VBMP マニュアル(翻訳文書)

1. Overview／概要

2. Using VB Migration Partner／VB Migration Partner を使用する

3. Converting Language Elements／言語要素を変換する

4. Advanced Topics／高度なトピック

5. Pragma Reference／プラグマリファレンス

Appendix A. ADOLibrary／付録 A ADOLibrary

1. Overview／概要

1. Overview／概要

VB Migration Partner is a tool that converts VB6 applications to VB.NET. It matches or exceeds the features of the conversion and assessment tools included in Microsoft Visual Studio 2005 or 2008, available on Microsoft's site, or provided by other vendors, and is aimed at both the developer and the team manager that needs to plan the migration process. Release 1.0 generates both VB2005 and VB2008 applications.

VB Migration Partner は VB6 アプリケーションを VB.NET アプリケーションに変換するツールです。それは、Microsoft Visual Studio 2005 または 2008 に同梱されていたり、Microsoft のウェブサイトから入手できたり、他社から提供されたりしている変換/評価ツール以上の機能を持っており、移行工程を計画しなければならない開発者とプロジェクト

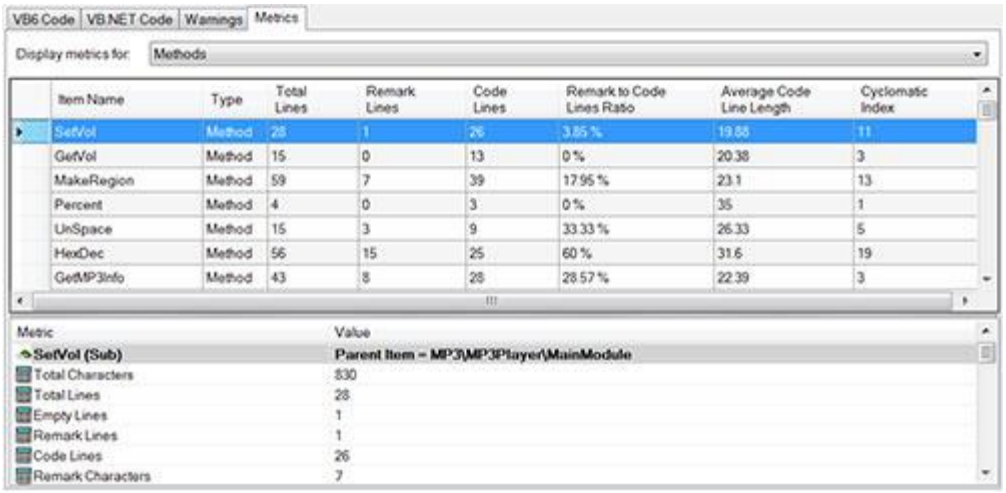
マネージャーの両者のために開発されました。リリース 1.0 では VB2005 と VB2008 の両方のアプリケーションを生成します。

VB Migration Partner's engine is so fast that VB6 developers can use it to see where the problematic code sections, have a draft version of the VB.NET application, and produce an estimation of the time required to complete the migration process, all in a fraction of the time needed to run the Upgrade Wizard tool included in Microsoft Visual Studio.

VB Migration Partner のエンジンは非常に速いので、疑わしいソースコードを見つけたり、ドラフト版の VB.NET アプリケーションを作成したり、移行工程を完了するために必要な工数見積作成など、すべての作業を Microsoft Visual Studio に含まれている Upgrade Wizard を実行する為に必要な時間の一部で行うことができます。

At the end of the migration process VB Migration Partner produces accurate reports about the problems it found together with metrics about the code being migrated. These reports include estimations of the time required to migrate the VB6 application and individual projects or classes. Reports also include sophisticated code metrics, such as total and average cyclomatic index, maximum and average depth of control structures, ratio of comments to code, in addition to a summary of all the migration issues found by the parser engine. The cost related to these metrics and issues (in terms of time and money) is fully configurable, and users can export metrics to Microsoft Excel for further analysis.

移行工程の終わりに VB Migration Partner は移行されたソースコードに関する測定値と共に見つかった問題に関する正確なレポートを生成します。これらのレポートは、VB6 アプリケーションと個々のプロジェクト、そしてクラスを移行するために必要な工数の見積も含まれます。レポートは洗練されたコードメトリクス、サイクロマティックインデックス)の総和および平均、制御構造の深さの最大と平均、コメント行の割合に加えて、解析エンジンによって発見された全てのマイグレーションの課題の要約などを含みます。これらのメトリクスと課題に関連するコスト(時間と費用)は自由に変更することができます。また、より深い分析のためにメトリクスを Microsoft Excel にエクスポートすることができます。



VB Migration Partner owes its high success ratio to its two main components: (a) a better parser and code generation engine, and (b) a support library that contains both the language support library and the control support library.

VB Migration Partner の高い変換率は二つの主要コンポーネントによるものです。すなわち、(a)高性能の解析エンジンとコード生成エンジンと、(b)言語サポートライブラリとコントロールサポートライブラリの両方を含むサポートライブラリです。

For example, VB Migration Partner's parser is able to convert a VB6 project groups into a VB.NET solution; it can convert GoSub and On Goto/Gosub statements; Declare parameters declared with "As Any" or that stand for callback addresses; User Define Type (UDT) blocks that require initialization, auto-instancing variables and arrays, IDisposable objects, fixed-length strings, and much more.

例えば、VB Migration Partner の解析エンジンは VB6 をのプロジェクトグループを VB.NET のソリューションに変換することができます。すなわち、GoSub と On Goto/Gosub ステートメント、「As Any」で宣言された宣言パラメータやコーリバックアドレスの代わりになる宣言パラメータ、初期化が必要なユーザー定義型、自動インスタンシング変数と配列、IDisposable オブジェクト、固定長文字列、それ他多くのステートメントを変換することができます。

For converted VB.NET applications to run correctly it is mandatory that the support library be distributed with the other executable files. While a few developers might dislike the approach based on the support library, it can be easily proved that only this approach can offer full compatibility with VB6 peculiarities and idiosyncrasies.

変換した VB.NET アプリケーションを正しく実行するためには他の実行可能ファイルと共に配布されたサポートライブラリが必須となります。サポートライブラリに基づくアプローチを嫌う少数の開発者もいますが、このアプローチのみが VB6 の独自性と特異性との完全な互換性を提供できることは簡単に証明することができます。

Code Architects plans to release more efficient and robust versions of the support library over time. When a new version of the support library is released, existing VB.NET applications that use the support library can be upgraded by simply deploying the new version on the end user's computer, without having to re-run VB Migration Partner.

コードアーキテクツ社は今後、より効率的かつ堅牢なサポートライブラリをリリースする予定です。新しいバージョンのサポートライブラリがリリースされた場合、VB Migration Partner を再実行することなく、エンドユーザのコンピュータに新しいライブラリを配布するだけでサポートライブラリを使用する既存の VB.NET アプリケーションをアップグレードすることができます。

1.1 Feature summary／機能要約

This section summarizes the main features of Code Architects' VB Migration Partner, with emphasis on those that are unique to this product.

このセクションではコードアーキテクツ社の VB Migration Partner の独自性を強調しつつ主要な機能を要約します。

General／全般

- high-speed conversion (up to 400 VB6 lines on a 3GHz system)
  高速変換
- runs outside Visual Studio
  VisualStudio と連携することなく実行できます。

- pragmas and extenders can affect migration behavior and help produce better code
  プラグマとエクステンダーはコード変換に影響を与え、良質のソースコードを生成するのに役立ちます。
- open architecture allows 3rd-party vendors to add support for their own ActiveX controls
  オープンアーキテクチャのおかげでサードパーティベンダーは自社の ActiveX コントロールをサポートすることができます。

Language／言語

- VB6 project groups are migrated to VB.NET solutions, project references are retained
  VB6 のプロジェクトグループはプロジェクト参照が保持された VB.NET ソリューションに移行されます。
- arrays with lower index other than zero
  ゼロ以下のインデックスの配列
- Gosub keyword, calculated On…Goto/Gosub
  Gosub キーワードと推測される On…Goto/Gosub
- auto-instancing (As New) variables and arrays
  自動インスタンス化(As New)変数/配列
- “As Any” parameters and callback parameters (AddressOf) in Declare statements
  Declare ステートメントにおける「AS Any」パラメータとコールバックパラメータ（AddressOf）
- most VB6 keywords not supported by VB.NET, including IsMissing, Array, DoEvents
  VB.NET ではサポートされない IsMissing、Array、DoEvents を含むほとんどの VB6 キーワード
- methods in support library exactly replicate the original VB6 behavior (e.g. Format, Dir, MsgBox), so that less time has to be spent on reviewing warnings
  サポートライブラリのメソッドは元の VB6 の動作を正確に複製（例 Format、Dir、MsgBox）するので、エラーを調査するために必要な時間を削減できます。
- full support for Type blocks (UDTs), fixed-length strings and arrays thereof
  タイプブロック（ユーザ定義型）とその固定長文字列と配列を完全にサポートします。
- reading and writing default properties, even in late-bound mode
  遅延バインディングにも対応するデフォルトプロパティの読み書き
- partial support for Variants, including Empty, Null, and null propagation in string expressions
  Empty、Null、Null 伝播を含むバリアントを部分的にサポートします。
- VB6 system objects, including Screen, Clipboard, App, and Printer
  Screen、Clipboard、App、そして Printer を含む VB6 システムオブジェクト

Forms and Controls／フォームとコントロール

- converts all controls installed with VB6 (with the exception of OLE container and Repeater control)
  VB6 によってインストールされる（OLE コンテナと Repeater コントロール以外の）全コントロールを変換します。

- controls in support library exactly replicate the original VB6 behavior
  サポートライブラリのコントロールは元の VB6 の動作を正確に複製します。
- control arrays, including arrays of menus and 3rd-party controls
  メニューとサードパーティ製コントロールの配列を含むコントロール配列
- dynamic control creation, both through control arrays and the Controls.Add method
  コントロール配列と Controls.Add を使った動的なコントロール作成
- popup menus and menu shortcuts
  ポップアップメニューとショートカットメニュー
- help-related properties and methods
  ヘルプに関連したプロパティとメソッド
- graphic methods: Line, Circle, PSet, Cls, PaintPicture, PrintForm methods and all graphics-related properties (with the only exception of ClipControls and DrawMode)
  グラフィックメソッド、すなわち、Line、Cirlcle、PSet、Cls、PaintPicture、PrintForm メソッドと（ClipControls と DrawMode 以外の）すべてのグラフィック関連プロパティ
- any value for the ScaleMode property, including custom coordinate systems
  カスタムコーディネートシステムを含む ScaleMode プロパティに対するすべての値
- “classic” (VB6-style) drag-and-drop
  「古典的」(VB6 様式)ドラッグアンドドロップ
- automatic and manual OLE drag-and-drop
  自動または手動 OLE ドラッグアンドドロップ
- DAO and RDO data binding
  DAO データバインディングと RDO データバインディング
- ADO Data binding, including binding to ADO Recordsets and BindingCollection objects, with support for custom data formatting and StdDataFormat objects
  カスタムデータフォーマットと StdDataFormat オブジェクトを伴う ADO レコードセットと BindingCollection オブジェクトに対するデータバインディングを含む ADO データバインディング
- DataEnvironment objects (excluding support for grouping and hierarchical recordsets)
  （グループ化されたレコードセットと階層化されたレコードセット以外の）DataEnvironment オブジェクト
- ADO data-source classes and ADO simple data consumer classes
  ADO データソースクラスと ADO シンプルデータコンシューマクラス
- full support for printing, including the Printer object, the Printers collection, and the Print and PageSetup common dialogs
  プリンタオブジェクト、プリンタコレクション、印刷とページセットアップコモンダイアログを含む印刷

COM Components／COM コンポーネント

- better support for IDisposable objects and finalization, including automatic disposal of fields and variables pointing to disposable objects
  廃棄可能なオブジェクトを指す自動的に廃棄されるオブジェクトのフィールドと変数を含む終了処理と IDisposable オブジェクト

- MTS/COM+ components, including support for most common objects in comsvcs.dll
  comsvcs.dll の最も一般的なオブジェクトを含む MTS/COM+コンポーネント
- private and public UserControl classes
  プライベートおよびパブリックのユーザーコントロールクラス
- persistable classes and the PropertyBag object
  Persistable クラスと PropertyBag オブジェクト
- Sub Main is correctly called before any class in a DLL (as in VB6)
  （VB6 のように）Sub Main が Dll のすべてのクラスより早く正しく呼び出されます。
- VB6 Description attribute translates to XML comments and (if inside a UserControl) to Description attributes
  VB6 の Description attributes を XML コメントと（ユーザーコントロール内部の場合には）Description attributes に変換します。
- support for common type libraries such as FileSystemObject, Dictionary, and RegExp, without requiring COM Interop
  COM Interop を必要としない辞書、正規表現、ファイルシステムオブジェクトのようなコモンタイプライブラリをサポートします。

1.2 The language support library／言語サポートライブラリ

The language support library is entirely contained in the CodeArchitects.VBLibrary.DLL file and provides support for language commands that behave differently (or are missing) in the Microsoft.VisualBasic.dll file that comes with VB2005. All the objects and methods implemented in this support library have a trailing “6” appended to the original VB6 name, as in DoEvents6 or App6.

言語サポートライブラリは完全に CodeArchitects.VBLibrary.Dll に含まれています。 そして、VisualStudio2005 に配付されている Microsoft.VisualBasic.Dll にあるものとは異 なる動作もしくはまったくない言語の命令のサポートを提供します。全てのオブジェクト とメソッドはオリジナル VB6 の名前に「6」を付与した名前でサポートライブラリに実装されてい ます。（例　App6、DoEvents6 など）

For example, this VB6 code fragment:
参考例　抜粋した VB6 コードより

```
If IsEmpty(value) Or IsNull(value) Then
   value = Array(1, 2, 3, 4, 5)
End If
```

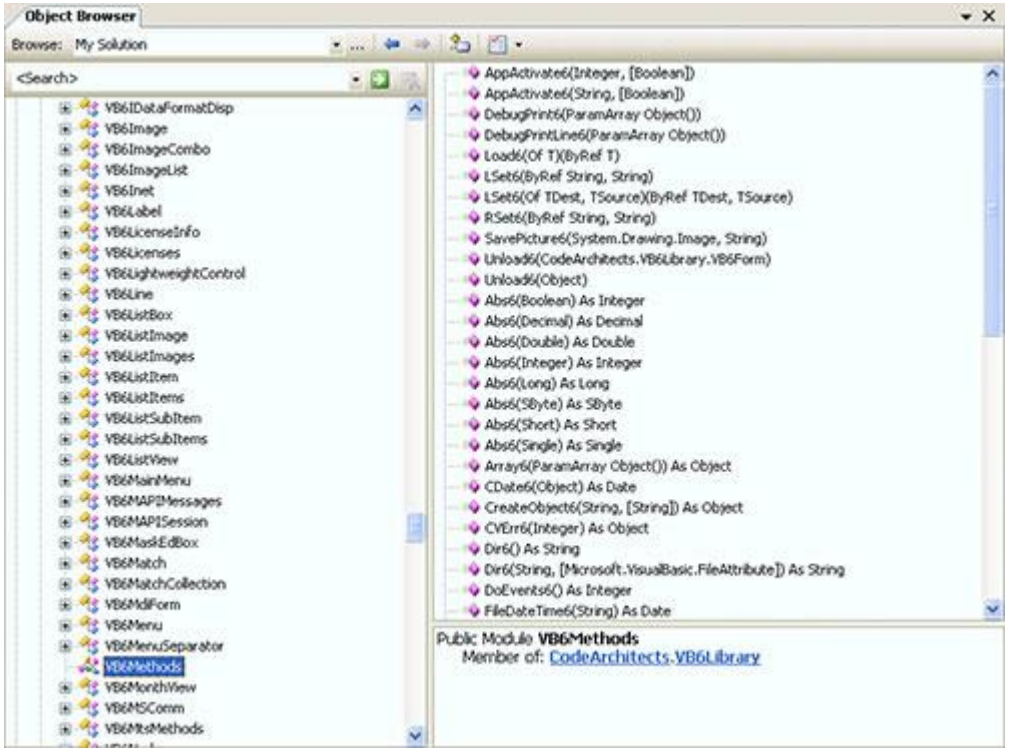
is translated to VB.NET as follows:

これを VB.NET に変換すると以下のようになります。

```
If IsEmpty6(value) Or IsNull6(value) Then
   value = Array6(1, 2, 3, 4, 5)
End If
```

Methods that replace or extend members of the Microsoft.VisualBasic.dll library are exposed as members of the VB6Methods module, or other modules defined in the CodeArchitects.VBLibrary.dll.

Microsoft.VisualBasic.Dll の拡張されたメンバや置き換えられたメソッドは VB6 メソッドモジュールまたは CodeArchitects.VBLibrary.Dll において定義された他のモジュールのメンバとして引き当てられます。



These are the VB6 keywords that aren't supported by VB.NET and that VB Migration Partner supports by means of the language support library:

以下の VB6 キーワードは VB.NET ではサポートされませんが、VBMigrationPartner の言語サポートライブラリではサポートされております。

- **Array6** creates an array of Object elements.
  Array6 はオブジェクト要素の配列を生成します。
- **CVar6** returns an instance of the VB6Variant class.
  CVar6 は VB6Variant クラスのインスタンスを戻します。
- **CVErr6** returns an instance of the VB6Error class.
  CVErr6 は VB6Error クラスのインスタンスを戻します。
- **IsEmpty6**, **IsMissing6**, **IsNull6**, and **IsObject6** methods account for VB6Variant values.
  VB6Variant Value のためのメソッドアカウントである **IsEmpty6**、**IsMissing6**、**IsNull6**、**IsObject6**

- **LoadPicture6** supports all the arguments as the original VB6 method, but throws an exception if these extra arguments can't be honored.
  LoadPicture6 はオリジナル VB6 メソッドの全ての引数をサポートします。ですが、もし特別な引数が認められない場合は、例外をスローします。
- **String6** works like StrDup but supports numeric values for its second argument.
  String6 は StrDup のように機能します。2 つ目の引数である数値をサポートします。
- **SavePicture6** saves an image to BMP format.
  SavePicture6 はビットマップフォーマットイメージを保存します。
- **VarType6** works correctly with scalar and array values stored in an Object element.
  VarType6 はオブジェクト要素の中にスカラーと配列の値を格納するように正確に機能します。

A few methods are supported by VB.NET but behave slightly differently from VB6, therefore VB Migration Partner re-implements them to ensure that no discrepancy exists:

いくつかのメソッドは VB.NET でサポートされておりますが VB6 の動作とは若干異なります。VBMigrationPartner はそれらの食い違いがないように実装されています。

- **Abs6** works with Boolean values, too.
  Abs6 はブール値でも動作します。
- **AppActivate6** supports a second *wait* argument .
  AppActivate6 は 2 番目の *wait* 引数をサポートします。
- **CDate6** accepts numeric values strings whose month and day values are reversed (locale-tolerant).
  CDate6 は月と日の値が逆になっている数値文字列でも（ロケールの設定として正しければ）許容します。
- **CreateObject6** works well also with public (managed) classes exposed by the current solution.
  CreateObject6 は管理されたパブリッククラスとして現在のソリューションで公開され、動作します。
- **DebugPrint6** and **DebugPrintLine6** display strings in the Debug window in the same format used by VB6.
  DebugPrint6 と **DebugPrintLine6** は VB6 とまったく同じフォーマットの DebugWindow で文字列を表示します。
- **Dir6** returns "." and ".." elements and then returns names of files in a directory.
  Dir6 は「.」や「..」という要素を戻します。またディレクトリ内のファイル名を返します。
- **DoEvents6** returns the number of open forms.
  DoEvents6 は開かれているフォームの数を返します。
- **FileDateTime6** works with both files and directories (the VB.NET method works only with files).
  FileDateTime6 はファイルとディレクトリの両方で動作します。(VB.NET のメソッドではファイルのみしか動作しません)
- **FileOpen6**, **FileClose6**, **FileGet6**, **FilePut6**, and all other file-oriented method read and write values and UDTs using the same format that VB6 uses.
  FileOpen6、**FileClose6**、**FileGet6**、**FilePut6**、また全ての他のファイル指向メソッドは値を読み書きします。VB6 で使用している、同じフォーマットのユーザー定義型で使われています。

- **Format6** supports named formats (e.g. “scientific”) and null values, and accounts for minor differences between VB6 and VB.NET.
  Format6 は Null 値 VB6 と VB.NET とのわずかな違いを占める定義されたフォーマット名をサポートします。（例　scientific）
- **Input6** converts coordinates and correctly handles CRs in prompt strings.
  Input6 はプロンプト文字列の中で CR を正しく処理し、座標に変換します。
- **IsDate6** accepts strings where month and day values are reversed.
  IsDate6 は月と日の値が反対になっている文字列を許容します。
- **Len6** works both with strings and User-Defined Types (UDTs).
  Len6 は文字列と、ユーザー定義型の両方で動作します。
- **LSet6** has support for strings and partial support for UDTs.
  LSet6 は文字列をサポートします。ユーザー定義型は一部サポートします。
- **MsgBox6** correctly handles CRs in prompt.
  MsgBox6 はプロンプト内の CR を正確に処理します。
- **RSet6** supports strings.
  RSet6 は文字列をサポートします。
- **Str6** works with dates.
  Str6 は日付と共に動作します。
- **StrConv6** can convert Byte arrays to a string and works better with conversions to and from Unicode strings.
  StrConv6 はバイト配列を文字列に変換することができ、Unicode 文字列からの変換も容易に可能です。
- **TypeName6** returns the value that would be returned under VB6; for example, when applied to an Int32 returns “Long”, when applied to a button control returns “Command”, and so forth.
  TypeName6 は VB6 での戻り値を返します。例として、Int32 で定義されたものは「Long」で。ボタンコントロールで定義されたものは「Command」でなど。

