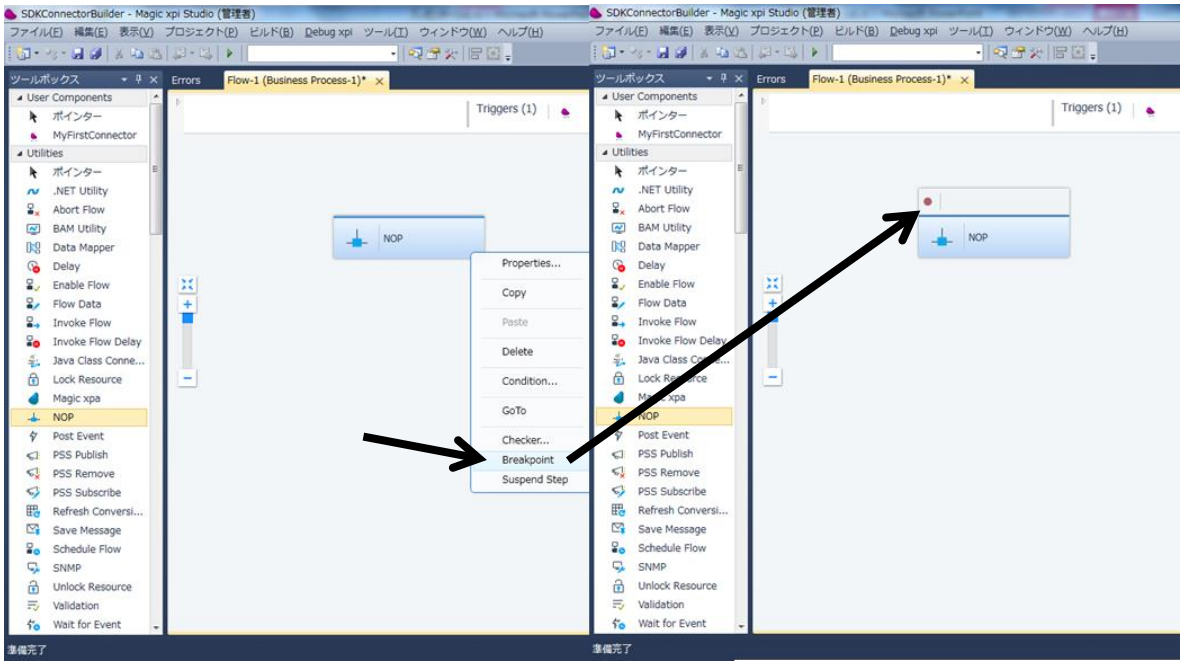



**図 93：トリガー動作確認：ブレイクポイントの指定**

以上で、動作確認の準備が完了しました。プロジェクトを保存後、ツールバーの実行ボタンをクリックし、動作確認を開始します。

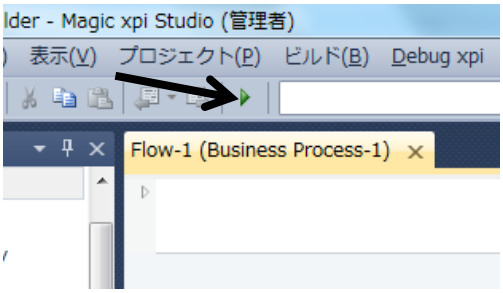



**図 94：トリガー動作確認：動作確認実行**

実行後、「C:\temp\Trigger.txt」が「C:\temp\Destination\Trigger.txt」へ移動していることが確認できます。

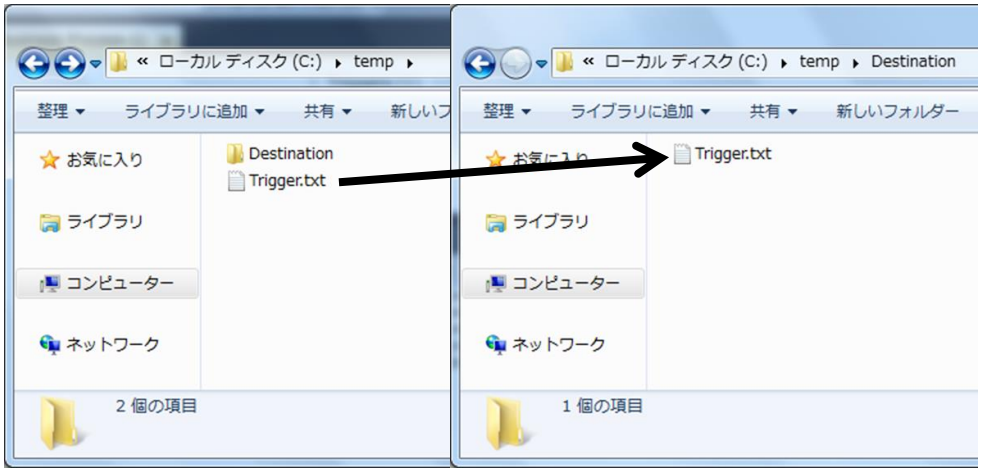