The following keywords have been re-implemented to support extra features – for example, Variants and null propagation in expressions – that aren’t natively supported by VB.NET:

次のキーワードは様々な機能をサポートする為に再実装されています。例えば、バリアント型、式の中での Null 伝搬。VB.NET ではそれらは基本的にサポートされていません。

- **Chr6, CurDir6**, **Environ6**, **Hex6**, **LCase6**, **Left6**, **Mid6**, **Oct6**, **Right6**, **RTrim6**, **Space6**, **Trim6**, and**UCase6** account for null values.
  Chr6、**CurDir6**、**Environ6**、**Hex6**、**LCase6**、**Mid6**、**Oct6**、**Right6**、**RTrim6**、**Space6**、**Trim6**、そして **UCase6** は Null 値をサポートします。
- **IsArray6**, **IsDate6**, **IsError6**, **IsNothing6**, and **IsNumeric6** recognize values stored in Object and VB6Variant variables.
  IsArray6、**IsDate6**、**IsError6**、**IsNothing6**、そして **IsNumeric6** はオブジェクトと VB6 バリアント型変数の中の値を保管することを認識します。

- **Erase6**, **Redim6**, **RedimPreserve6**, **IsArray6**, **LBound6**, and **UBound6** work with regular arrays, arrays stored in Object variables, and VB6Array objects.
  Erase6、**Redim6**、**RedimPreserve6**、**IsArray6**、**LBound6**、**UBound6** は一般的な配列をサポートします。オブジェクト変数の中に格納された配列、VB6Array オブジェクトもサポートします。
- **Load6** and **Unload6** perform additional processing that might be required in VB.NET applications converted from VB6.
  Load6 と **Unload6** は VB6 から変換された VB.NET アプリケーションで必要になるかもしれない追加処理を実行します。

The support library contains the counterpart of VB6 methods that can't be implemented or mimicked perfectly under VB.NET. All the methods in this group are marked as Obsolete, thus they cause a warning message to be displayed in the Error List window. When invoked, these methods either do nothing or throw an exception:

サポートライブラリは VB6 のメソッドに対応したものを含みます。それは VB.NET 下において完全に実行できないか、まねることができません。以下のグループの全てのメソッドはサポートされないものとして明らかにされます。よって、それらはエラーリストウィンドウに警告メッセージを表示します。起動時に、これらのメソッドは何も動作しないか、例外をスローします。

- **ImeStatus6** and **Calendar6** always return 0, assignments are ignored.
  ImeStatus6 と **Calendar6** は常に 0 （ゼロ）を戻します。値の割当は無視されます。
- **AscB6, ChrB6, InstrB6, LeftB6, RightB6, MidB6, InputB6**, and **LenB6** return an "approximate" value in VB.NET, and the warning message encourages the developer to edit the original or migrated code to get rid of such warnings.
  AscB6、**ChrB6**、**InstrB6**、**LeftB6**、**RightB6**、**MidB6**、**InputB6**、**LenB6** は VB.NET で"近似値"を返します。また、警告メッセージは開発者が警告を取り除く為に、オリジナルのコードやマイグレーション後のコードを編集するのに役立ちます。
- **VarPtr6**, **StrPtr6**, and **ObjPtr6** throw an exception.
  VarPtr6、**StrPtr6**、**ObjPtr6** はエラーをスローします。

The six VB6 system objects can be referenced by means of the following members:

6つの VB6 システムオブジェクトは以下のメンバーの手法で参照することができます。

- **App6:** most members are supported, included PrevInstance and methods related to event logging; the OleRequest* and OleServer* properties aren't supported and are marked as obsolete.
  App6 : ほとんどのメンバはサポートされます。イベントログに関連した PrevInstance とメソッドを含みます。OleRequest*と OleServer*プロパティは廃止されたものですのでサポートされません。
- **Clipboard6:** all members are supported.
  Clipboard6 : 全てのメンバはサポートされます。

- **Screen6:** all members are supported, except Fonts, FontCount, MousePointer, and MouseIcon are flagged as obsolete; assignments to MousePointer and MouseIcon throw an exception.
  Screen6：全てのメンバはサポートされます。Fonts を除き、FontCount、MousePointer、MouseIcon は廃止されました。割り当てられた MousePointer と MouseIcon は例外をスローします。
- **Printer6** and **Printers6**: all members are supported and behave exactly in VB6.
  Printer6 と **Printers6**：VB6 の動作と同様、全てのメンバはサポートされます。

Note: support for the Printer6 and Printers6 objects is provided by the VBSupportLib.dll library. This DLL is a VB6 executable, therefore it requires that the VB6 runtime be installed on the target computer.

注記：Printer6 と Printers6 オブジェクトのサポートは VB6SupportLib.dll ライブラリで提供されています。この DII は VB6 で実行可能なファイルです。よって VB6 ランタイムをターゲットコンピュータにインストールする必要があります。

In addition to methods and properties, the language DLL support most of the objects defined in the VB6 runtime. All the objects in this group have a trailing “VB6” prefix, as in “VB6Variant” and “VB6PropertyBag”.

メソッドとプロパティに加えて、言語 DII は VB6 の定義されたほとんどのオブジェクトをサポートします。以下のグループの全てのオブジェクトは「VB6Variant」とか「VB6PropertyBag」のように「VB6」という接頭語をもっています。

- **VB6AsyncProperty** simulates asynchronous properties in user controls.
  VB6AsyncProperty はユーザコントロールの非同期プロパティをシミュレートします。
- **VB6DataBinding** and the **VB6DataBindings** collection correspond to the DataBinding and DataBindings objects defined in the VB6 runtime.
  VB6DataBinding と **VB6DataBindings** コレクションは VB6 ランタイムの定義された DataBinding と DataBindings オブジェクトに相当します。
- **VB6DataEnvironment** provides most of the functionality offered by the DataEnvironment object, except support for grouping, relations, and hierarchical recordsets.
  VB6DataEnvironment は DataEnvironment オブジェクトが提供するほとんどの機能を提供します。ただし、グループ化、リレーション、階層化レコードセットを除外します。
- **VB6StdDataFormat** and the **VB6StdDataFormats** collection provide support for custom formatting in data binding scenarios.
  VB6StdDataFormat と **VB6StdDataFormats** コレクションはデータバインディングシナリオのカスタムフォーマットサポートを提供します。
- **VB6DataObject** and **VB6DataObjectFiles** are used in drag-and-drop scenarios to contain data moved from one control or application to another.
  VB6DataObject と **VB6DataObjectFiles** は 1 つのコントロールまたはアプリケーションから別のアプリケーションまで動かされたデータを含むのにドラッグ・アンド・ドロップシナリオで使用されます。
- **VB6Error** is the object returned by the CVErr method.
  VB6Error は CVE r r メソッドで返されるオブジェクトです。
- **VB6LicenseInfo** and the **VB6Licenses** collection simulate license features of user controls.
  VB6LicenseInfo と **VB6Licenses** コレクションはユーザコントロールのライセンス機能をシミュレートします。

- **VB6PropertyBag** mimics the functionality of the VB6 PropertyBag object, even though the storage format differs from VB6’s (in other words, it isn’t possible to read VB6 serialized objects from VB.NET apps, and vice versa).
  VB6PropertyBag は VB6 の PropertyBag オブジェクトの機能性を模倣したものです。ですが VB6 のものとは異なったストレージフォーマットです。（言い換えると、VB.NET のアプリからシリアル化された VB6 オブジェクトを読み込むことはできません。その逆も同様です。）
- **VB6Variant** duplicates some of the functionality of the original VB6 Variant data type, such as support for Null and Empty values.
  VB6Variant はオリジナル VB6 バリアントデータ型の機能性のいくつかは重複します。Null と空の値などをサポートします。
- **VB6VBControlExtender** is the alias for the VBControlExtender object used to trap events from controls that are added dynamically by means of the Controls.Add method.
  VB6VBControlExtender は Control.Add メソッドによって動的に追加されたコントロールからのトラップイベントに使用される VBControlExtender オブジェクトのエイリアスです。

A few objects have no direct counterpart in VB6:

いくつかのオブジェクトは VB6 と直接対照のものがないものもあります。

- **VB6Array** provides support for arrays with any lower index.
  VB6Array はどんなに低いインデックスの配列にも対応します。
- **VB6ArrayNew** provides support for arrays of auto-instancing objects (as in Dim arr() As New Person).
  VB6ArrayNew は自動インスタンスオブジェクトの配列をサポートします。（例　Dim arr() As New Person）
- **VB6ControlArray** mimics VB6 control arrays and can contain both built-in controls and 3rd party controls.
  VB6ControlArray は VB6 のコントロール配列をと同様の動作をし、組込コントロールと 3 rdParty コントロールの両方に対応。
- **VB6ControlCollection** is the collection returned by the Controls property of forms and user control; unlike the .NET Controls collection, it contains controls all the controls hosted by the form or the user control, including those contained in container controls (e.g. a PictureBox or a Frame control).
  VB6ContolCollection は form とユーザコントロールのコントロールプロパティによって戻すコレクションです。.NET のコントロールコレクションとは異なります。それはコンテナコントロールに含まれたものを含んでいて、form かユーザコントロールで立てられた全てのコントロールを含んでいます。（例　PictureBox や Frame Control）
- **VB6FixedString** offer support for the translation of fixed-length strings (FLS).
  VB6FixedString は固定長文字列を解釈するサポートを提供します。
- **VB6WindowSubclasser** can be used to implement safer and more robust window subclassing.
  VB6WindowSubclasser はより安全で堅牢なウィンドウをサブクラス化するのを実装する為に使うことが出来ます。

Finally, the support library includes managed counterparts of the following COM objects:

最後に、サポートライブラリには以下にある COM オブジェクトに対応した対象物を含みます。

- **VB6Binding** and **VB6BindingCollection** duplicate the functionality of the Binding and BindingCollection objects in the MSBind type library.
  VB6Binding と **VB6BindingCollection** は MSBind Type Library の中の Binding と BindingCollectionObject の機能を再現しています。
- **VB6Dictionary** is the alias for the keyed collection defined in the Scripting type library.
  VB6Dictionary は Scripting Type Library の定義されたキーコレクションのエイリアスです。
- **VB6FileSystemObjects** and all related classes, such as **VB6Drive** and **VB6File**, mimic the behavior of file-related objects defined in the Scripting type library.
  VB6FileSystemObjects と全ての関連のあるクラス、例えば **VB6Drive** と **VB6File** は Scripting Type Library の定義されたファイル関連オブジェクトの動作を再現しています。
- **VB6ObjectContext** is the VB.NET counterpart of the COMSVCSLib.ObjectContext used by MTS/COM+ components.
  VB6ObjectContext は MTS/COM+コンポーネントによって使用されている COMSVCSLib.ObjectContext の VB.NET 対象物です。
- **VB6RegExp** and all related classes support the migration of VB6 apps that rely on the VBScript Regex Engine type library.
  VB6RegExp と全ての関連するクラスは VBScript Regex Engine type library の信頼のもとに VB6 アプリケーションのマイグレーションをサポートします。

1.3 The control support library／コントロールサポートライブラリ

Here is the complete list of the 64 controls that VB Migration Partner supports:

以下に VBMigrationPartner でサポートしている 64 個のコントロールを一覧します。

Top-level objects／最上位オブジェクト:

Form	MDIForm	UserControl

Built-in controls／組み込みコントロール:

CheckBox	ComboBox	CommandButton
Data	DirListBox	DriveListBox
FileListBox	Frame	HScrollBar

Image	Label	Line
ListBox	Menu	OptionButton
PictureBox	Shape	TextBox
Timer	VScrollBar	

Windows Common controlsWindows／Windows 共通コントロール:

Animation	DTPicker	FlatScrollBar
ImageCombo	ImageList	ListView
MonthView	ProgressBar	Slider
StatusBar	TabStrip	Toolbar
TreeView	UpDown	

Window-less controls／非ウインドウコントロール:

WLCheck	WLCombo	WLCommand
WLFrame	WLHScroll	WLList
WLOption	WLText	WLVScroll

Other controls／その他のコントロール:

ADO Data	CommonDialog	DataCombo
DataList	MaskEdBox	PictureClip
Remote Data	RichTextBox	SSTab
SysInfo	WebBrowser	

ActiveX Components (invisible at runtime)／ActiveX コンポーネント(実行時非表示):

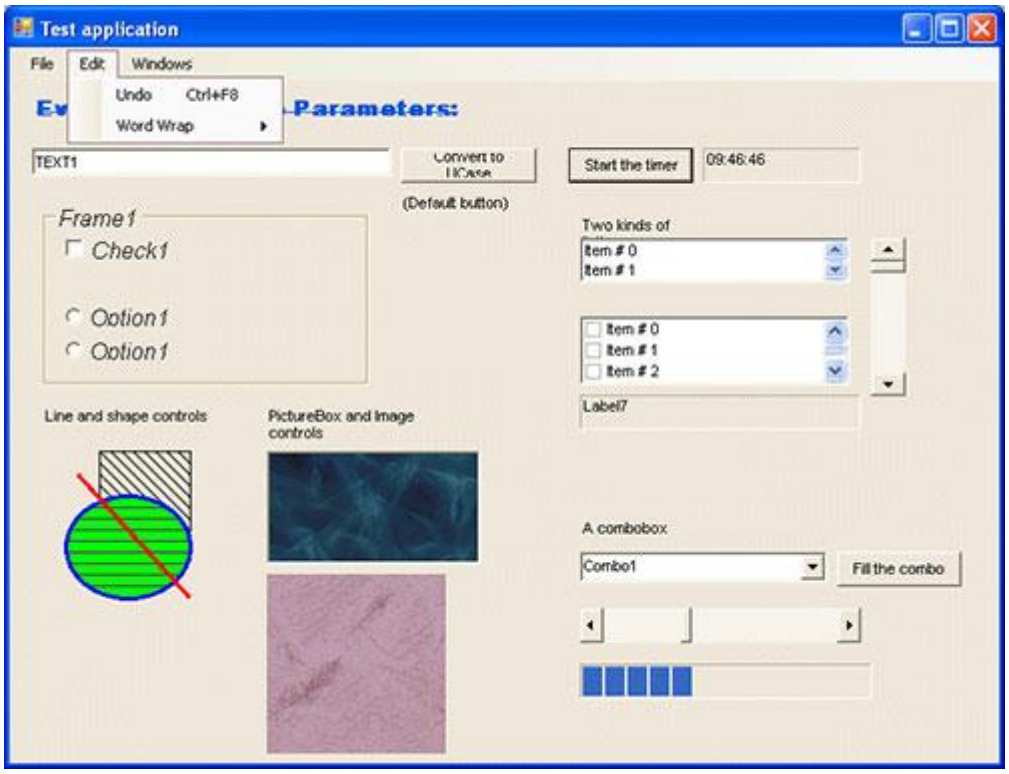
INet	MAPIMessage	MAPISession
MSComm	ScriptControl	Winsocket

ActiveX Controls (visible at runtime)／ActiveX コンポーネント(実行時表示):

MMControl	MSCalendar (MSCAL.Calendar)	MSChart
MSDataGrid	MSHierarchicalFlexGrid	

Notice that the list includes all the controls that are installed with Visual Basic 6, with the only exception of the OLE container control and the Repeater control. *(We plan to add support for the Repeater in a future version of this support library.)*

このリストは VisualBasic6 にインストールされている全てのコントロールを含んでいます。例外としては OLE コンテイナーコントロールとリピーターコントロールは含みません。(将来リリースされる予定の Version ではリピーターのサポートを追加することを検討しています。)



In general, the name of all the classes in the control support library is formed by prefixing "VB6" to the name of the original VB6 control. For example, the VB6CommandButton control renders the VB6 CommandButton control, and so on.

一般に、サポートライブラリの全てのクラスの名前は、オリジナル VB6 コントロールの名前の前に「VB6」を付与した名前で生成されています。例えば、VB6CommandButton コントロールは VB6 の CommandButton コントロールを表現しています。

In most cases, a class that replaces a VB6 control inherits from a Windows Forms control and adds or overrides members that behave exactly as they do under VB6. For example, the class that supports VB6's TabStrip control inherits from System.Windows.Forms.TabControl. This approach ensures that the converted VB.NET application has no dependency from the original ActiveX control.

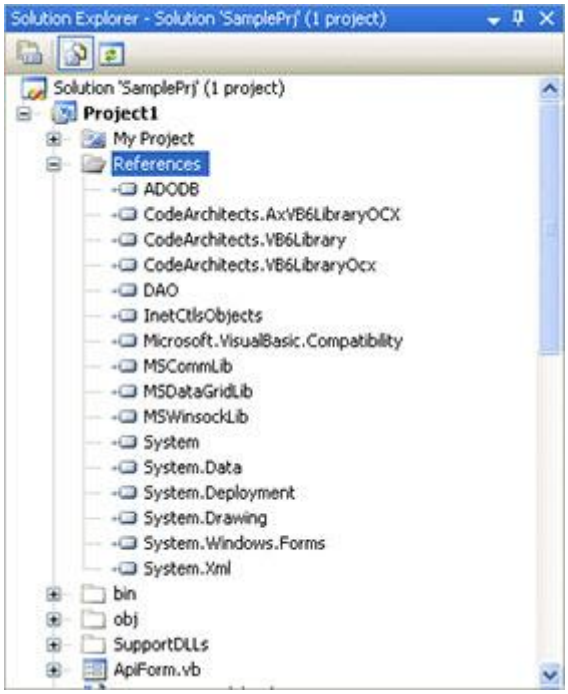
ほとんどのケースにおいて、クラスは Windows フォームコントロールから継承した VB6 コントロールを置き換えています。追加またはオーバーライドメンバーは VB6 と正確な動作をします。例えば、VB6 の TabStrip コントロールをサポートしたクラスは、System.Windows.Forms.TabControl から継承しています。このアプローチは変換された VB.NET アプリケーションがオリジナル ActiveX コントロールからの依存をもたないということを保証します。

Only the controls that belong to the ActiveX Components and ActiveX Controls groups listed above are implemented as wrappers on the original ActiveX objects.

ActiveX コンポーネントに属すコントロールと上記にリストされた ActiveX コントロールグループのみがオリジナルの ActiveX オブジェクトのラッパーとして実装されています。

If the original VB6 application uses one or more controls listed in the ActiveX Components group, then the converted VB.NET project includes a reference to a TlbImp-generated DLL (for example MSCommLib for the MS Comm control). If the original VB6 application uses one or more controls listed in the ActiveX Controls group, then the converted VB.NET project includes a reference to the CodeArchitects.VBLibraryOCX.dll and CodeArchitects.AxVBLibraryOCX.dll libraries:

オリジナル VB6 アプリケーションが ActiveX コンポーネントグループにリストされたひとつもしくは複数のコントロールを使用する場合、変換された VB.NET プロジェクトは TlbImp により生成された DLL を参照したものを含みます。（例えば、MS Comm Control の MSCommLib）オリジナル VB6 アプリケーションが ActiveX コントロールグループにリストされたひとつもしくは複数のコントロールを使用する場合、変換された VB.NET プロジェクトは CodeArchitects.VBLibraryOCX.dll と CodeArchitects.AxVBLibraryOCX.dll ライブラリを参照したものを含みます。



Here is a list of relevant features that VB Migration Partner supports:

以下は VBMigrationPartner のサポートに関連する機能一覧です。

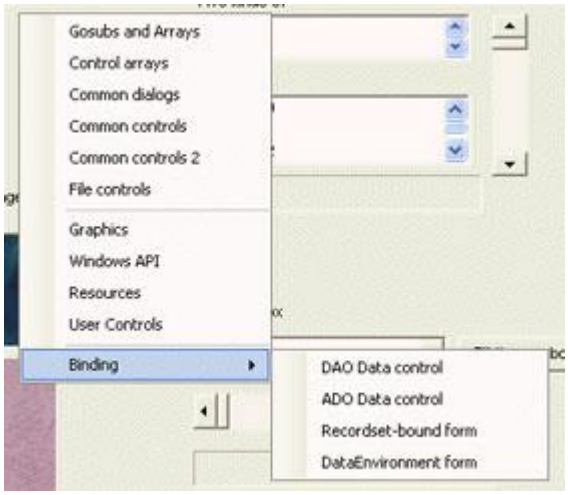
- Late binding／遅延バインディング

  The fact that members of the VB.NET control have the same name, return type, and syntax as the original VB6 control ensures that existing code accessing the control in late-bound mode continues to work after the migration to VB.NET.

  VB.NET コントロールのメンバは同じ名前、戻り値の型を保持し、そしてオリジナル VB6 コントロー

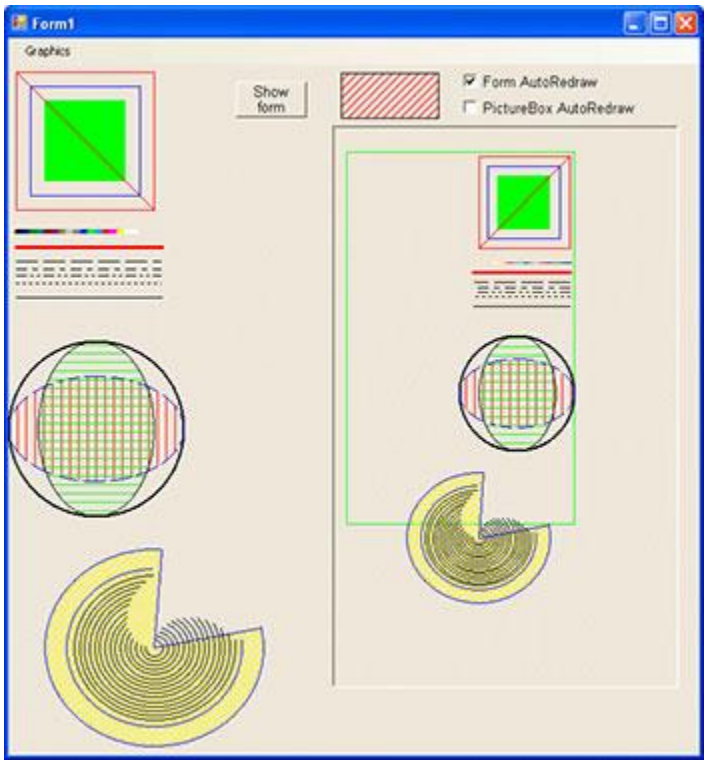
ルの構文は Late-Bound モードでコントロールにアクセスし続け、VB.NET に移行された後も機能し続けるということを保証します。

- **Standard and popup menus／スタンダードメニューとポップアップメニュー**
  Standard and popup menus are fully supported, including shortcut keys and control arrays of menu items.
  スタンダードメニュー、ポップアップメニューは完全にサポートされています。ショートカットキー、メニューアイテムのコントロール配列もサポートします。



- **Control arrays／コントロール配列**
  All control array features are supported, including dynamic loading and events. Support is provided by means the VB6ControlArray(Of T) type. Because of the generic nature of this type, VB Migration Partner supports arrays of *any* controls, including 3rd party controls. Arrays of menu items are supported as well.
  動的読込やイベントを含む全てのコントロール配列をサポートしています。サポートは VB6ControlArray のタイプによる提供です。このタイプの一般的な性質のために、VBMigrationPartner は 3 rdParty コントロールを含む複数のコントロール配列をサポートします。メニューアイテムの配列もサポートしています。
- **Dynamic control creation／動的なコントロールの作成**
  In addition to loading a control by means of a control array, the CodeArchitects.VBLibrary DLL fully supports the Controls.Add method. The return value from this method can be assigned to a VBControlExtender variable, and VB.NET code can handle the ObjectEvent event exactly as the original VB6 code does.
  コントロール配列のコントロールを読み込むことに加えて、CodeArchitects.VBLibrary DLL は Controls.Add メソッドを完全にサポートしています。このメソッドからの戻り値は VBControlExtender 変数に代入することができます。そして VB.NET コードは ObjectEvent イベントをオリジナルの VB6 コードと同じく正確に動作させることができます。
- **Graphic methods／グラフィックメソッド**
  Cls, Line, Circle, PaintPicture, PSet, Point, Print, Scale, TextWidth, and TextHeight methods are supported for the Form, PictureBox, and UserControl classes. All graphics-related properties are fully supported (including AutoRedraw and persistent graphic), except ClipControls and DrawMode. The PrintForm method is supported, too.

Cls, Line, Circle, PaintPicture, PSet, Point, Print, Scale, TextWidth, TextHeight メソッドは Form, PictureBox, UserControl クラスをサポートします。全てのグラフィック関連プロパティは完全にサポートされています。(AutoRedraw と持続性グラフィックを含む) ただし、ClipControls と DrawMode を除く。PrintForm メソッドはサポートされます。



- **Coordinate systems／コーディネートシステム**
  ScaleMode property can be set to values other than vbTwips, both at design-time and at run-time; ScaleLeft, ScaleTop, ScaleWidth, and ScaleHeight properties and Scale, ScaleX, and ScaleY methods are supported as well.
  ScaleMode プロパティは vbTwips、デザイナーと実行時の両方以外で値をセットすることができます。ScaleLeft、ScaleTop、ScaleWidth、ScaleHeight プロパティと Scale、ScaleX、ScaleY メソッドはサポートしています。
- **Data binding／データバインディング**
  The control library supports binding with the Data, RDO Data, and ADODC controls, perfectly reproducing the VB6 behavior, including custom formatting by means of the StdDataFormat object and its Parse and Format events. VB Migration Partner supports also binding to ADO Recordsets, DataEnvironment objects, ADO data source classes, ADO simple data consumer classes, and BindingCollection objects.
  コントロールライブラリは StdDataFormat オブジェクトとその解析とフォーマットイベントのカスタム書式を含み VB6 の動作を完全に再現しデータを RDO データ、ADODC コントロールで結合することをサポートします。VBMigrationPartner は ADO Recordsets、DataEnvironment オブジェクト、ADO Data Source クラス、ADO simple data consumer クラス、BindingCollection オブジェクトでデータ結合することもサポートします。
- **Error codes／エラーコード**
  When the support library throws an error, the error is raised by means of the Err.Raise method (rather than a Throw statement). Care has been taken in using exactly the same error codes that

would be produced in VB6. This detail is essential to ensure that existing VB6 error handlers work correctly after the migration to VB.NET.
サポートライブラリがエラーをスローした際に、エラーは Err.Raise メソッド（Throw ステートメントではない）により、提起されます。VB6 で提供されているエラーコードと同じものを使用することで、エラーを捉えることができます。この詳細は VB.NET に変換された後で、既存の VB6 エラーハンドラを正しく機能することを保証することは不可欠です。

- **Enum properties／列挙型プロパティ**
  All enumerated values have retained the value they have in VB6. This feature ensures that if an enum property is assigned a value returned by a Function or read from a configuration file, such a piece of code continues to work as expected after the migration to VB.NET. Spaces in enumerated values – as in [Test Value] – are replaced by underscores.
  全ての列挙された値は、VB6 で持っている値を保有します。この機能は Function または設定ファイルから読まれ戻された値が Enum プロパティに割り当てられるなら、そのようなひとつのコードが VB.NET に変換された後で期待されたように動作し続けることを保証します。列挙値の中のスペースはアンダースコアに置換されます。（例　「TEST VALUE」、変換後「TEST_VALUE」）
- **Help support／HELP 対応**
  All the help-related properties and methods are supported, including HelpContextID and WhatsThisHelpID. The converted VB.NET program can continue to use the help file provided with the original VB6 application.
  全ての HELP 関連のプロパティとメソッドは HelpContextID と WhatsThisHelpID を含みサポートされます。変換された VB.NET プログラムはオリジナル VB6 アプリケーションで提供された HelpFile を使い続けることができます。

1.4 Pragmas and the "convert-test-fix" cycle プラグマと「convert-test-fix」サイクル

Pragmas are special remarks that developers can add to the VB6 code to affect the behavior of the VB Migration Partner. The parser considers as a pragma any comment that starts with the "##" sequence; if the pragma name isn't recognized, a warning appears in the Log Activity window.

開発者は VBMigrationPartner の動作に影響を与えるように VB6 のコードを追加することができるというのがプラグマの特徴です。解析ツールはプラグマで考えられたものは「##」でつなげたコメントで始めます。もしプラグマ名が認められない場合は、警告が Log Activity Window に表示されます。

Pragmas encourage the process we call ***convert-test-fix cycle***. The convert-test-fix cycle is essential in converting large VB6 applications that need to be maintained or expanded until the migration process is completed and the VB.NET application is ready for the market. For large applications, in fact, the easiest way to ensure that the VB6 and VB.NET versions are always in-sync is doing as much work as possible on the original VB6 code and annotating it with pragmas. These pragmas tell VB Migration Partner how to migrate given pieces of code without producing errors.

Pragmas は私たちが提案する「**convert-test-fix サイクル**」のプロセスで働きかけます。変換-検証-修正サイクルは変換しようとする巨大な VB6 アプリケーションには必要不可欠です。その作業は、VB.NET アプリケーションとしてリリースされるまで、またはマイグレーション作業が完了するまで維持管理と拡張が必要になります。巨大なアプリケーションとして、実際に VB6 と VB.NET バージョンがいつも同期することを保証する最も簡単な方法としては、オリジナル VB6 コードをできるだけ多く機能するようにプラグマで注釈を施すようにします。これらのプラグマは VB Migration Partner がどのように生成するエラーがないようにコードの部分を変換すればよいか、教えます。

A key feature of pragma is that they can be scoped at the project, class, method, and variable level. Project-level pragmas can appear anywhere in the VB6 source code and use the **project:** prefix. For example, the following pragma tells the code generator to use the Arial 10pt font for all the forms in the current project, unless another FormFont pragma at the form level overrides it:

Pragma の主要な機能はプロジェクト、クラス、メソッド、変数レベルを捉えられることです。プロジェクトレベルのプラグマは VB6 ソースコードのどこにおいても出現し、「**project:**」を前に付与した形式で使います。例えば、以下のプラグマは現在のプロジェクトの全てのフォームに対し、Arial 10pt のフォントを使うようにコード生成プログラムに指示します。FormFont プラグマがフォームレベルにおいて上書きしない限り。

```
'## project:FormFont Arial, 10
```

Pragma arguments are separated by commas; if an argument is a string literal that contains commas, it must be enclosed in double quotes. If an argument contains a command and a double quote character (in a remark, for example), it must be enclosed in double quotes and all double quotes in the original value must be doubled, as you would do if it were a VB literal string.

プラグマの引数はコンマによって分割されます。引数がコンマを含むストリングリテラルの場合、ダブルコーテーションで囲む必要があります。引数にコマンドとダブルコーテーション文字が含まれている場合（例えばコメントなどに）、元のダブルコーテーションの文字列を二重のダブルコーテーションで囲む必要があります。VB でのリテラル文字列についてすることと同様です。

There are a few of exceptions to the above rule, most notably the InsertStatement, ReplaceStatement, Rem, and Note pragmas. These pragmas take just one argument, which is an entire VB.NET statement, and never require that their only argument be enclosed in double quotes, because the comma can't be misinterpreted as an argument separator.

上記のルールには例外がいくつかあります。特に InsertStatement、ReplaceStatement、Rem、Note プラグマです。これらのプラグマはただひとつの引数を取ります。全ての VB.NET ステートメントにおいて、それらの唯一の引数はダブルコーテーションで囲む必要はありません。コンマは引数区切りとして誤解されることはできないので。

A pragma can be applied to a specific member by prefixing its name with the member name, using the “dot” syntax. For example, the following VB6 code snippet applies the DeclareImplicitVariables pragma to the Test method (this pragma forces VB Migration Partner to generate a Dim statement for each variable that is implicitly declared):

プラグマはドット構文を使って、メンバ名の名前を前に置くことによって、特定のメンバを適用させることができます。例えば、次の VB6 コード断片は Test メソッドに DeclareImplicitVariables プラグマを適用します。(このプラグマは暗黙的に宣言された各変数用に Dim ステートメントを生成するように VB Migration Partner を勧めます。)

```
'## Test.DeclareImplicitVariables True
...

Sub Test()
    ...
End Sub
```

You use the “dot” syntax to refer to specific variables, if the pragma can be applied to a variable. The following code tells the code generator to consider the *frm* variable as an auto-instancing variable (in this case VB Migration Partner generates code that preserves the As New semantics):

プラグマが変数に適用できるのであれば、具体的な変数を参照するためにドット構文を使用します。次のコードは、frm 変数を自動インスタンス変数としてみなすように、コードジェネレータに指示します。(この場合、VB Migration Partner は As New という定義で保持するコードを生成します)

```
'## frm.AutoNew True
Dim frm As New Form1
```

If a pragma isn't prefixed by **project:** or by a member name, its scope depends on where it appears in the VB6 code. The scoping rules are the same as in VB6: if the pragma appears at the class-, form-, or module-level – that is, it isn't inside a method – it affects the entire form, module, or class and all its members; if the pragma appears inside a method, then it affects the current method and all its local variables:

もしプラグマが「**Project:**」またはメンバ名に接頭辞として付けられない場合、その範囲は VB6 コードがどこに現れるかによります。そのスコーピングルールは VB6 と同じです。もし Pragma が class-、form-、または module-level で現れるのであれば、メソッドの内部ではありません。それは、全体の Form、モジュール、またはクラス、そして全てのそのメンバに影響を与えます。もし Pragma がメソッドの内部に現れる場合は、カレントメソッドと全てのそのローカル変数に影響を与えます。

```
Sub Test()
```

```
        ' this pragma affects all local variables in Test method
        '## AutoNew True
        ...
    End Sub
```

The effect of a pragma can be overridden by a pragma with a narrower scope. For example, you can use a project-level AutoNew pragma that affects all the fields and variables, except those that are affected by AutoNew pragmas at the class, method, or variable level. This hierarchical mechanism adds a lot of flexibility and lets developers precisely define the outcome from VB Migration Partner with few additions to the original VB6 code.

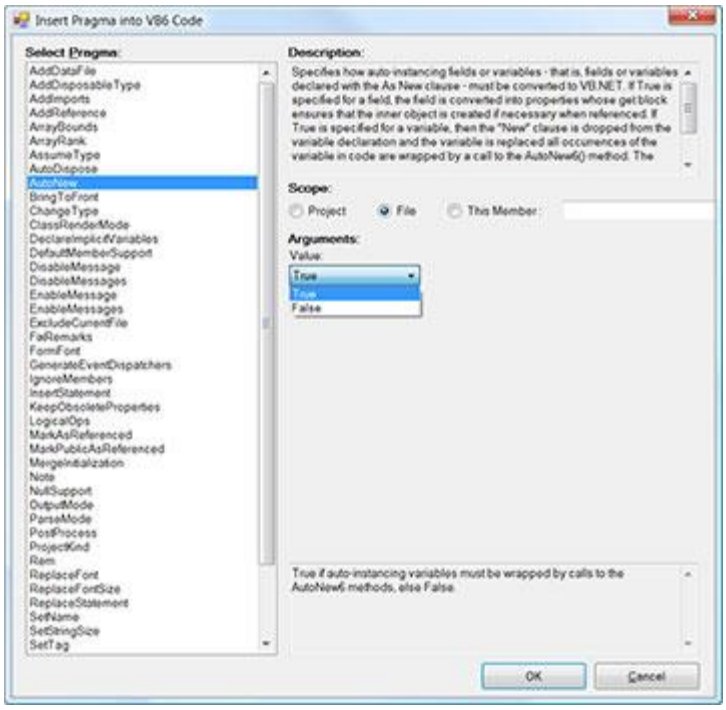
プラグマの効果は狭い範囲でプラグマによって上書きできることです。例えば、クラス、メソッドまたは変数レベルで AutoNew プラグマによって影響を受けるもの以外は、全てのフィールドと変数に現れるプロジェクトレベルの AutoNew プラグマを使うことができます。この階層的メカニズムはたくさんの柔軟性を追加し、開発者が VB Migration Partner にいくつか追加したものからオリジナルの VB6 コードに結果を正確に定義することができます。

VB Migration Partner checks the syntax of all pragmas and doesn't support pragmas with arbitrary names; however, we provide a one-size-fits-all pragma named SetTag, which developers can use to associate values to code entities. The SetTag pragma is especially useful with extensions.

VB Migration Partner は全てのプラグマの構文と任意の名前のサポートしないプラグマをチェックします。しかし、SetTag という名前の万能なプラグマを提供しています。開発者はコード自体に値を関連付けさせることができます。SetTag プラグマは拡張性を持ち非常に便利です。

You can easily insert new pragmas by means of a dialog box that explains what each pragma does and what each argument means, and that ensures that the syntax is correct.

ダイアログボックスで新しいプラグマを挿入するのはとても簡単です。それぞれのプラグマと引数の意味を説明し、構文が正しいことを保証します。



In addition to processing pragmas in VB6 source code files, VB Migration Partner looks for the following files:

VB6 ソースコードファイルで処理をするプラグマに加えて、VB Migration Partner は以下のファイルを検索します。

1. A file named **VBMigrationPartner.pragmas**, in VB Migration Partner's main directory. (This is known as the "master" pragma file.)
   VBMigrationPartner のメインディレクトリに入っている、**VBMigrationPartner.pragmas** という名前のファイル。（これは「マスター」プラグマファイルです）
2. A file named after the project's file and with the **.pragmas** extensions - for example, Widgets.vbp.pragmas for the Widgets.vbp project.
   プロジェクトファイル名の後に**.pragmas** 拡張子で付けられるファイル。例：Widgets.vbp プロジェクトでは Widgets.vbp.pragmas
3. A file named **VBMigrationPartner.pragmas**, in the same directory as the project's .vbp file. (This file is processed only if the search for previous file fails.)
   プロジェクトの.vbp ファイルと同じディレクトリにある **VBMigrationPartner.pragmas** という名前のファイル。（前のファイルの検索が失敗したときのみ、このファイルが処理されます。）

Storing project-level pragmas inside these files is necessary or convenient in two cases. First, you can store project-level **PreProcess, ImportTypeLib,** and **AddLibraryPath** pragmas only inside these files. Second, this mechanism allows you to easily share pragma among different projects.

これらのファイルの中にプロジェクトレベルプラグマを保管するのは、次の2つのケースにおいて必要であるか、または便利です。最初に、プロジェクトレベルの **PreProcess**、**ImportTypeLib**、**AddLibraryPath**、をこれらのファイルの内部に保管できます。2つ目にこのメカニズムを異なるプロジェクトの間で簡単にプラグマを共有して使用することができます。

For example, you can ensure that all the form fonts in multiple projects are converted in the same way by creating a file named VBMigrationPartner.pragmas containing this text:

例えば、多重プロジェクトにおける全てのフォームフォントを次のテキストを含んでいる VBMigrationPartner.pragmas というファイル名を作成することにより同じように変換されるということを保証します。

```
'## FormFont Arial, 10
```

and then copying it to all the directories that contain the projects you plan to convert. Notice that the **project:**prefix is optional for pragmas stored in *.pragmas files.

そして、次にプロジェクトを含むすべてのディレクトリにそれをコピーし、変換の計画を行います。プロジェクトに対する注意点：*.pragmas ファイルに保存されたプラグマに接頭語「**project:**」は任意です。

Keep in mind that the “master” pragma file in VB Migration Partner’s main directory (step 1) is always processed, whereas the VBMigrationPartner.pragmas file in the VB6 project’s folder (steps 3) is processed only if the search at step 2 fails. The order in which these files are processed ensures that the settings in files inside the project’s folder can override the settings specified in the “master” pragma file. For example, if both the “master” pragma file and the pragma file in the project’s folder contain an **ImportTypeLib** pragma that refers to the same type library, the setting specified in the latter file wins.

VB Migration Partner のメインディレクトリ（Step1）に入っている「マスター」プラグマは常に処理されます。それに対して VB6 のプロジェクトフォルダ（Step3）の中にある VBMigrationPartner.pragmas が Step2 の検索に失敗した場合にのみ処理されます。これらのファイルが処理される順番は、プロジェクトのフォルダにある設定ファイルが「マスター」プラグマファイルの特定の設定を優先できることを保証します。例えば、もし「マスター」プラグマファイルとプロジェクトフォルダ内のプラグマファイルの両方が同じタイプのライブラリを参照している **ImportTypeLib** プラグマを含んでいるのであれば、後者のファイルの特定の設定が優先されます。

2. Using VB Migration Partner／VB Migration Partner を使用する

2. Using VB Migration Partner／VB Migration Partner を使用する

VB Migration Partner requires that Microsoft Visual Studio 2005 or Visual Studio 2008 be installed on the local computer. We recommend that you run the migration process on the same computer where you developed and tested the original VB6 application, therefore also Visual Basic 6 must be installed on the local computer.

VB Migration Partner はローカルコンピュータに Microsoft Visual Studio 2005 または Visual Studio 2008 がインストールされている必要があります。オリジナル VB6 アプリケーションを開発やテストしていた同じコンピュータ上に変換プロセスを行うことをお勧めします。その為には、Visual Basic 6 がローカルコンピュータにインストールされていなければなりません。

Using the program is quite simple and revolves around a few simple actions that can be reached from the toolbar.

このプログラムを使用するのはとても簡単で、シンプルなアクションを考慮されています。それらはツールバーから利用することができます。

2.1 Loading the VB6 project／VB6 プロジェクトの読み込み

Select the File–Open menu command or click the Open toolbar button to load a VB6 project (.vbp) or project group (.vbg). If the project was written with a version of VB6 prior to VB6 you must convert it to VB6 before attempting the conversion. If the loaded file isn't recognized as a VB6 project or project group a message error is displayed.