**図 95：トリガー動作確認：動作確認実行後**

ファイル移動後、指定したブレイクポイントで Magic xpi の変数内容を確認します。先ほど指定したブレイクポイントで処理が停止しているので、ここで Magic xpi 内の変数内容を確認します。

Magic xpi の画面左にある Context Tree から、ブレイクポイントを指定した「NOP」を右クリックします。コンテキストメニューから「Context View」をクリックします。

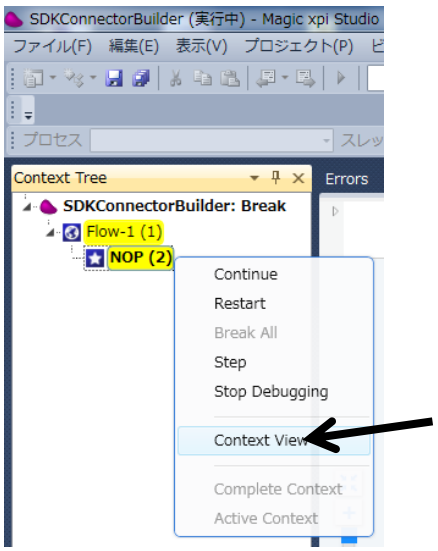



**図 96：トリガー動作確認：Context View の選択**

「Context View」ウィンドウを開き、トリガーで移動したファイル内容格納先の変数として指定した「C.UserBlob」の内容を確認します。変数リストから「C.UserBlob」を探し、[…]ボタンをクリックします。