「File メニュー　Open」を実行するか、もしくはツールバーの「Open」ボタンをクリックして VB6 プロジェクト(.vbp)またはプロジェクトグループ(.vbg)を選択してください。もし対象のプロジェクトが VB6 以前の Version で書かれている場

合、変換を試す前に VB6 へ VersionUp しておく必要があります。読み込まれたファイルが VB6 プロジェクトもしくはプロジェクトグループとして認識されない場合、メッセージエラーが表示されます。

If the EXE or DLL file created by compiling the VB6 project is present, VB Migration Partner compares its datetime stamp with the datetime stamp of all the files in the project. If the executable file is older than any of the corresponding source files, VB Migration Partner displays a warning.

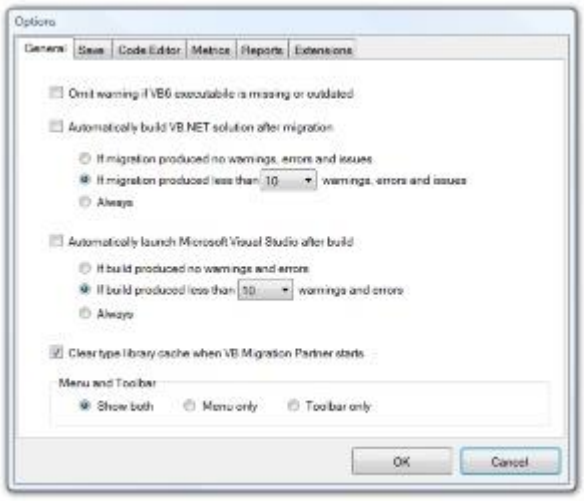
VB6 プロジェクトによるコンパイルされた EXE や DLL ファイルがあれば、VB Migration Partner はプロジェクト内の全てのファイルの DateTime スタンプを比較します。もし実行ファイルが対応する Source ファイルより古い場合は、VB Migration Partner は警告を表示します。

Warning: We strongly recommend that you always recompile the original VB6 project before attempting the conversion to VB.NET and ensure that the VB6 code doesn't contain syntax errors. While VB Migration Partner is able to spot such errors and solve them in most cases, in some cases these errors might cause invalid VB.NET code to be emitted and even crash VB Migration Partner.

警告: VB.NET に変換を試みる前に、オリジナルの VB6 プロジェクトをリコンパイルし、構文エラーがないことをまず確認してから変換を行うことを強く推奨します。VB Migration Partner がそのようなエラーを見つけられる間は、ほとんどのケースを解決できますが、いくつかのケースにおいてはこれらのエラーは無効な VB.NET コードが放出し、VB Migration Partner をクラッシュさせるかもしれません。

Note: you can suppress the display of this message box by selecting the corresponding option in the General tab of the Tools–Options window.

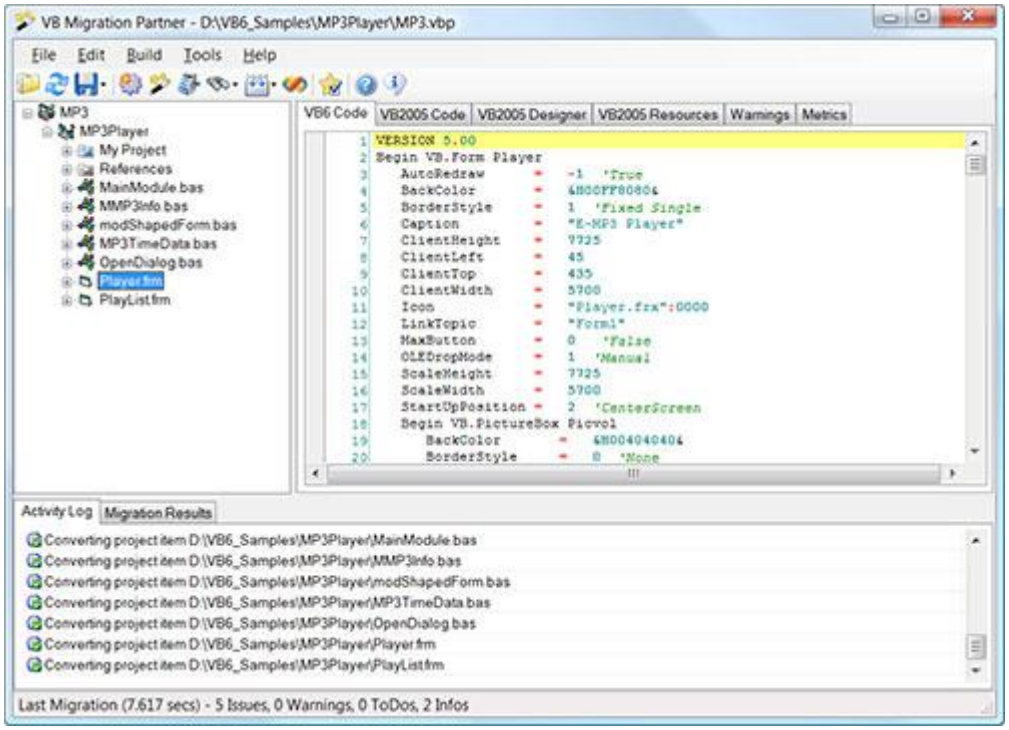
注記: このメッセージボックスの表示については、「Tools メニュー　Options」ウインドウの「General」タブにある対応したオプションを選択することによって表示を制御することができます。



2.2 Converting to VB.NET／VB.NET に変換

Select the Build–Convert to VB.NET menu command or click the Convert toolbar button to start the conversion process. Each source file undergoes three distinct migration stages: the parsing stage, the processing stage, and the conversion stage. During the process an activity log is created, so that you can watch which files are being converted.

変換プロセスを始めるために、「Build メニュー　Convert to VB.NET」を実行するか、ツールバーの「Convert」ボタンをクリックしてください。それぞれのソースファイルは3つの異なった段階を経ます。解析段階、処理段階、変換段階です。アクティビティログが作成されている処理の間は、どのファイルが変化されているのか見ることができます。



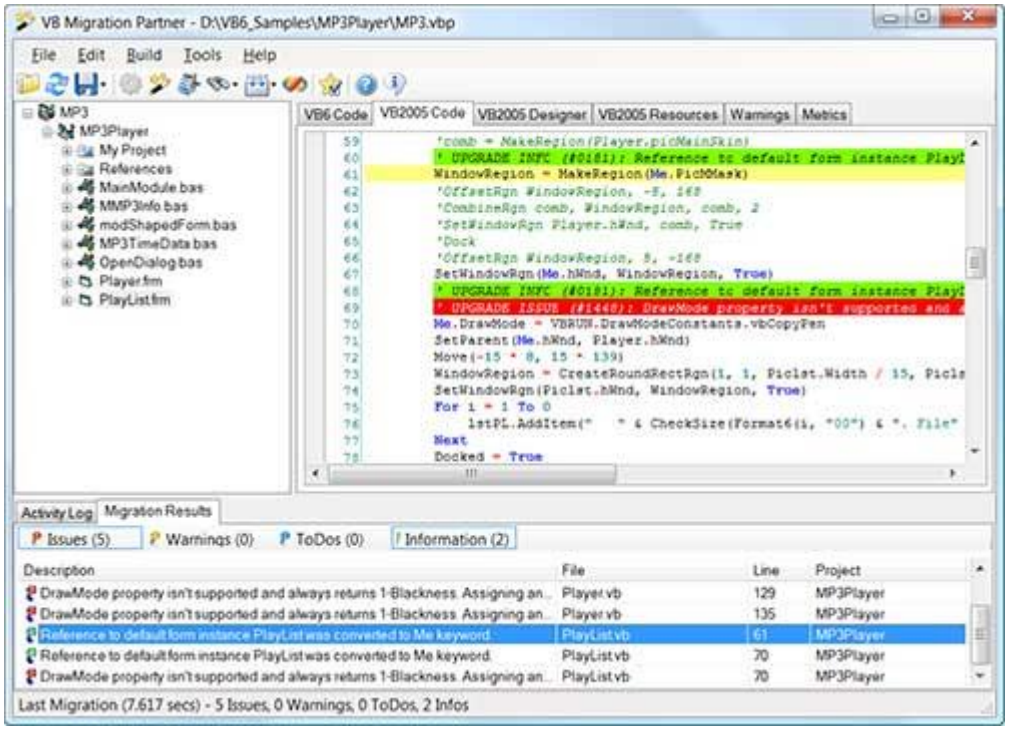
When the migration process is over, a few new tabs appear in the rightmost pane of VB Migration Partner's main window:

変換作業が終了した際に、VB Migration Partner のメインウィンドウの一番右枠にいくつかのタブが表示されます。

- the VB.NET Code tab shows how the code in the currently selected file has been migrated
  VB.NET Code タブは現在選択されているファイルのコードがどのように変換されたかを表示します。
- the Warnings tab shows all the warnings and issues for the entire project or file that is currently selected
  Warnings タブは現在選択されている全体プロジェクトもしくはファイルの全ての警告と問題を表示します。
- the Metrics tab shows code statistics for the entire project or file that is currently selected
  Metrics タブは現在選択されている全体プロジェクトもしくはファイルのコード統計を表示します。

The Migration Result pane displays the list of all the issues, warnings, info, and to–do messages that have been emitted as special remarks in the VB.NET code. You can filter these messages by clicking on one of the four tabs at the top of this pane and you can double–click a message to quickly jump to the code portion where the message has been emitted.

Migration Results 枠は VB.NET コードの特記事項として出力され「Issues」、「Warnings」、「ToDos」、「Information」としてリスト表示されます。この枠の上部にある4つのタブのひとつをクリックすることによって、これらのメッセージをフィルタすることができます。そしてそのメッセージをダブルクリックすることで、そのメッセージが作成されたコード箇所にダイレクトにジャンプします。



2.3 Compiling the VB.NET solution／VB.NET Solution のコンパイル

Select the Build–Compile Entire Solution menu command or click the Compile toolbar button to compile all the projects in the current solution to VB.NET without leaving VB Migration Partner. You can also compile individual projects, by means of the Build–Compile Selected Project menu command.「Build メニュー　Compile Entire Solution」を実行するか、ツールバーの「Compile」ボタンをクリックしてください。現在の Solution の全てのプロジェクトを VB.NET に VB Migration Partner を離れることなくコンパイルできます。また個々のプロジェクトを「Build メニュー　Compile Selected Project」を実行することでコンパイルすることもできます。

Before running the actual compilation, VB Migration Partner has to save VB.NET source code files to disk. By default, these files are stored in a folder in the same directory as the folder that contains the original VB6 project; the name of the new folder is obtained by appending “_NET” to the name of the original folder. You can modify such default behavior in the Save tab of the Tools–Options dialog.

実際のコンパイルを実行する前に、VB Migration Partner は VB.NET のソースコードファイルをディスクに保存する必要があります。デフォルトでは、オリジナルの VB6 プロジェクトが格納されているフォルダと同じ場所のフォルダに格納されます。新しいフォルダの名前はオリジナルフォルダ名に“_NET”を追加することによって決まります。「Tools メニュー　Options」を実行したダイアログの「Save」タブの中でそのようなデフォルトの動作を変更することができます。

Before proceeding, VB Migration Partner displays a dialog that allows you to select a different directory:

作業の前に、VB Migration Partner は異なるディレクトリを選択できるダイアログを表示します。

If the target directory already exists and contains files – presumably created by a previous migration attempt – VB Migration Partner displays a message box that asks you to confirm the selection.

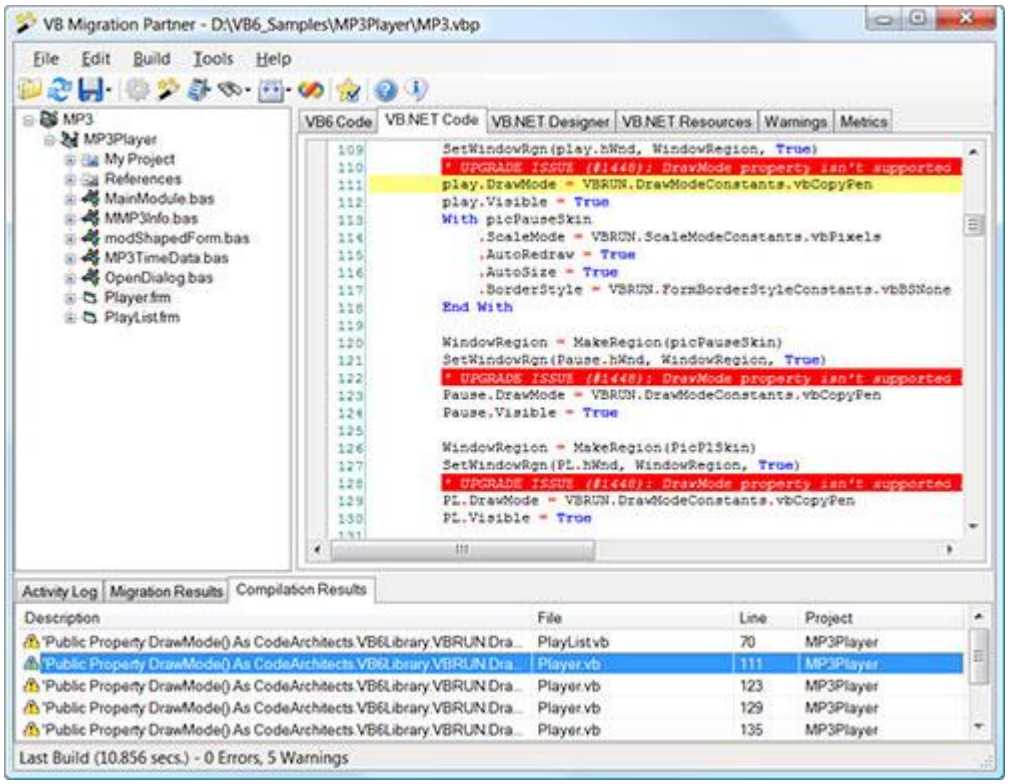
ターゲットディレクトリが既に存在しファイルが含まれている場合は、恐らく過去に移行していることが考えられます。その場合、VB Migration Partner は選択したフォルダの場所を確認依頼するメッセージボックスを表示します。

If the default choice is OK, you can tell VB Migration Partner not to display the dialog by enabling the Automatically Select Output Folder For VB.NET Solutions option, in the Tools–Options window. Likewise, you tell VB Migration Partner not to display the message box by enabling the Automatically Overwrite Existing Output Folder option, in the same window.

デフォルト選択で OK な場合は「Tools メニュー Options」を実行したダイアログの「Save」タブ内の「Options」枠、「Automatically select output folder for VB.NET solutions」チェックボックスを有効にすることによってダイアログを表示させないこともできます。同様に VB Migration Partner に同じウィンドウ内にある「Automatically overwrite existing output folder」チェックボックスを有効にすることによって、メッセージボックスを表示させないこともできます。

Once the compilation process is completed, you can browse all the errors and warnings from the VB.NET compiler in the Compilation Results pane, near the bottom border.

コンパイルプロセスが完了した際には表示下部にある「Compilation Results」枠内に VB.NET コンパイラからの全ての警告とエラーを参照することができます。



Note: the compilation process is optional, especially when migrating a simple VB6 project or a project that you have already migrated previously and that has been already fixed to avoid compilation errors. In such cases you can directly load the converted code in Microsoft Visual Studio.

注記:コンパイル処理はオプションです。特に簡単な VB6 プロジェクト移行している時、または既に以前移行したコンパイルエラーを修正済のプロジェクトの場合は有効です。そのようなケースにおいては変換されたコードを Microsoft Visual Studio にダイレクトに読み込むことができます。

Alternatively, you can direct VB Migration Partner to compile the VB.NET code immediately after the migration process, by means of the Automatically Build VB.NET Solution After Migration option, in the Tools–Option window. You have the option to compile unconditionally or only if the conversion process generated fewer errors, warnings, and issues than a threshold that you specify.

また別の方法としては、「Tools メニュー Options」を実行したダイアログの「General」タブ内の「Automatically build for VB.NET solution after migration」チェックボックスを有効にすることによって、変換作業直後に VB.NET コードをコンパイルするように VB Migration Partner に指示することができます。また無条件にコンパイルするように設定したり、変換プロセスにおいて自身が設定した閾値より下回った警告、エラー、問題数の場合にのみコンパイルするように設定することもできます。

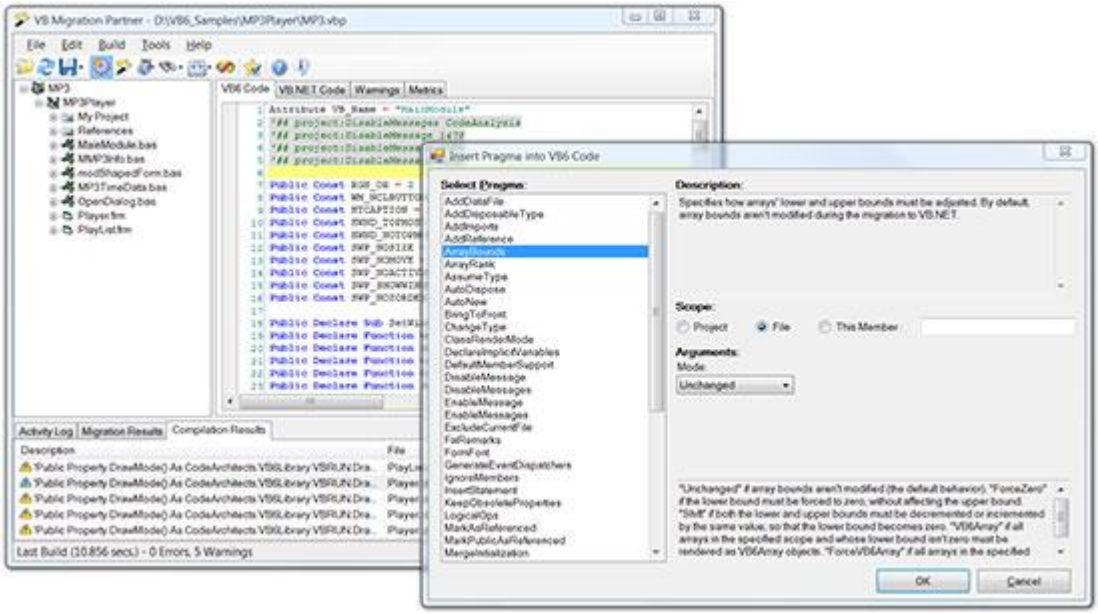
2.4 Fixing the VB6 code／VB6 コードの修正

Except for trivial applications, you *never* come up with a working VB.NET application at the first attempt. Migrating VB6 code to the .NET Framework platform is better described as an iterative process: you are expected to go back

to the original application, edit it to fix all the migration and compilation errors and issues, and convert the code once again. This is the so-called *convert-test-fix* cycle.

小規模のアプリケーション以外では変換後の最初の試みにおいて、VB.NET アプリケーションとして変換前と同様に動作することはあまりありません。VB6 コードを.NET Framwork Platform に変換することは、以下に示す繰り返し行うプロセスがより適していると言われています。全てのコードの移行とコンパイルエラーと問題を修正し、再びコードを変換する作業をすることによって、オリジナルのアプリケーションに戻ることを期待します。この一連の作業を「convert-test-fix」サイクルと呼びます。

If the converted application has one or more compilation errors you should probably insert one or more pragmas in the original VB6 code. You can do so by selecting the VB6 Code tab and the Edit-Insert Pragma menu command or by clicking on the Insert Pragma toolbar button. This action displays the Insert Pragmas dialog, which assists you in picking up the right pragma and assigning the right arguments. When you click the OK button, the dialog is closed and the actual pragma is inserted at the current position in the VB6 code.
もし変換されたアプリケーションがひとつもしくはそれ以上のコンパイルエラーがあるならば、恐らくひとつもしくはそれ以上の Pragmas をオリジナルの VB6 コードに追加すべきです。VB6 Code タブを選択し「Edit　メニュー　Insert Pragma」を実行するかツールバーの「Insert Migration Pragma」ボタンをクリックすることで実行できます。この動作は Insert Pragmas ダイアログを表示し、適切な Pragma を見つけ、そして適切な引数を割り当てるように補助します。OK ボタンをクリックするとダイアログが消え、VB6 コードの現在の位置に実際の Pragma が挿入されます。



If you have added one or more pragmas, or have modified the original VB6 code in any way from inside VB Migration Partner, you should now save the new files to disk, by selecting the File-Save-Save VB6 Files menu command or by clicking on the Save toolbar button.

ひとつもしくはそれ以上の Pragma を追加または、VB Migration Partner よりオリジナルの VB6 コードを修正したことがある場合は、「File メニュー Save の Save VB6 Files」を実行するかツールバーの「Save」ボタンをクリックし新しいファイルをディスクに保存してください。

Alternatively, if you have modified the original VB6 code from outside VB Migration Partner – For example, from inside Microsoft Visual Basic 6 IDE or from an external editor – you can reload the project or project group by selecting the File–Reload menu command or by clicking on the Reload toolbar button.

別な方法として、VB6 のオリジナルコードを VB Migration Partner の外部より修正したことがある場合、たとえば Microsoft Visual Basic 6 IDE もしくは外部エディタからの修正は、「File メニュー Reload」を実行するかツールバーの「Reload」ボタンをクリックし Project もしくは Project Group を読み込むことができます。

2.5 Launching Visual Studio／VisualStudio の起動

When you are satisfied of the results from the migration process you can load the converted VB.NET solution inside Microsoft Visual Studio, by selecting the Tools–Run Microsoft Visual Studio menu command or by clicking the Load the generated VB.NET code in Microsoft Visual Studio toolbar button.

Migration プロセスの結果に問題がなければ、「Tools メニュー Run Microsoft Visual Studio」を実行するかツールバーの「Load the generated VB.NET code in Microsoft Visual Studio」ボタンをクリックし、変換された VB.NET Solution を Microsoft Visual Studio に読み込むことができます。

VB Migration Partner displays the dialog box that allows you to select the target folder – as for the Compile to VB.NET command – then it launches Microsoft Visual Studio and loads the converted VB.NET in it.

VB Migration Partner は VB.NET にコンパイルするためのコマンドとして、対象のフォルダを選択するようにダイアログボックスを表示します。その後、Microsoft Visual Studio を起動し、変換された VB.NET を読み込みます。

The first thing to do at this point is checking all the errors and warnings in Visual Studio's Error List window, and then all the items in the Task List window.

この時点で最初にすることは、Visual Studio の Error List ウィンドウにすべてのエラーと警告をチェックすることと、Task List ウィンドウにすべてのアイテムがあることをチェックします。

VB Migration Partner can generate for different type of comments in the converted application:

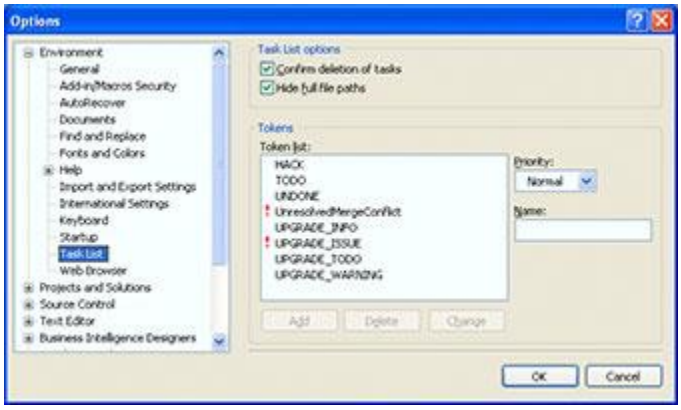
VB Migration Partner は変換されたアプリケーションに異なるタイプのコメントを生成することができます。

- **UPGRADE_ISSUE:** serious migration issues that you should solve immediately
  UPGRADE_ISSUE： 速やかに改善すべき深刻な移行問題点。
- **UPGRADE_WARNING:** migration warnings that might or might not affect the converted application
  UPGRADE_WARNING： 変換されたアプリケーションとして、影響があるかどうかわからない移行の警告。

- **UPGRADE_TODO:** suggestions about how to manually edit the migrated code to avoid a potential problem
  UPGRADE_TODO： 潜在的な問題を回避するために、手動でコードを編集することを提案。
- **UPGRADE_INFO:** information about the generated VB.NET code, including information about unused constant and methods and recommendations about the .NET type or methods that can replace a Declare statement.
  UPGRADE_INFO： 生成した VB.NET コードについての情報。使用されていない定数やメソッド、宣言文に置き換えることができる.NET の型やメソッドについての提案。

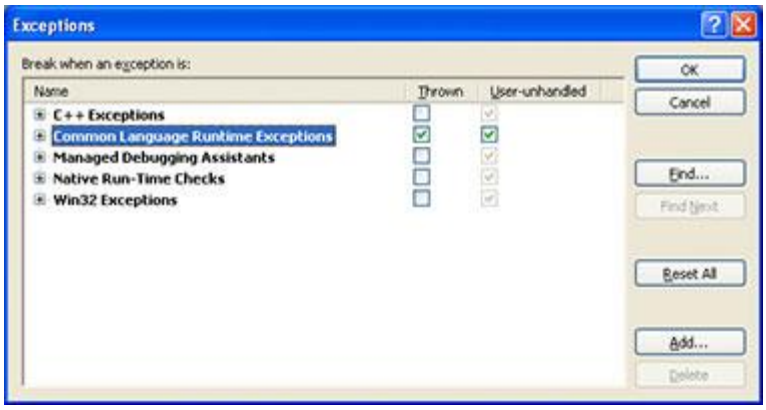
By default, only warnings and to-do comments appear in Visual Studio's Task List pane. We suggest that you use the Tools-Options command to add the UPGRADE_ISSUE comment to the list of comments that the Task List recognizes and set its importance to High.

デフォルトでは警告と ToDo コメントのみ Visual Studio のタスク一覧画面に表示されます。.NET の「ツール　メニュー　オプション」を実行し、「UPGRADE_ISSUE」コメントをコメントリストに追加し、タスク一覧が認識できるように優先順位を「高」にしてセットすることをお勧めします。



If there are no compilation errors you can run the VB.NET application and see how it behaves. It's unlikely that a complex application runs smoothly at the first attempt, therefore be prepared for runtime exceptions. We recommend that you enable the option Break When An Exception Is Thrown in the Debug-Exceptions dialog box, so that you immediately catch unexpected runtime errors that would go unnoticed because of an On Error statement.

コンパイルエラーがなければ、VB.NET アプリケーションを実行し、どのように動作しているか見ることができます。複雑なアプリケーションにおいては最初の起動でスムーズにいくことはあまりありませんのでランタイムエラーが起こることを想定しておいてください。「デバッグ　メニュー　例外」を実行し、ダイアログボックスの Common Language Runtime Exceptions の「スローされるとき」チェックボックスを有効にすることをお勧めします。その設定により、On Error Statement による気がつかない、予期せぬランタイムエラーが起こった際にすばやくキャッチすることができます。



2.6 Using code analysis features／コード分析機能を使用

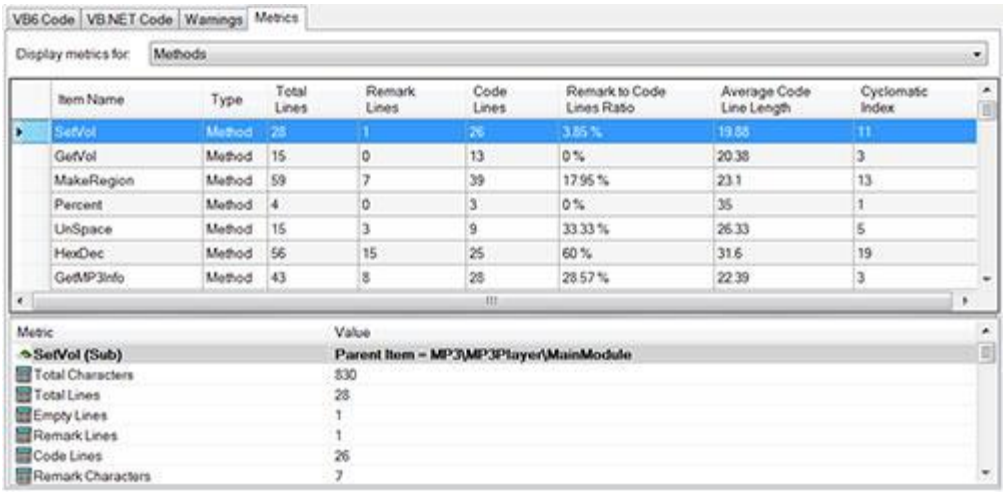
At the end of the conversion process VB Migration Partner displays a detailed report in the Metrics tab in the right portion of the main window.

変換プロセスの終わりにVB Migration Partnerはメインウィンドウの右側にある「Metrics」タブの中に詳細なレポートを表示します。

1. Select an item in the tree on the left, to display code metrics at the solution, project, or file level.
   Solution、Project、または File レベルにおけるコード指標を表示させるために、左にあるツリーの中のアイテムを選択してください。
2. Select an item from the combobox to further restrict the report to just forms, classes, public or private members, methods or properties, and so forth.
   レポートに制限させるために、フォーム、クラス、パブリックまたはプライベートのメンバ、メソッドやプロパティなどの項目をコンボボックスから選択します。
3. Select an item in the grid to see a more detailed report in the lower area of the Metrics tab.
   表の中の項目を選択すると、Metrics タブの下段にあるより詳細なレポートを見ることができます。
4. Sort the items in the grid in either ascending or descending order, by clicking on a column header.
   表の中の項目で列ヘッダをクリックすることで、各項目列の昇順、降順の並び替えをすることができます。

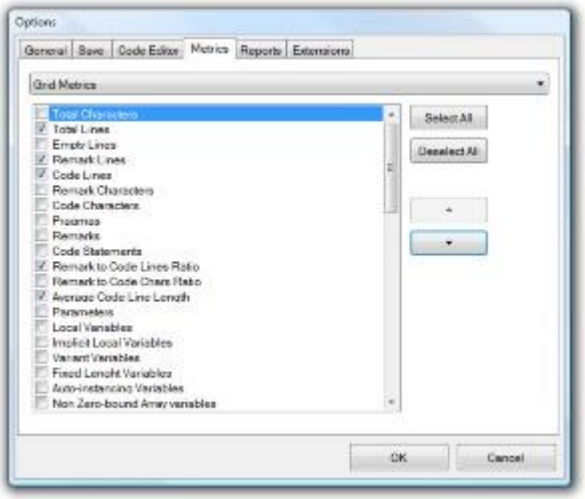
The sorting feature is especially useful when focusing on the most problematic portions of the VB6 solution to be converted. For example, you can sort all methods on their cyclomatic index in descending order to immediately find the most complex methods in the application.

ソート機能は変換された VB6 ソリューションで特に最も問題となる部分に焦点をおくように利用すると便利です。たとえば、アプリケーションの中で最も複雑なメソッドを見つけるために、「Cyclomatic Index」にあるすべてのメソッドを降順で並び替えることができます。



The Metrics tab of the Tools–Options dialog allows you to configure which code metrics appear in the grid and in the area underneath the grid.

「Tools –> Options」を実行し、Options ダイアログの「Metrics」タブでグリッドの下にあるエリアにどのコード指標を表示させるか設定することができます。



The name of most code metrics values is self–explanatory, but a few might require an explanation.

ほとんどのコード統計値の名称は一目瞭然かと思いますが、いくつか説明をします。

- **Total Lines** is the sum of all code lines in the file, project, or solution. It includes the lines in the hidden portion of .frm and .ctl files.
  Total Lines は File、Project、Solution のすべてのコード行の合計です。.frm と.ctl の隠れたコードも行数に含みます。
- **Code Lines** is the number of lines that contain actual executable code; it doesn't include empty and remark lines (which are counted by separate code metrics).
  Code Lines は実際の実行コードを含むコード行数です。空行とコメント行は含みません。(分割したコード統計によりカウントされます)
- **Remark to Code Lines Ratio** provides a broad measure of how well the original VB6 project is documented; the higher this value, the better.

Remarks to Code Lines Ratio はオリジナルの VB6 のドキュメントに記載されているものよりもより高く、より広範囲に測定したコード比率を提供します。

- **Implicit Local Variables** is the number of local variables that aren't explicitly declared; they are typically converted as Object variables, therefore you might need to add the original VB6 code to use a more efficient type.

  Implicit Local Variables は明示的に宣言されていないローカル変数の数です。それらは通常オブジェクト変数として変換されます。そのために、より効果的な型を使用するようにオリジナルの VB6 コードに追加する必要があるかもしれません。

- **Variant Variables** is the number of Variant variables; they are migrated as Object variables, therefore it is recommend that you carefully scrutinize each of them and change them to a more efficient type if possible, or possibly use a SetType pragma to convert them to VB6Variant variables.

  Variant Variables はバリアント型変数の数です。それらはオブジェクト変数として移行されます。そのために、それらをひとつひとつ注意深く精査し、より効果的な型に変えることを勧めます。できれば、VB6Variant 変数にそれらを変換するように SetType　Pragma を使用してみてください。

- **Fixed Length Variables** is the number of fixed-length strings. By default they are converted to instances of the VB6FixedLength class, but you might want to edit the original VB6 code (or use a UseSystemString pragma) to transform them into regular string.

  Fixed Length Variables は固定長型変数の数です。デフォルトではそれらは VB6FixedLength クラスのインスタンスに変換されます。しかし、通常の String に変換するためにオリジナルの VB6 コードを編集して下さい。（UseSystemString　Pragma を使うこともできます）

- **Auto-Instancing Variables** is the number of variables declared with the "As New" keyword. These variables have a different semantics in VB.NET and might behave differently.

  Auto-Instancing Variables は「As New」キーワードで宣言されている変数の数です。これらの変数は、VB.NET では異なる意味を持っており、違った動作をすることもあります。

- **Non-Zero Bound Array Variables** is the number of arrays whose lower index is nonzero; such arrays require a pragma to compile correctly under VB.NET.

  Non-Zero Bound Array Variables は最小 Index が Non-Zero の配列の数です。VB.NET 配下でそのような配列を正しくコンパイルするための Pragma が必要です。

- **Gotos, Gosubs, On Gotos and Gosubs** are the total number of GoTo keywords, GoSub keywords, and calculated GoTo/GoSub keywords, respectively. VB.NET supports the GoTo keyword and VB Migration Partner generates code that simulates GoSubs and calculated GoTo/GoSub; nevertheless, it is strongly recommended that you edit the original VB6 application to get rid of them.

  Gotos, Gosubs, On Gotos and Gosubs は Goto キーワード、Gosub キーワード、計算された GoTo/GoSub キーワードの合計数です。それぞれ VB.NET は GoTo キーワードをサポートし、VB Migration Partner はシミュレートされた GoSubs、そして計算された GoTo/GoSub をコード生成します。しかしながら、それらを除去するためにオリジナルの VB6 コードを編集することを強く推奨します。

- **On Errors** and **Resumes** are the number of On Error statements and of Resume/Resume Next statements. VB.NET supports them, but you should replace them with more structured and efficient Try…Catch blocks.

  On Errors and **Resumes** は On Error ステートメント、Resume/Resume Next ステートメントの数です。VB.NET はそれらをサポートします。しかし、より構造的かつ効果的に Try･･･Catch ブロックに置き換えて下さい。

- **File Operations** is the number of Open, Get#, Put#, and other file-related statements. These keywords don't behave in exactly the same way in VB6 and VB.NET and VB Migration Partner doesn't automatically account

for all these differences, therefore you might need to carefully test each of them.

File Operations は Open、Get#、Put#、その他ファイル関連ステートメントの数です。これらのキーワードは、VB6 と VB.NET ではまったく同じ動作をするとは限りません。VB Migration Partner はこれらの違いを自動的に把握するわけではないので、注意深くテストを十分に行う必要があります。

- **Exit Points** is the number of Exit Sub, Exit Function, and Exit Property keywords. This value is included in the code metrics report because many developers prefer to have a single exit point for each method.

  Exit Points は Exit Sub、Exit Function、Exit Property キーワードの数です。ほとんどの開発者はそれぞれのメソッドにひとつの Exit があることを好むために、この値をコード分析レポートに含めました。
- **Cyclomatic Index** is the number of all possible code execution paths in a method and is therefore equal to the number of tests that should be performed to prove that the method behaves correctly in all situations. When evaluated at the file, project, or solution level it returns the sum of cyclomatic index of all contained methods and therefore can be assumed as a broad measure of the overall complexity of that file, project, or solution.

  Cyclomatic Index はメソッド内のすべてのコードが実行されるパスの数です。この数はすべての状況下でメソッドが正確に動作することを証明するために実行すべきテストの数とイコールになります。File、Project、Solution レベルを評価するときに、すべての含まれたメソッドのサイクロマティック指数の合計を返します。したがって、File、Project、Solution の広範な複雑性の測定として想定することができます。
- **Nesting Level** is the maximum nesting level of blocks inside a method. For example, a method that contains a For loop that contains an If block has a nesting level equal to 2. When evaluated at the file, project, or solution level it returns the sum of nesting level of all contained methods.

  Nesting Level はメソッド内部のブロックのネストしているレベルの最大値です。たとえば、For Loop を含んだメソッドが含んでいるブロックをネストレベル2に相当します。File、Project、Solution レベルを評価するときに、すべての含まれたメソッドのネストしているレベルの合計を返します。
- **If Directives** is the number of #If, #ElseIf, and #Else keywords. VB Migration Partner is capable to evaluate #If conditions and converts only the portion of code that is contained in the “true” portion of the #If block. All other sections must be converted manually, therefore it is a good idea to revise the original VB6 application and ensure that #If and #Const expressions exactly define the code that you want to convert.

  If Directives は#If、#ElseIf、#Else キーワードの数です。VB Migration Partner は If 条件を評価することができ、If ブロックの“True”部分に含まれているコードの部分だけを変換します。その他すべてのセクションは手動による変換を行って下さい。オリジナルの VB6 アプリケーションを修正し、#If と#Const 表現で変換したいコードを正しく定義することで動作を保証します。

2.7 Using assessment features／評価機能を使用

If the VB6 project has been already converted, VB Migration Partner can generate an assessment report and export it Microsoft Excel. The report contains detailed information about each project in the original application, including code metrics, number of migration issues and warnings, and an estimation of time (and money) required to complete the conversion process.

もし VB6 プロジェクトが既に変換されているのであれば、VB Migration Partner は評価レポートを生成し、Microsoft Excel にてエクスポートすることもできます。そのレポートはオリジナルアプリケーションのそれぞれの Project につい

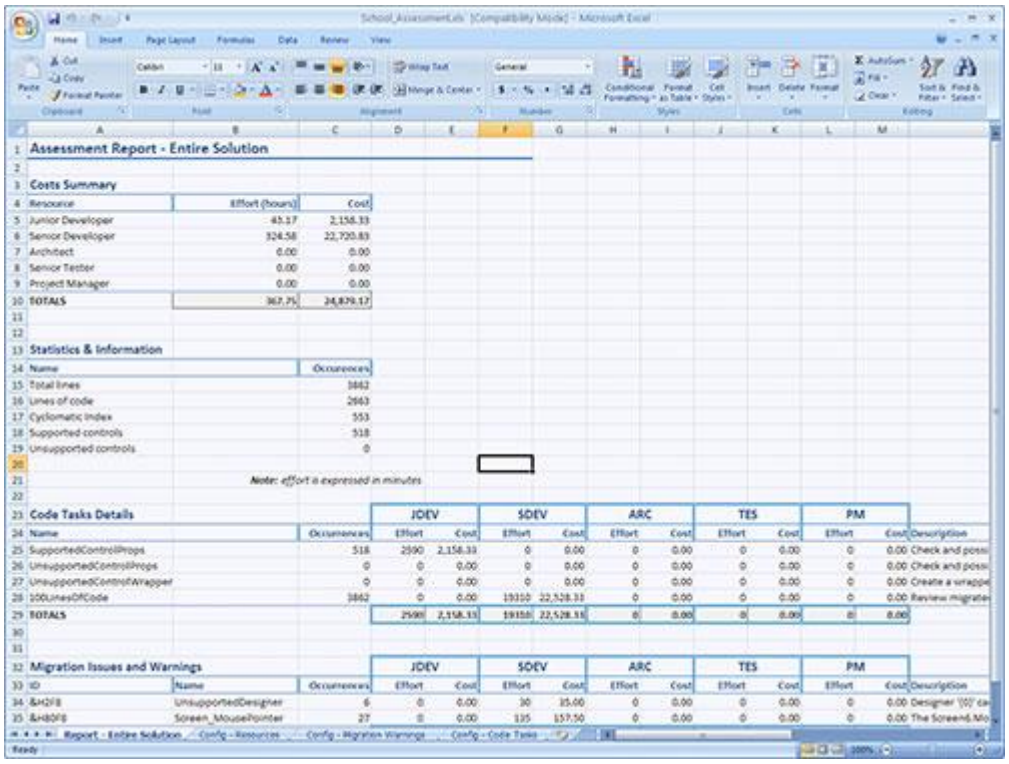
て詳細な情報を含みます。ほかにコード分析、移行の警告と問題の数、変換プロセスを完成させるに必要な見積時間（費用）などもレポートに含まれます。

Run the assessment feature by selecting the Tools–Generate Assessment Report menu command or by clicking on the Assessment toolbar button.

「Tools　メニュー　Generate Assessment Report」を実行するか、ツールバーの「Generate the assessment report」ボタンをクリックして評価機能を起動してください。

VB Migration Partner lets you select the name of the target Microsoft Excel file and then creates the assessment report. This step can take several seconds, or even minutes, for long and complex VB6 applications. At the end of the process a message box allows you to load the generated report inside Microsoft Excel.