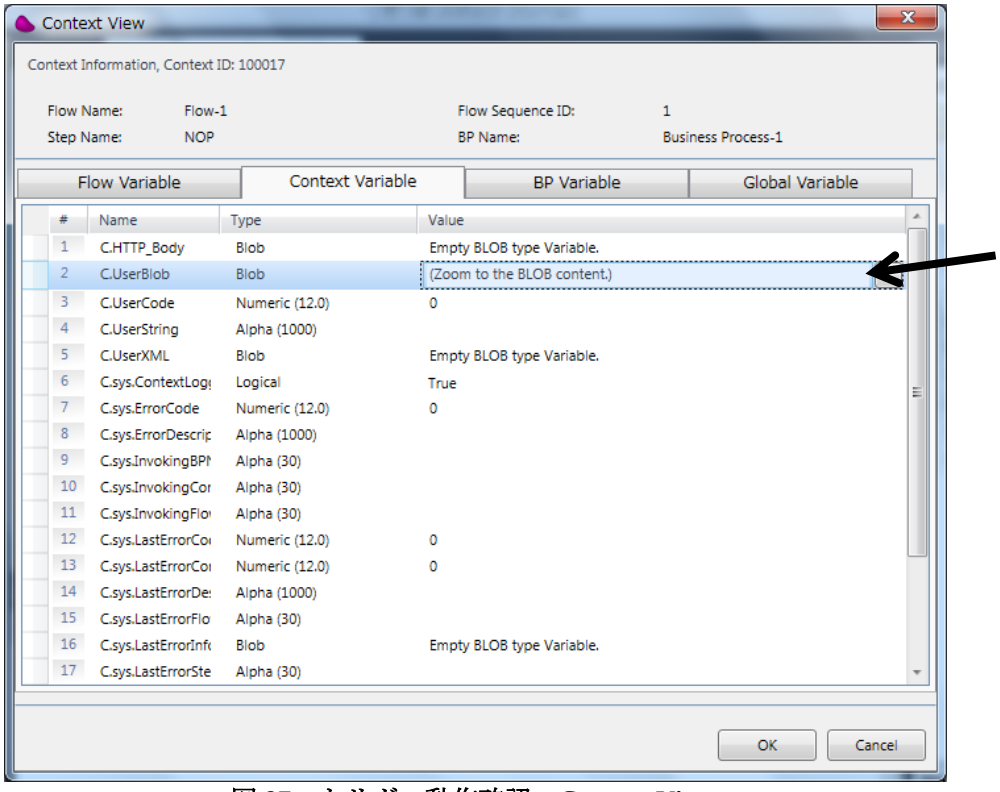



**図 97：トリガー動作確認：Context View**

「View Variable: C.UserBlob」ウィンドウ表示後、[Open]ボタンをクリックします。

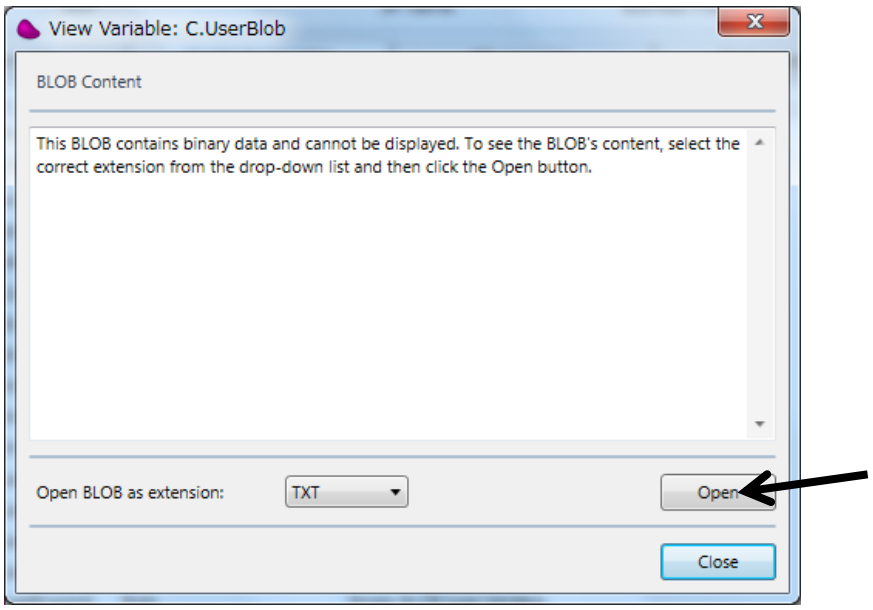



**図 98：トリガー動作確認：C.UserBlob**

テキストエディタが開きます。移動対象となったファイルファイル内容を取得できていることが確認できます。

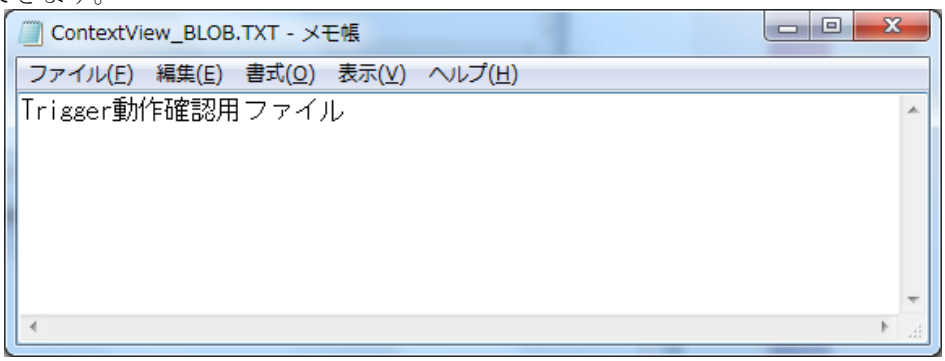



**図 99：トリガー動作確認：ステップ戻り値確認**

このまま実行を終了せずに、移動対象となるファイルを移動前ディレクトリに再配置することで、再度フローが起動します。

# About Magic Software Enterprises

Magic Software Enterprises (NASDAQ: MGIC) empowers customers and partners around the globe with smarter technology that provides a multi-channel user experience of enterprise logic and data.

We draw on 30 years of experience, millions of installations worldwide, and strategic alliances with global IT leaders, including IBM, Microsoft, Oracle, Salesforce.com, and SAP, to enable our customers to seamlessly adopt new technologies and maximize business opportunities.

For more information, visit www.magicsoftware.com.