VB Migration Partner は Microsoft Excel ファイル名を指定し、評価レポートを生成します。この手順には長くて複雑な VB6 アプリケーションでは数秒もしくは数分かかります。プロセスの最後に生成されたレポートを Microsoft Excel に読み込ませるかどうかのメッセージボックスを表示します。



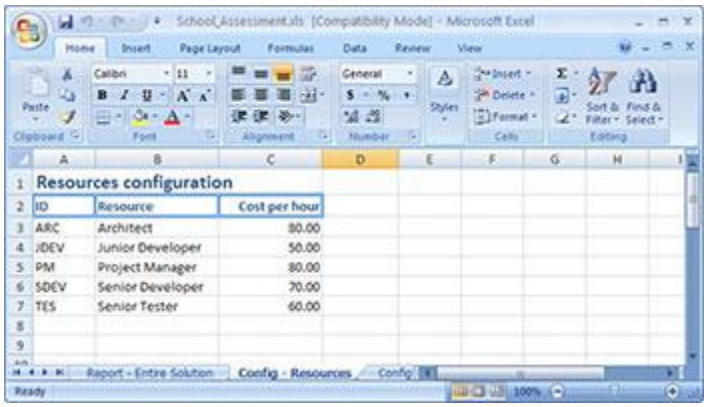
The first worksheet generated Microsoft Excel file provides a summary of estimated costs for the entire application. If the VB6 application contains two or more project, a separate worksheet is generated for each project.

生成された Microsoft Excel の最初のワークシートは、アプリケーション全体の費用見積合計を提供します。もし VB6 アプリケーションが2つもしくはそれ以上の Project を含んでいる場合は、ワークシートを分割し、それぞれの Project 毎に生成します。

The “Config – Resources” worksheet allows you to define five different developer roles and assign a different hourly cost to each of them. The predefined roles are Project Manager, Architect, Junior Developer, Senior

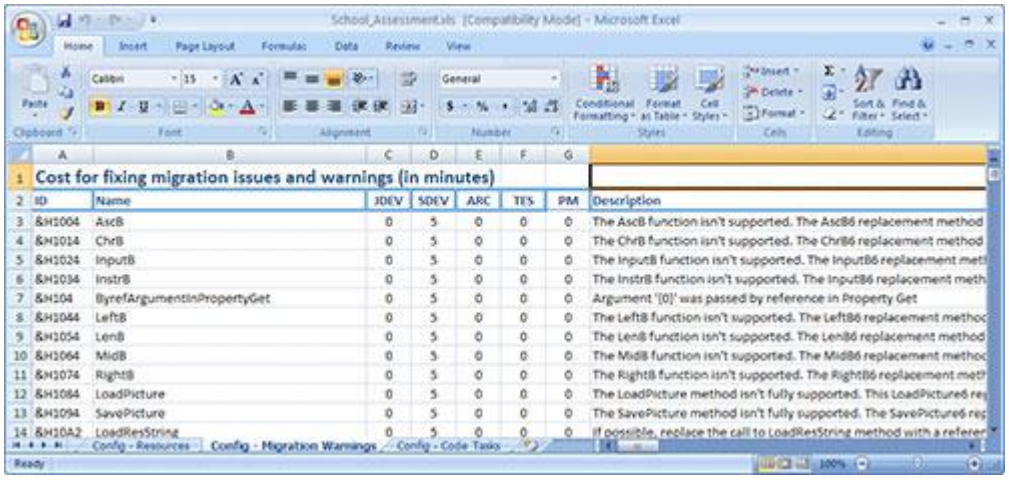
Developer, and Senior Tester, but you can change their name and cost as you see fit. (You can also add more roles, if you are an expert Microsoft Excel user.)

「Config – Resources」ワークシートは5つの異なった開発者の役割を定義し、それぞれに1時間あたりの費用を割り当てます。その所定の役割とは、プロジェクトマネージャー、設計者、中級レベル開発者、上級レベル開発者、上級レベルテスターです。それらの名前やコストは変更することができます。(Microsoft Excel エキスパートユーザであれば他の役割を追加することもできます。)



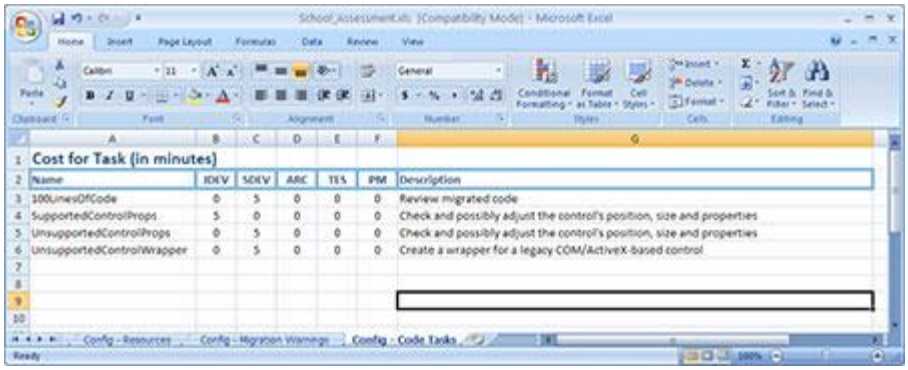
The "Config – Migration Warnings" worksheet is where you configure how many minutes it takes for each developer role to fix a given issue or warning message. Each message is identified by an obscure hexadecimal ID, but you can read its description in the rightmost column.

「Config – Migration Warnings」ワークシートはそれぞれの開発者役割が与えられた問題や警告メッセージを修正するのにどれくらいの時間(分)がかかるかを設定します。それぞれのメッセージは 16 進数の表記による ID で識別されますが、右端の列の説明を読むことができます。



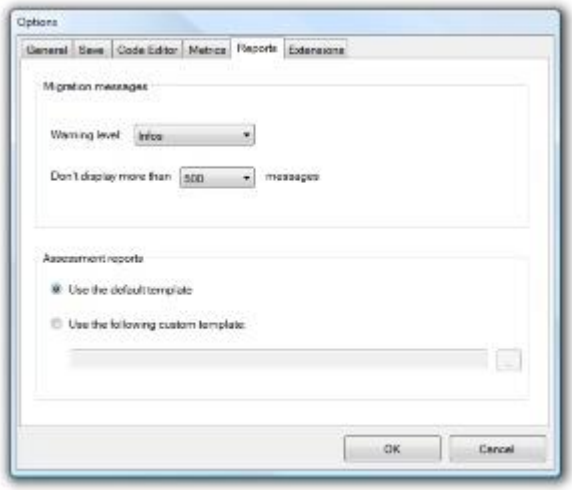
Finally, in the "Config – Code Tasks" you configure the duration (in minutes) for tasks that aren't related to specific migration error and warning messages – for example, the time required to check 100 lines of code to ensure that they have been migrated correctly.

最後の「Config – Code Tasks」ワークシートは仕様的な移行の問題や、警告メッセージに関連しないタスクにかかる時間(分)を設定します。たとえば、正しく移行されたのを保証するために 100 行のコードをチェックするのに要求される時間。



After configuring a report according to your preferences, you should save it to a different Microsoft Excel file, so that you can reuse it as a template for subsequent assessments – either for the same or a different VB6 project. You select which template should be used in the Assessment Report tab of the Tools–Options dialog box.

好みに応じたレポートの設定の後で、異なった Microsoft Excel ファイルを保存してください。同じ Project や別の VB6Project の次回の評価のためのテンプレートとして再利用することができます。「Tools メニュー Options」のダイアログの Reports タブでどのテンプレートをデフォルトで使うかを指定することができます。



2.8 Customizing the code window／コード Window のカスタマイズ

You can customize the colors used by the VB6 and VB.NET code windows from inside the Code Editor tab in the Tool–Options dialog box.

「Tools –> Options」のダイアログの Code Editor タブで VB6 と VB.NET のコード Window で使われる色をカスタマイズすることができます。

3. Converting Language Elements／言語要素を変換する

3. Converting Language Elements／言語要素の変換

This section illustrates how VB Migration Partner converts VB6 language elements and how you can apply pragmas to generate better VB.NET code.

このセクションでは VB Migration Partner がどのように VB6 言語要素を変換するのか、より良い VB.NET コードを生成するのにどのようにプラグマを適用するのかを説明します。

3.1 Array bounds／配列の範囲

VB Migration Partner can adopt five different strategies when converting arrays with non-zero LBound. Developers can enforce a specific strategy by means of the ArrayBounds pragma, as in:

VB Migration Partner はノンゼロ LBound 配列を変換する際に 5 つの異なった手法を採用することができます。開発者は ArrayBound プラグマを使用し、以下のように特定の手法によって変換することができます。

```
' all arrays in scope (class or method) must have LBound = 0
範囲（クラスかメソッド）の全ての配列は LBound=0 がなければなりません
```

```
'## ArrayBounds ForceZero
' ...except the arr variable, which is declared as a VB6Array object
VB6Array オブジェクト変数で宣言された arr 変数を除く
'## arr.ArrayBounds VB6Array
```

Please notice that there is a minor but important limitation in how you can apply this pragma to an array variable: if the array isn't explicitly declared by means of a Dim statement and is only implicitly declared by means of a ReDim statement, pragma at the variable scope are ignored. An example is in order:

配列変数にこのプラグマをどのように適用できるのか小さいですが、重要な制限があることに注意してください。もし配列が明示的に Dim ステートメントで宣言されていない場合、そして、ただ暗黙的に ReDim ステートメントで宣言されているのであれば、プラグマは変数のスコープで無視されます。以下に順番に例を示します。

```
Private Sub Test()
    '## ArrayBounds ForceZero '## arr.ArrayBounds VB6Array
    ReDim arr(1 To 10) As String
    Redim arr2(1 to 10) As Long
End Sub
```

Both arrays are implicitly declared by means of a ReDim statement and lack of an explicit Dim keyword. The abovementioned rules states that the second pragma (scoped at the variable level) is ignored, therefore both arrays will be affected by the first pragma and will be forced to have a zero lower index:

両方の配列は暗黙的に ReDim ステートメントで宣言されています。明示的な Dim キーワードが不足しています。上記のルールでは2番目のプラグマ(変数レベルのスコープ)は無視されます。したがって、両方の配列は最初のプラグマによる影響を受け、ゼロインデックスを持つように定義されます。

```
Private Sub Test()
    Dim arr() As String     ' Implicitly declared array 暗黙的に配列を宣言
    Dim arr2() As Integer   ' Implicitly declared array 暗黙的に配列を宣言
    ReDim arr(10)
    ReDim arr2(10)
End Sub
```

(This limitation is common to all pragmas that apply to array variables, not just the ArrayBounds pragma.)

(この制限は ArrayBounds プラグマだけではなく、配列変数に適用されるすべてのプラグマに共通の制限です。)

Unchanged

The array is emitted as-is, and generates a compilation error in VB.NET if it has a nonzero lower bound. This is the default setting thus you rarely need to use an ArrayBounds pragma to enforce this mode (unless you want to override a pragma with broader scope).

もし配列がそのままの状態で、ノンゼロ範囲であるならば、.NET ではコンパイルエラーとなります。これはデフォルトの設定ですので、このモードを実施するために ArrayBounds プラグマを使用することはほとんどありません。(あえてスコープの範囲を無効にすることをしない場合に限ります)

ForceZero

When this option is selected, the array's lower bound is changed to zero and the upper bound isn't modified. This strategy is fine when the VB6 code processes the array using a loop such as this:

このオプションが選択されている場合、配列の最下限値がゼロに変更されます、上限値は変更されません。この方法は、VB6 コードが次のようなループを使う配列を処理する場合には特に有効です。

```
For i = 1 To UBound(arr)
    ...
Next
```

Shift

VB Migration Partner decreases (or increases) both the lower and the upper bounds by the same value, in such a way the LBound becomes zero. For example, consider the following VB6 fragment
VB Migration Partner は同じ値で上限値と下限値を減少させます(または増加させます)。LBound がゼロになるような手法です。例えば次のような VB6 の断片をご参照下さい。

```
'## arr.ArrayBounds Shift
Dim arr(LoIndex To HiIndex) As String
```

is translated as follows:

上記のコードは次のように変換されます。

```
Dim arr(0 To HiIndex - LoIndex) As String
```

This approach is recommended when it is essential that the number of elements in the array doesn't change after the migration, and is the right choice when the VB6 code processes the array using a loop such as this:

このアプローチは配列の要素の数が移行後に変化しないことが不可欠であるときに推奨されます。そして VB6 コードが次のようなループを使う配列を処理する場合には特に有効です。

```
For i = LBound(arr) To UBound(arr)
    ...
Next
```

Also, this is often the best strategy for arrays defined inside UDTs, if the UDT is often passed to a Windows API method (in which case it's essential that their size doesn't change).

またこれは、ユーザー定義型配列においてもとてもよい手法です。ユーザー定義型配列がWindowsAPIメソッドによく引き渡される場合は。(そのケースにおいてはそれらのサイズが変わらないことは不可欠です)

VB6Array

If the array has a nonzero lower bound, VB Migration Partner replaces the array with an instance of the VB6Array(Of T) generic class. For example, the following VB6 statements:

配列の下限値がゼロでない場合は、VB Migration Partner は配列を一般的なクラスである VB6Array(of T)インスタンスに置き換えます。例として次の VB6 ステートメントを参照下さい。※(of T)は(of Type 型)の意

```
'## ArrayBounds VB6Array
Dim arr(1 To 10) As String
Dim arr2(1 To 10, -5 To 5) As Integer
Dim arr3(0 To 10) As Long
```

are translated as follows:

変換すると次のようになります。

```
Dim arr As New VB6Array(Of String)(1, 10)
Dim arr2 As New VB6Array(Of Short)(1, 10, -5, 5)
Dim arr3(0 To 10) As Integer
```

Instances of the VB6Array class behave much like regular arrays; they support indexes, assignments between arrays, and For Each loops:

VB6Array クラスのインスタンスは標準の配列と同様に動作します。インデックス、配列間の値渡し、For Each ループもサポートします。

```
arr(1) = "abcde"
For Each v In arr2
    sum = sum + v
Next
```

Interestingly, when traversing a multi-dimensional array in a For Each loop, elements are visited in a column-wise manner (as in VB6), rather than in row-wise manner (as in VB.NET), thus no bugs are introduced if the processing order is significant.

興味深い話ですが、For・・・Each ループ内の多次元配列を横断するとき、要素は VB.NET のルールの行方向に移動するよりも、VB6 のルールと同様列方向に移動します。処理順序が明示的であるならば、バグが導入されることはありません。

In order to support VB6Array objects – and for other reasons as well, such support for Variants – VB Migration Partner translates the LBound and UBound methods to the LBound6 and UBound6 methods, respectively. Likewise,

the Erase6, Redim6, and RedimPreserve6 methods are used to clear or resize arrays implemented as VB6Array objects. (These methods are defined in the language support library CodeArchitects.VBLibrary.dll.)

VB6Array オブジェクトをサポートするために、他の機能と同様に様々なものをサポートします。VBMigrationPartner は LBound、UBound メソッドを LBound6、UBound6 へ、それぞれ同様に、Erase6、Redim6、RedimPreserve6 メソッドへ変換します。そしてそれぞれのメソッドは、VB6Array オブジェクトとして実装された配列をクリアまたはリサイズされるように使用されます。(これらのメソッドは言語サポートライブラリの CodeArchitects.VBLibrary.dll に定義されています)

VB Migration Partner fully honors the Option Base directive:

VB Migration Partner は Option Base ステートメントも完全にサポートします。

```
Option Base 1
...
'## ArrayBounds VB6Array
Dim arr(10) As String
numEls = UBound(arr)
```

which is translated as:

上記を変換すると次のようになります。

```
Dim arr As New VB6Array(Of String)(1, 10)
numEls = UBound6(arr)
```

Unfortunately, a syntactical limitation of VB.NET prevents from using a VB6Array object to hold an array of UDTs (i.e. Structure blocks). More precisely, if a VB6Array contains structures, you can read a member of a structure stored in the VB6Array but you can't assign any member. For example, consider the following VB.NET code:

残念ながら、VB.NET の構文の制限は、ユーザー定義型の配列(すなわち Structure ブロックのことです)を持つ VB6Array オブジェクトの使用は制限されます。もっと正確にいえば、VB6Array が構造体を含んでいる場合は、VB6Array に格納されている構造体のメンバを読み込むことができます。しかし、他のメンバを割り当てることはできません。次の VB.NET コードをご覧下さい。

```
Structure MyUDT
    Public ID As Integer
End Structure
...
Sub Main()
    Dim arr As New VB6Array(Of MyUDT)(1, 10)
    Dim value As Integer = arr(1).ID
    ' reading a member is OK
    メンバを読むことはできます
    ' assigning a member causes the following compilation error
```

```
        メンバの割り当てはコンパイルエラーになります
        ' Expression is a value and therefore cannot be the target of an assignment.
        式は値で、割り当ての対象にはなりません

        arr(1).ID = value
    End Sub
```

Therefore, in general you should avoid using the VB6Array option to convert an array of structures. However, this is just a rule of thumb and there can be exceptions to it. For example, if your code assigns whole structures to array elements (as opposed to individual structure members) and then reads their individual members, then storing structures in a VB6Array object is fine.

そのため、基本的に構造体の配列を変換する場合、VB6Array のオプションを使用することを避けてください。しかしながら、これはただの経験値からです。そして例外もあります。たとえば、プログラムコードが全体の構造体を配列の要素に割り当てているのであれば(個々の構造体と対照的に)、個々のメンバを読み込むことはでき、そして VB6Array オブジェクトに構造体を割り当てることもできます。

ForceVB6Array

This option is similar to the previous one, except it applies to *all* arrays in the pragma's scope, regardless of whether the array has a non-zero LBound. This option is useful when the array is declared and created in two different steps – in this case the parser can't decide which strategy to use by looking at the declaration alone – or when the developer knows that the array is going to be passed to a method that exposes parameters of VB6Array type. For example, consider this VB6 fragment:

このオプションは前のものと同様です。ただし、配列がノンゼロ LBound を持っているかどうかに関わらず、プラグマのスコープの全ての配列に適用されます。このオプションは配列が宣言され、そして2つの異なる手順で作られている場合に便利です。ひとつは、パーサーが単独で宣言しているのを見て、どの方針で使用すればいいか決められない場合。または開発者が配列が VB6Array タイプのパラメータを明確にメソッドに渡されるということを知っている場合です。例として次の VB6 コードを見てください。

```
    '## ArrayBounds VB6Array
    Dim arr() As String

    Sub Test()
        ReDim arr(1 To 10) As String
    End Sub
```

Remember that the VB6Array strategy applies only to arrays that have a nonzero lower index. However, when VB Migration Partner parses the *arr* variable it can't decide whether it has a nonzero lower index, therefore it ignores the pragma and renders the variable as a standard array (thus causing a compilation error). This is the correct way to handle such a case:

VB6Array の方針がノンゼロの最下位インデックスを持っている配列だけに適用されるということを覚えておいてください。しかしながら、VB Migration Partner が arr 変数を分析する際に、それがノンゼロ最下位インデックスを持っているかどうか判断できません。そのため、プラグマを無視し、標準の配列として解釈します（そのためコンパイルエラーになります）。以下はそのようなケースを扱う正しい方法です。

```
'## ArrayBounds ForceVB6Array
Dim arr() As String

Sub Test()
    ReDim arr(1 To 10) As String
End Sub
```

which is rendered as:

これを適用すると、次のようになります。

```
Private arr As VB6Array(Of String)

Public Sub Test()
    Redim6(arr, 1, 10)
End Sub
```

Unlike other ArrayBounds options, you can apply the ForceVB6Array strategy to methods' parameters and return values, either with a pragma inside the method with no explicit scope or with a pragma outside the method but that is scoped opportunely:

他の ArrayBound オプションとは異なり、ForceVB6Array の方針をメソッドのパラメータと戻り値に適用できます。明確な範囲のないメソッド内にプラグマを、またはメソッドの外側にプラグマを適用できます。しかし、それはスコープされた機会でもあります。

```
Function GetValues(arr() As String) As Integer()
    '## ArrayBounds ForceVB6Array
    Dim res() as Integer
    ...
    GetValues = res
End Function

'## InitArray.ArrayBounds ForceVB6Array
Function InitArray() As Integer()
    ...
End Function
```

which is translated as follows:

これを適用すると、次のようになります。

```
Function GetValues(arr As VB6Array(Of String)) As VB6Array(Of Short)
    Dim res As New VB6Array(Of Short)
    ...
    Return res
End Function

Function InitArray() As VB6Array(Of Short)
    ...
End Function
```

When dealing with arrays having nonzero lower bound, another pragma can be quite useful. Consider the following VB6 code:

ノンゼロ最下位配列を持っている配列に対処する際には他のプラグマはとても便利です。次の VB6 コードをご覧下さい。

```
Dim primes(1 To 10) As Long
primes(1) = 1: primes(2) = 2: primes(3) = 3: primes(4) = 5: primes(5) = 7
primes(6) = 11: primes(7) = 13: primes(8) = 17: primes(9) = 19: primes(10) =
23
```

You can use an ArrayBounds pragma to force a zero lower bound or to shift both bounds toward zero, but you need a separate ShiftIndexes pragma to account for the indexes used in the last two lines:

ゼロ下限値に制限させるか、またはゼロに向けて両方の値をシフトするように ArrayBound プラグマを使用することができます。しかし、最後の2行で使用しているインデックスのために別々の ShiftIndexes プラグマを使用する必要があります。

```
'## primes.ArrayBounds Shift
'## primes.ShiftIndexes false, 1
Dim primes(1 To 10) As Long
primes(1) = 1: primes(2) = 2: primes(3) = 3: primes(4) = 5: primes(5) = 7
primes(6) = 11: primes(7) = 13: primes(8) = 17: primes(9) = 19: primes(10) =
23
```

this is the result of the migration to VB.NET:

以下が VB.NET に移行された結果です。

```
Dim primes(9) As Integer
primes(0) = 1: primes(1) = 2: primes(2) = 3: primes(3) = 5: primes(4) = 7
primes(5) = 11: primes(6) = 13: primes(7) = 17: primes(8) = 19: primes(9) = 23
```

The first argument of the ShiftIndexes is False if the delta value specified in the second argument must be applied only to constant indexes, True if the delta value must be applied even when the index is a variable or an expression. Using True or False makes a difference when the array is referenced from inside a loop. Consider this example:

もし2番目の引数で指定されたデルタ値が定数のインデックスだけに適応しなければいけないのであれば、False に、インデックスが変数もしくは式である場合に適応しなければいけないのであれば、True に、ShiftIndexes の最初の引数を定義します。True や False を使用することは配列がループ内から参照される際に、効果が得られます。以下に例を示します。

```
'## powers.ArrayBounds Shift
'## Fibonacci.ArrayBounds Shift
'## powers.ShiftIndexes true, 1
'## Fibonacci.ShiftIndexes false, 1

Dim powers(1 To 10) As Double
Dim Fibonacci(1 To 10) As Double
Dim n As Integer

powers(1) = 2
For n = 2 To 10
    powers(n) = powers(n - 1) * 2
Next

Fibonacci(1) = 1: Fibonacci(2) = 1
For n = LBounds(Fibonacci) + 2 To Ubound(Fibonacci)
    Fibonacci(n) = Fibonacci(n - 2) + Fibonacci(n - 1)
Next
```

The difference is in how the loop bounds are specified for the two arrays: for the *powers* array the loop bounds are constant values, therefore it is necessary to compensate in the indexes inside the loop; for the *fibonacci*array the loop bounds are specified in terms of LBound and UBound functions, therefore the indexes inside the loop should not be altered. This is the resulting VB.NET code:

相違点は2つの配列がどのように指定されるのかです。powers 配列のループの範囲が定数の値です、したがってループ内のインデックスで補償することが必要です。フィボナッチ配列のループの範囲は LBound と UBound 関数の範囲で指定されます。したがってループ内のインデックスは修正されるべきではありません。次は VB.NET コードの結果例です。

```
Dim powers(9) As Double
Dim Fibonacci(9) As Double
Dim n As Short

powers(0) = 2
```

```
For n = 2 To 10
    powers(n - 1) = powers(n - 1 - 1) * 2
Next

Fibonacci(0) = 1: Fibonacci(1) = 1
For n = LBounds(Fibonacci) + 2 To Ubound(Fibonacci)
    Fibonacci(n) = Fibonacci(n - 2) + Fibonacci(n - 1)
Next
```

Notice that the ShiftIndexes pragma support up to three delta values, thus you can shift indexes also for 2- and 3-dimension arrays, as in this code:

ShiftIndexes プラグマは最大3つのデルタ値をサポートすることに注意してください。従って、2次元、3次元配列にもインデックスをシフトすることができます。次のコードを参照下さい。

```
'## mat.ArrayBounds Shift
'## mat.ShiftIndexes false, 1, -1
Dim mat(1 To 10, -1 To 1) As Double
```

Delta values can be negative, can be variables and expressions.

デルタ値はネガティブになることもでき、変数や式にもなれます。

The first argument of ShiftIndexes can also be a regular expression that specifies more precisely to which expressions the pragma should be applied. For example, consider the following VB6 code:

ShiftIndexes の最初の引数はプラグマがどの表現に適用させるべきかをより正確に指定するために正規表現になります。例として VB6 のサンプルを次に示します。

```
'## arr.ArrayBounds Shift
'## arr.ShiftIndexes "(k|row)", 1, 1, ""
Dim arr(1 To 10, 1 To 20) As Integer
Dim k As Integer, row As Integer, col As Integer

arr(1, 1) = 0
For k = 2 To 10
    arr(k, 1) = arr(k - 1) + 10
Next

For row = 1 to 10
    For col = LBound(arr, 2) + 1 To UBound(arr, 2)
        arr(row, 1) = arr(row, 1) + arr(row, col)
    Next
Next
```

In this case you want to apply the index adjustments only when the index expression is “k” or “row”, hence the regular expression used in the ShiftIndexes pragma. Here’s the result after then conversion to VB.NET:

この場合、インデックス表現が「k」か「row」である場合にのみインデックスの調整を適用します。したがって、正規表現の ShiftIndexes プラグマで使用されます。

```
Dim arr(9, 19) As Short
Dim k As Short, row As Short, col As Short

arr(0, 0) = 0
For k = 2 To 10
    arr(k - 1, 0) = arr(k - 1 - 1, 0) + 10
Next

For row = 1 To 10
    For col = LBound6(arr, 2) + 1 To UBound6(arr, 2)
        arr(row - 1, 0) = arr(row - 1, 0) + arr(row - 1, col)
    Next
Next
```

Notice that numeric indexes are always affected by the ShiftIndexes pragma, but symbolic numeric constants are affected only you specify a suitable regular expression (or True) in the first argument.

数値のインデックスはいつも ShiftIndexes プラグマによって影響されますが、、シンボリックな数値定数は最初の引数にある適当な正規表現（または True）を指定する場合にのみ影響されるということに注意してください。

3.2 Default members／初期メンバ

The way VB Migration Partner deals with default members depends on how and where the member is defined, and how it is referenced.

VB Migration Partner がデフォルトメンバに対応する方法は、メンバがどのように、またどこで定義されているか、そしてどのように参照されているかによります。

Default property definitions／デフォルトプロパティ定義

When converting a the definition of a property that is marked as the default member of its class, VB Migration Partner adds the Default keyword if the property has one or more arguments; if the property has no parameters, an upgrade warning is issued, because .NET doesn’t support default properties with zero parameters. For example, if this property is the default member of its class:

クラスのデフォルトメンバとしてマークされたプロパティの定義を変換する際に、もしプロパティがひとつもしくはそれ以上の引数を持っている場合、VB Migration Partner はデフォルトキーワードを追加します。プロパティがパラメータを持っていない場合は、アップグレード警告が発生します。なぜなら、.NET はゼロパラメータのデフォルトプロパティをサポートしないからです。例として、プロパティがクラスのデフォルトメンバである場合を示します。

```
Public Property Get Text() As String
    Text = "..."
End Property
```

VB Migration Partner converts it as:

VB Migration Partner は次のように変換します。

```
<System.Runtime.InteropServices.DispId(0)> _
Public ReadOnly Property Text() As String
    ' UPGRADE_WARNING (#0154): Default properties with zero arguments aren't supported.
    Get
        Return "..."
    End Get
End Property
```

Notice that the Property block is tagged with a DispID(0) attribute, so that COM clients see the property as the default member.

プロパティブロックは DispID(0)の属性としてつけられるということに注意してください。よって COM クライアントがデフォルトメンバとしてプロパティを見ます。

Default method and field definitions／デフォルトメソッドとフィールドの定義

When converting a default method or field's definition, VB Migration Partner doesn't modify the definition, except for the addition of the DispID attribute. In this case no Default keyword can be used, because this keyword can be applied only to VB.NET properties.

デフォルトのメソッドやフィールドの定義を変換する際に、VB Migration Partner は定義を修正しません。DispID 属性の追加をのぞきますが。このケースにおいてデフォルトキーワードはまったく使えません。このキーワードは VB.NET プロパティにのみ適用できるからです。

References to default members in early-bound mode／Early Bound モードでのデフォルトメンバへの参照

If the VB6 code references a default property, method, or field through a strong-typed variable, the code generator correctly adds the name of the member. The conversion works correctly for regardless of whether the member belongs to a class defined in the current project, in another project in the solution, or in a type library.

VB6 コードがデフォルトプロパティやメソッド、型宣言をされた変数を通したフィールドを参照している場合、コード生成は正確にメンバの名前を追加します。変換は当該プロジェクトにて参照されたクラス、ソリューションのほかのプロジェクト、またはタイプライブラリの中に属したメンバにかかわらず、正確に機能します。

Accessing default members in late-bound mode／Late Bound モードでのデフォルトメンバのアクセス

If the VB6 code references a default property, method, or field through a Variant, Object, or Control variable, by default VB Migration Partner emits a warning. For example, the following VB6 code

VB6 コードがデフォルトプロパティ、メソッド、バリアントを通したフィールド、オブジェクト、またはコントロール変数を参照している場合はデフォルトで VB Migration Partner は警告を放出します。次に示す VB6 コードをご覧下さい。

```
Sub Test(ByVal obj As Object)
    MsgBox obj
    obj = "new value"
End Sub
```

is translated as:

変換されると次のようになります。

```
Sub Test(ByVal obj As Object)
    ' UPGRADE_WARNING (#0354): Unable to read default member of symbol 'obj'.
    ' Consider using the GetDefaultMember6 helper method.
    MsgBox6(obj)
    ' UPGRADE_WARNING (#0364): Unable to assign default member of symbol 'obj'.
    ' Consider using the SetDefaultMember6 helper method.
    obj = "new value"
End Sub
```

The VB.NET code compiles correctly but delivers bogus results at runtime. You can generate better code by means of the DefaultMemberSupport pragma:

VB.NET は正確にコンパイルします。ですが、ランタイムで正しくない結果を返します。DefaultMemberSupport Pragma を使用してより良いコードを生成することができます。

```
Sub Test(ByVal obj As Object)
    '## DefaultMemberSupport
    MsgBox obj
    obj = "new value"
End Sub
```

which delivers this VB.NET code:

結果、このような VB.NET コードになります。

```
Sub Test(ByVal obj As Object)
    MsgBox6(GetDefaultMember6(obj))
    SetDefaultMember6(obj, "new value")
End Sub
```

The GetDefaultMember6 and SetDefaultMember6 methods are defined in the VBMigrationPartner_Support module. These methods discover and resolve the default member reference at runtime and work correctly also if the default member takes one or more arguments. For example, the following VB6 code:

GetDefaultMember6 と SetDefaultMember6 メソッドは VBMigrationPartner_Support モジュールにて定義されています。これらのメソッドはランタイムのデフォルトメンバ参照と正確な動作を見出し、解決します。また、デフォルトメンバはひとつもしくはそれ以上の引数をとります。次の VB6 コードサンプルを参照下さい。

```
Sub Test(ByVal obj As Object)
    '## DefaultMemberSupport
    Dim res As Integer
    x = obj(1)
    obj(1) = res + 1
End Sub
```

translates to:

変換すると次のようになります。

```
Sub Test(ByVal obj As Object)
    Dim res As Short
    res = GetDefaultMember6(obj, 1)
    SetDefaultMember6(obj, 1, res + 1)
End Sub
```

The discovery process is carried out only the first time the GetDefaultMember6 and SetDefaultMember6 process an object of given type, because the result of the discovery is reused by subsequent calls on variables of the same type. All subsequent references are faster and add no noticeable overhead to the late-bound call.

GetDefaultMember6 と SetDefaultMember6 が初期のみに特定の型のオブジェクトを処理する際に、発見プロセスが行われます。なぜなら、発見の結果がその後の同じ型の変数呼び出しによって再利用されるからです。全てのその後の参照はより速く、Late Bound 呼び出しによる顕著な間接費は発生しません。

3.3 GoSub, On GoTo, and On GoSub keywords／GoSub と On GoTo と On GoSub

VB.NET doesn't support GoSub, On Goto, and On Gosub statements. VB Migration Partner, however, is able to correctly convert these VB6 keywords, at the expense of code readability and maintainability. For this reason we *strongly* recommend that you edit the VB6 application to get rid of all the statements based on these keywords.

VB.NET は GoSub、On GoTo、On GoSub ステートメントはサポートしません。しかし VB Migration Partner ではコードの読みやすさとメンテナンス性を犠牲にしますがこれらの VB6 キーワードを正確に変換できます。この理由として、これらのキーワードに基づくすべてのステートメントを取り除くために VB6 アプリケーションを編集することを強く推奨しているからです。

Anyway, you can surely take advantage of VB Migration Partner ability to handle these statements during the early stages of the migration process. Let's start with the following VB6 method:

とにかく、マイグレーション過程の初期段階の間に、これらのステートメントを扱う VB Migration Partner の能力を活かすことができます。次の VB6 メソッドを見てみましょう。

```
Sub Main()

    GoSub First
    GoSub Second
    Exit Sub

First:
    Debug.Print "First"
    GoSub Third
    Return

Second:
    Debug.Print "Second"
    ' flow into the next section
    次のセクションに移動


Third:
    Debug.Print "Third"
    Return

End Sub
```

This is how VB Migration Partner converts the code:

以下は VB Migration Partner がどのように変換するかを示したものです。

```
Public Sub Main()
    Dim _vb6ReturnStack As New System.Collections.Generic.Stack(Of Integer)

    _vb6ReturnStack.Push(1): GoTo First

ReturnLabel_1:
    _vb6ReturnStack.Push(2): GoTo Second
ReturnLabel_2:
    Exit Sub

First:
    Debug.WriteLine("First")
    _vb6ReturnStack.Push(3): GoTo Third
ReturnLabel_3:
    GoTo _vb6ReturnHandler

Second:
    Debug.WriteLine("Second")
    ' flow into the next section
    次のセクションに移動


Third:
    Debug.WriteLine("Third")
    GoTo _vb6ReturnHandler

    Exit Sub

_vb6ReturnHandler:
    Select Case _vb6ReturnStack.Pop()
        Case 1: GoTo ReturnLabel_1
        Case 2: GoTo ReturnLabel_2
        Case 3: GoTo ReturnLabel_3
    End Select
End Sub
```

As you can see, the GoSub keyword is transformed into a GoTo keyword that uses the *_vb6ReturnStack*variable to "remember" where the Return statement must jump to. The *_vb6ReturnStack* variable holds a stack that keeps the ID of the return address, a 32-bit integer from 1 to N, where N is the number of GoSub statements in the current method.

お分かりになるように、GoSub キーワードは GoTo キーワードに置き換えられます。そのキーワードは、リターンステートメントをどこに移動すればよいかを「覚える」為に、_vb6ReturnStack 変数を使います。_vb6ReturnStack 変数は

リターンアドレスの ID を保持するスタックを持ちます。アドレスは 32Bit の整数1から N、N は現在のメソッドにある GoSub ステートメントの番号の場所です。

The Return keyword is transformed into a GoTo keyword that points to the _vb6ReturnHandler section, where the return address is popped off the stack and used to go back to the statement that immediately follows the GoSub.

Return キーワードは GoTo キーワードに置き換えられ_vb6ReturnHandler セクションを指し示します。リターンアドレスはスタックから取り出され、すぐに GoSub に続くステートメントに戻るために使用されます。

Converting a calculated GoSub delivers similar code, except that the GoSub becomes a GoTo pointing to a Select Case block. For example, the following VB6 code:

計算された GoSub を変換すると、同様なコードを提供します。ただし、GoSub が Select Case ブロックを示す GoTo になることを除きます。次の VB6 コードを見てください。

```
Dim x As Integer
x = 2
On x GoSub First, Second, Third
Exit Sub
```

is converted as:

変換されると次のようになります。

```
    Dim x As Short = 2
    _vb6ReturnStack.Push(4): GoTo OngosubTarget_1
ReturnLabel_4:
    ' ... (other portions omitted for brevity)
    省略


OngosubTarget_1:
Select Case x
    Case 1: GoTo First
    Case 2: GoTo Second
    Case 3: GoTo Third
    Case Is <= 0, Is <= 255: GoTo ReturnLabel_4
    Case Else: Err.Raise(5)
End Select
```

On…GoTo statements are converted in a similar way.

On・・・GoTo ステートメントは同様に変換されます。

Important note: We can't emphasize strongly enough that the code that VB Migration Partner delivers should be never left in a production application, because it is unreadable and hardly maintainable. For this reason, all occurrences of GoSub, On GoTo, and On GoSub keywords cause a warning to be emitted in the generated VB.NET. (This warning has been dropped in examples shown in this section.)

重要事項: VB Migration Partner が提供するコードが生産適用に決して残されているべきではないと強調することはできません。なぜなら可読性とメンテナンス性に欠けるからです。この理由としては、GoSub、On GoTo、On GoSub キーワードのすべての存在が、生成された VB.NET で警告を発するからです。(この警告はこのセクションで示したサンプル中で削除されました)

3.4 Fixed-length strings (FLSs)／固定長文字列

A fixed-length strings (FLS) is converted to an instance of the VB6FixedString class. This class exposes a constructor (which takes the string's length) and the Value property (which takes or returns the string's value). For example, the following VB6 code:

固定長文字列は、VB6FixedString クラスのインスタンスに変換されます。このクラスはコンストラクタ(文字列の長さを持つ)と Value プロパティ(文字列の値を取得もしくは戻す)を公開します。次の VB6 コードを見てください。

```
Dim fs As String * STRINGSIZE
fs = "abcde"
```

is converted as follows:

変換されると次のようになります。

```
Dim fs As New VB6FixedString(STRINGSIZE)
fs.Value = "abcde"
```

The Value property returns the actual internal buffer, an important detail which ensures that VB6FixedString instances work well when they are passed to Windows API methods that store a result in a ByVal string argument. Thanks to this approach, calls that pass FLS arguments to Declare methods work correctly after the migration to VB.NET.

Value プロパティは実際の内部バッファを戻し、重要な詳細情報として、VB6FixedString のインスタンスが ByVal ストリング引数の結果を保管した WindowsAPI メソッドに渡された際に、うまく機能するということを保証します。この手法のおかげで、Declare メソッドが固定長文字列の引数が渡される呼び出しが、VB.NET に移行された後も正しく動作しています。

Arrays of FLSs require a special treatment and are migrated differently. Consider the following VB6 code:

固定長文字列の配列は特別な処理を要求し異なる移行が行われます。次の VB6 コードを見てください。

```
Dim arr(10) As String * 256
arr(0).Value = "abcde"
```

becomes:

変換されると次のようになります。

```
Dim arr() As VB6FixedString_256 = CreateArray6(Of VB6FixedString_256)(0, 10)
arr(0).Value = "abcde"
```

where VB6FixedString_256 a special class in the VisualBasic6.Support.vb module:

VB6FixedString_256 は VisualBasic6.Support.vb モジュールの特別なクラスです。

```
<StructLayout(LayoutKind.Sequential)> _
Public Class VB6FixedString_256
    Private Const SIZE As Integer = 256
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=SIZE)> _
    Private Buffer As String = VB6FixedString.GetEmptyBuffer(SIZE)

    Public Property Value() As String
        Get
            Return VB6FixedString.Truncate(Buffer, SIZE,
ControlChars.NullChar)
        End Get
        Set(ByVal value As String)
            Buffer = VB6FixedString.Truncate(value, SIZE)
        End Set
    End Property
End Class
```

A distinct VB6FixedString_NNN class is generated for each distinct size that appears in FLS declarations inside the current project.

異なった VB6FixedString_NNN のクラスは当該 Project の内部にある固定長文字列宣言に現れるそれぞれの異なったサイズのために生成されます。

As you see above, the FLS array is initialized by means of a call to the CreateArray6 method. This method ensures that all the elements in the array are correctly instantiated, so that no NullReference exception is thrown when accessing any element.

上記のように、固定長文字列の配列は CreateArray6 メソッドによって、初期化されます。このメソッドは配列のすべての要素を正確にインスタンス化されます。NullReference 例外はどんな要素にアクセスする際にもスローされません。

If the array has a nonzero lower index, you can use the ArrayBounds pragma to maintain full compatibility with VB6:

もし配列がノンゼロ最下位インデックスを持っている場合は、VB6 との完全な互換性を維持するのに ArrayBounds プラグマを使用できます。

```
'## arr.ArrayBounds ForceVB6Array
Dim arr(1 to 10) As String * 256
```

which is translated to:

変換されると次のようになります。

```
Dim arr As New VB6ArrayNew(Of VB6FixedString_256)(1, 10)
```

The VB6ArrayNew(Of T) generic class differs from the VB6Array(Of T) class in that it automatically creates an instance of the T type for each element of the array. Using a plain VB6Array(Of T) type would throw a NullReference exception when accessing any array element.

VB6ArrayNew(Of T)というジェネリッククラスは配列のそれぞれの要素のために T 型のインスタンスを自動的に作成するということが VB6Array(Of T)のクラスと異なっています。単純な VB6Array(Of T)の型を使用するとどんな配列の要素にアクセスする際にも NullReference 例外をスローするでしょう。

Finally, notice that you can force a scalar (not array) FLS to be rendered as a VB6FixedString_NNN class by means of a SetStringSize pragma, as in this example:

最後に、SetStringSize プラグマによる VB6FixedString_NNN クラスとしてスカラー(配列ではない)固定長文字列を強制的にレンダリングされることができるということに注意してください。以下が例です。

```
'## s.SetStringSize 128
Dim s As String * 128
```

Such a pragma can be useful if you plan to assign a FLS to an array of FLSs. In practice, however, applying this pragma to scalar FLSs is rarely necessary.

そのようなプラグマは固定長文字列を固定長文字列の配列へアサインする予定があるのであれば、便利です。しかしながら実際にはこのプラグマを固定長文字列のスカラーに適用することはあまり必要ではありません。

3.5 Type…End Type blocks (UDTs)／Type・・・End Type ブロック(ユーザー定義型)

The main problem in converting Type…End Type blocks – a.k.a. User-Defined Types or UDT – to VB.NET is that a .NET structure can't include a default constructor or fields with initializers. This limitation makes it complicated to convert UDTs that include initialized arrays, auto-instancing (As New) object variables, and fixed-length strings, because these elements need to be assigned a value when the UDT is created.

Type・・・End Type ブロックを変換する際の主な問題点は、VB.NET の別名：ユーザー定義型 UDT は.NET の構造体が初期化子があるフィールドまたはデフォルトコンストラクタを含めることができないということです。この制限で初期化された配列、自動インスタンス化オブジェクト変数（As New）、固定長文字列を含むユーザー定義型を変換するのは複雑になります。なぜなら、これらの要素はユーザー定義型が作成される際に値を割り当てる必要があるからです。

VB Migration Partner solves this problem by generating a structure with a constructor that takes one dummy parameter and by ensuring that this constructor is used whenever a new instance of the UDT is created. Consider the following UDT:

VB Migration Partner はひとつのダミーパラメータを持つコンストラクタと共に構造体を生成し、ユーザー定義型の新しいインスタンスが作成される際にはいつもこのコンストラクタが使用されるということを確実にすることによってこの問題を解決します。次のユーザー定義型を参照ください。

```
Type TestUdt
    ' use VB6Array for all arrays with nonzero LBound.
    すべての配列にノンゼロ LBound と共に VB6Array を使用してください。
    '## ArrayBounds VB6Array
    a As Integer
    b As New Widget
    c() As Long
    d(10) As Double
    e(1 To 10) As Currency
    f As String * 10
    g(10) As String * 10
    h(1 To 10) As String * 10
End Type
```

This is how it is translated to VB.NET:

以下は VB.NET にどのように変換されたかの結果です。

```
Structure TestUdt
    Public a As Short
    Public b As Object
    Public c() As Integer
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=11)> _
    Public d() As Double
    Public e As VB6Array(Of Decimal)
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=10)> _
    Public f As VB6FixedString
    <MarshalAs(UnmanagedType.ByValArray, SizeConst:=11)> _
    Public g() As VB6FixedString_10
    Public h As VB6ArrayNew(Of VB6FixedString_10)
```

```
        Public Sub New(ByVal dummyArg As Boolean)
            InitializeUDT()
        End Sub

        Public Sub InitializeUDT()
            b = New Object
            ReDim d(10)
            e = New VB6Array(Of Decimal)(1, 10)
            f = New VB6FixedString(10)
            g = CreateArray6(Of VB6FixedString_10)(0, 10)
            h = New VB6ArrayNew(Of VB6FixedString_10)(1, 10)
        End Sub
    End Structure
```

Note: *Previous example uses the ArrayBounds VB6Array pragma only to prove that VB6Array objects are initialized correctly; in most cases, the most appropriate setting for this pragma inside UDTs is Shift, because this setting ensures that the size of UDTs doesn't change during the migration.*

注記:先述のサンプルは VB6Array オブジェクトが正確に初期化されるということを分かるためにのみ ArrayBounds の VB6Array プラグマを使用します。ほとんどのケースにおいては、ユーザー定義型の中のこのプラグマのための最も適した設定は Shift です。なぜならこの設定はユーザー定義型のサイズが移行作業の間に変わらないということを保証するためです。

Notice that the constructor takes an argument only because it is illegal to define a parameterless constructor in a Structure, but the argument itself is never used. Such a constructor is generated only if the UDT contains one or more members that require initialization, as in previous listing.

構造体の中でパラメータを必要としないコンストラクタを定義することは認められていないので、コンストラクタは引数をとりますが、引数自体が決して使用されないということに注意してください。前に記述したリストのように、初期化を必要とするひとつもしくはそれ以上のメンバを含むユーザー定義型の場合にのみそのようなコンストラクタは生成されます。

The key advantage of having this additional constructor is that it is possible to declare and initialize a UDT in a single operation. For example, the following VB6 statement:

この追加コンストラクタの主要な利点はひとつの操作でユーザー定義型を宣言し、そして初期化するということが可能であるということです。例として次にあげる VB6 ステートメントを参照ください。

```
    Dim udt As TestUdt
```

is translated to:

変換されると次のようになります。

```
Dim udt As New TestUdt(True)
```

VB Migration Partner supports nested UDTs, too. For example, the following VB6 definition:

VB Migration Partner はネストされたユーザー定義型もサポートします。例として、次の VB6 定義を参照ください。

```
Type TestUdt2
    ID As Integer
    Data As TestUdt
End Type
```

is converted to:

変換されると次のようになります。

```
Friend Structure TestUdt2
    Public ID As Short
    Public Data As TestUdt

    Public Sub New(ByVal dummyArg As Boolean)
        InitializeUDT()
    End Sub

    Public Sub InitializeUDT()
        Data = New TestUdt(True)
    End Sub
End Structure
```

A special case occurs when migrating a function or a property that returns a UDT. In this case, the return value is automatically initialized at the top of the code block, as this example demonstrates:

ユーザー定義型を戻す Function またはプロパティを移行する時に特別なケースが生じます。戻り値がコードブロックの上位に自動的に初期化されるケースです。この例を示します。

```
Function GetUDT() As TestUdt
    GetUDT.InitializeUDT()
    ...
End Function
```

Arrays of UDTs are migrated correctly, even if the UDT requires initialization. In such cases, in fact, the array is initialized by means of the CreateArray6 method, which ensures that the InitializeUDT method be called for each element in the array:

ユーザー定義型の配列は正確に移行されます。たとえ、ユーザー定義型が初期化を必要とするとしても。そのようなケースにて、実際には CreateArray6 メソッドにて配列は初期化されます。配列のそれぞれの要素に InitializeUDT メソッドが呼ばれるということを保証します。

```
Dim arr() As TestUdt = CreateArray6(Of TestUdt)(0, 10)
```

In some cases, a FLS defined inside a UDT must be rendered as a standard string rather than a VB6FixedString object. This replacement is necessary, for example, when the UDT is passed to an external method defined by a Declare statement, because the external method expects a standard string.

いくつかのケースにおいては、ユーザー定義型の内部の固定長文字列は VB6FixedString オブジェクトよりもむしろより標準的な文字列としてレンダリングされなければなりません。例として、ユーザー定義型が Declare ステートメントにより定義された外部メソッドに渡されるとき、この置き換えは必要です。なぜなら、外部メソッドが標準的な文字列と予想するからです。

You can force VB Migration Partner to migrate a FLS as a standard string by means of the UseSystemString pragma. A FLS affected by this pragma is rendered as a private regular System.String field which is wrapped by a public property which ensures that values being assigned are always correctly truncated or extended. For example, consider the following VB6 code:

UseSystemString プラグマによって標準的な文字列としての固定長文字列を VB Migration Partner で強制的に移行することができます。このプラグマによって影響を受けた固定長文字列はプライベートなコードとファンクションによってレンダリングされます。文字列フィールドは共通のプロパティによってラッピングされ、割り当てられている値がいつも正確に省略されたり、拡張されたりすることを保証します。例として次の VB6 コードを参照ください。

```
Public Type CDInfo
    '##Title.UseSystemString
    Title As String * 30
    Artist As String * 30
End Type
```

Even though the two items are declared in the same way, the UseSystemString pragma changes the way the Title item is rendered:

二つのアイテムは同じ手法で宣言されているにも関わらず、UseSystemString プラグマは Title アイテムがレンダリングされるように変更します。

```
Friend Structure CDInfo
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=30)> _
    Private m_Title As String
    <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=30)> _
    Public Artist As VB6FixedString

    Public Sub New(ByVal dummyArg As Boolean)
```

```
            InitializeUDT()
        End Sub

        Public Sub InitializeUDT()
            m_Title = VB6FixedString.GetEmptyBuffer(30)
            Artist = New VB6FixedString(30)
        End Sub

        Public Property Title() As String
            Get
                Return VB6FixedString.Truncate(m_Title, 30,
ControlChars.NullChar)
            End Get
            Set(ByVal value As String)
                m_Title = VB6FixedString.Truncate(value, 30)
            End Set
        End Property
    End Structure
```

The UseSystemString pragma can take a boolean value, where True is the default value assumed if you omit the argument. For example, in the following UDT all items are rendered as regular strings except the Year argument:

UseSystemString プラグマはブール値をとることができ、True は引数を省略するのであれば想定されたデフォルト値です。以下のユーザー定義型のすべてのアイテムが Year 引数を省略する通常の文字列としてレンダリングされたサンプルをご参照ください。

```
    Public Type MP3Tag
        '## UseSystemString
        Title As String * 30
        Artist As String * 30
        Album As String * 30
        '## Year.UseSystemString False
        Year As String * 4
    End Type
```

3.6 Auto-instancing variables／自動インスタンス変数

By default, a declaration of an auto-instancing variable is migrated to VB.NET verbatim. For example, the following statement is translated “as-is”:

デフォルトでは、自動インスタンス変数の宣言は一語一語正確に VB.NET に移行されます。例として次の“as □is”変換サンプルステートメントをご覧ください。

```
Dim obj As New Widget
```

In most cases, this behavior is correct, even though the VB6 and VB.NET semantics are different. More precisely, a VB6 auto-instancing variable supports lazy instantiation and can't be tested against the Nothing value, because the very reference to the variable recreates the instance if necessary.

ほとんどのケースにおいて、VB6 と VB.NET の動作が異なっているとしても、この変換動作は正確に行われます。もっと正確に言うと、VB6 自動インスタンス変数は遅延インスタンス化をサポートし、値がないものに対してテストすることができません。なぜなら、参照する変数は必要に応じてインスタンスを再作成するからです。

VB Migration Partner can generate code that preserves the VB6 semantics, if required. This behavior can be achieved by means of the AutoNew pragma, which can be applied at the project, class, method, and variable level.

VB Migration Partner は必要であれば VB6 の動作を保持するコードを生成することができます。この動作は AutoNew プラグマによって成し遂げられます。それは Project、class、メソッド、そして変数レベルに対し適用することができます。

The actual effect of this pragma on local variables is different from the effect on class-level fields:

このプラグマのローカル変数への実際の効果はクラスレベルのフィールドの効果とは異なります。

```
Function GetValue() As Integer
    '## obj.AutoNew True
    Dim obj As New Widget
    ' ...
    obj.Value = 1234
    ' ...
    GetValue = obj.Value
End Function
```

An auto-instancing local variable that is under the scope of an AutoNew pragma is declared without the "New" keyword; instead, all its occurrences in code are automatically wrapped by the special AutoNew6 method:

AutoNew プラグマのスコープである自動インスタンスローカル変数は「New」キーワードを付けずに定義されます。その代わり、コードのそのすべてのオカレンスは特別な AutoNew6 プラグマによって自動的にラッピングされます。

```
Function GetValue() As Short
    Dim obj As Widget
    ' ...
    AutoNew6(obj).Value = 1234
    ' ...
    GetValue = ByVal6(obj)
End Function
```

The AutoNew6 method ensures that the variable abides by the “As New” semantics: a new Widget is instantiated (and assigned to the *obj* variable) when the method is called the first time and it is automatically recreated if the variable is set to Nothing.

AutoNew6 メソッドは変数が「As New」動作によって守られることを保証します。メソッドが最初に呼ばれ、そして変数が Nothing とセットされるのであれば、それは自動的に再作成されます。その際に、新しい Widget がインスタンス化されます。（そして obj 変数に割り当てられます）

A class-level field under the scope of an AutoNew pragma is rendered as a property, whose getter block ensures that the lazy instantiation semantics is honored. For example, if *obj* is a class-level auto-instancing field, VB Migration Partner converts as follows:

AutoNew プラグマのスコープであるクラスレベルのフィールドはプロパティとしてレンダリングされます。そのプロパティのゲッターブロックは遅延インスタンス化動作が称えられるということを保証します。もし obj がクラスレベルの自動インスタンスフィールドである場合、VB Migration Partner が次のように変換する例をご覧ください。

```
Public Property obj() As Widget
    Get
        If obj_InnerField Is Nothing Then obj_InnerField = New Widget ()
        Return obj_InnerField
    End Get
    Set(ByVal value As Widget)
        obj_InnerField = value
    End Set
End Property
Private obj_InnerField As Widget
```

VB6 also supports arrays of auto-instancing elements, and VB Migration Partner fully supports them. If either an appropriate ArrayBounds or AutoNew pragma are in effect for such an array, VB Migration Partner renders it as an instance of the VB6ArrayNew(Of T) type. For example, the following VB6 code:

VB6 は自動インスタンス化要素の配列もサポートしています。そして VB Migration Partner は完全にそれらをサポートします。そのような配列に適切な ArrayBounds または AutoNew プラグマによって効果が得られるならば、VB Migration Partner は VB6ArrayNew（Of T）型のインスタンスとしてそれをレンダリングします。次の VB6 コードを見てください。

```
'## arr.AutoNew
'## arr.ArrayBounds ForceVB6Array
Dim arr(10) As New TestClass()
```

is translated as

変換されると次のようになります。

```
Dim arr() As New VB6ArrayNew(Of TestClass)(0, 10)
```

The VB6ArrayNew(Of T) generic type behaves exactly as VB6Array(Of T), except the former automatically ensures that all its elements are instantiated before they are accessed.

ジェネリックタイプの VB6ArrayNew(Of T)は VB6Array(Of T)として正確に動作します。前者を除き自動的にそれらがアクセスされる前にそのすべての要素がインスタンスを作成されるということを保証します。

3.7 Declare statements／宣言文

VB Migration Partner is able to automatically solve most of the issues related to converting VB6 Declare statements to VB.NET. More specifically, in addition to data type conversion (e.g. Integer to Short, Long to Integer), the code generator adopts the following techniques:

VB Migration Partner は VB6 宣言ステートメントを VB.NET に変換するのに関係があるほとんどの問題を自動的に解決することができます。より明確にいうと、データ型の変換(Integer→Short、Long→Integer など)に加えて、コード生成は次のようなテクニックを採用します。

“As Any” parameters／「As Any」パラメータ

If the Declare statement includes an “As Any” parameter, VB Migration Partner takes note of the type of values passed to it and the passing mechanism used (ByRef or ByVal), and then generates one or more overloads for the Declare statement. An example of a Windows API method that requires this treatment is SendMessage, which can take an integer or a string in its last argument:

宣言ステートメントが「As Any」パラメータを含んでいる場合は、VB Migration Partner はそれに渡された値のタイプと使用される一時的なメカニズム(ByRef または ByVal)を注意します。そして、宣言ステートメントのためにひとつもしくはそれ以上のオーバーロードを発生させます。この処理を要求する WindowsAPI メソッドの例は SendMessage です。その最後の引数に Integer か String をとることができます。

```
Private Declare Function SendMessage Lib "user32.dll" _
    Alias "SendMessageA" (ByVal hWnd As Long, _
    ByVal wMsg As Long, ByVal wParam As Long, _
    lParam As As Any) As Long

Sub SetText()
    ' here we pass a string
    ここで文字列を渡します

    SendMessage Text1.hWnd, WM_SETTEXT, 0, ByVal "new text"
End Sub

Sub CopyToClipboard()
```

```
    ' here we pass a 32-bit integer
    ここで 32bit の数字を渡します

    SendMessage Text1.hWnd, WM_COPY, 0, ByVal 0
End Sub
```

This is the VB.NET code that VB Migration Partner generates. As you see, the As Any argument is gone and two overloads for the SendMessage method have been created:

これは VB Migration Partner が生成した VB.NET コードです。見てわかるように、As Any 引数はなくなり、SendMessage メソッドのために二つのオーバーロードを作成しています。

```
Private Declare Function SendMessage Lib "user32.dll" _
    Alias "SendMessageA" (ByVal hWnd As Integer, _
    ByVal wMsg As Integer, ByVal wParam As Integer, _
    ByVal lParam As String) As Integer

Private Declare Function SendMessage Lib "user32.dll" _
    Alias "SendMessageA" (ByVal hWnd As Integer, _
    ByVal wMsg As Integer, ByVal wParam As Integer, _
    ByVal lParam As Integer) As Integer
```

AddressOf keyword and callback parameters／AddressOf キーワードと callback パラメータ

If client code uses the AddressOf keyword when passing a value to a 32-bit parameter, VB Migration Partner assumes that the parameter takes a callback address and overloads the Declare to take a delegate type. For example, consider the following VB6 code inside the ApiMethods BAS module:

32bit のパラメータに値を渡す際に AddressOf キーワードをクライアントコードが使用しているのであれば、VB Migration Partner はパラメータが callback アドレスをとり、デリゲートタイプの宣言をオーバーロードすることを担います。例として API メソッドの BAS モジュール内の VB6 コードをご覧ください。

```
Declare Function EnumWindows Lib "user32" _
    (ByVal lpEnumFunc As Long, ByVal lParam As Long) As Long

Sub TestEnumWindows()
    EnumWindows AddressOf EnumWindows_CBK, 0
End Sub

' The callback routine
callback ルーチン

Function EnumWindows_CBK(ByVal hWnd As Long, _
    ByVal lParam As Long) As Long
```

```
        ' Store the window handle and return 1 to continue enumeration
        列挙を続けるための戻り値１と Window ハンドルを格納します

        ' ...
        EnumWindows_CBK = 1
    End Function
```

This is how VB Migration Partner converts the code to VB.NET:

VB Migration Partner がどのように VB.NET コードに変換するかを示します。

```
    ' List of Public delegates used for callback methods
    callback メソッドに使用される Public デリゲードのリスト

    Public Delegate Function EnumWindows_CBK(ByVal hWnd As Integer, ByVal lParam
As Integer) As Integer

    Friend Module Module1
        Declare Function EnumWindows Lib "user32" (ByVal lpEnumFunc As Integer, _
            ByVal lParam As Integer) As Integer
        Declare Function EnumWindows Lib "user32" (ByVal lpEnumFunc As
EnumWindows_CBK, _
            ByVal lParam As Integer) As Integer

        Public Sub TestEnumWindows()
            EnumWindows(AddressOf EnumWindows_CBK, 0)
        End Sub

        ' The callback routine
        callback ルーチン

        Function EnumWindows_CBK(ByVal hWnd As Integer, ByVal lParam As Integer)
As Integer
            ' Store the window handle and return 1 to continue enumeration
            列挙を続けるための戻り値１と Window ハンドルを格納します

            ' ...
            Return 1
        End Function
    End Module
```

Notice that only the Declare needs to be overloaded: the code that use the Declare doesn't require any special treatment.

宣言だけがオーバーロードされることが必要であるということを注意してください。特別な処理を必要としない宣言を使うコードです。

Windows API methods that can be replaced by calls to .NET methods／.NET メソッドへの呼び出しによって置き換えられる WindowsAPI メソッド

VB Migration Partner is aware that calls to some specific Windows API methods can be safely replaced by calls to static methods defined in the .NET Framework, as is the case of Beep (which maps to Console.Beep), Sleep (System.Threading.Thread.Sleep), and a few others. When a call to such a Windows API method is found, it is automatically replaced by the corresponding call to the .NET Framework.

VB Migration Partner はいくつかの特定の WindowsAPI メソッドの呼び出しを.NET Framework で定義された Static メソッドを呼ぶことによって安全に置き換えられることができるということを知っておいてください。それらのケースとして、Beep（Console.Beep にマップ）、Sleep（System.Threading.Thread.Sleep）、その他にもいくつかあります。そのような WindowsAPI の呼び出しが見つけられた際には、自動的に.NET Framework の呼び出しに対応したものによって置き換えられます。

Windows API methods that have a recommended .NET counterpart／推奨される.NET 対応版を持つ WindowsAPI メソッド

VB Migration Partner comes with a database of about 300 Windows API methods, where each method is associated with the recommended replacement for .NET. If the parser finds a Declare in this group, a warning is emitted, as in this example:

VB Migration Partner はおよそ 300 個の WindowsAPI メソッドのデータベースを搭載しています。そこではそれぞれのメソッドが推奨される.NET の対象 API に関連付けられています。パーサーがこのグループの宣言を見つけた場合は警告を出力します。次の例をご覧ください。

```
    ' UPGRADE_INFO (#0241): You can replace calls to the GetSystemDirectory'
unmanaged method
    ' with the following .NET member(s): System.Environment.SystemDirectory
    Private Declare Function GetSystemDirectory Lib "kernel32.dll" _
        Alias "GetSystemDirectoryA" (ByVal lpBuffer As String, _
        ByVal nSize As Integer) As Integer
```

3.8 Variant and Control variables／バリアントとコントロール変数

By default, Variant variables are converted to Object variables. This default behavior can be changed by means of the ChangeType pragma, which changes the type of all Variant members (within the pragma's scope) into something else. More specifically, developers can decide that Variant variables are rendered using the special VB6Variant type, as in this code:

デフォルトではバリアント変数はオブジェクト変数に変換されます。このデフォルト機能は ChangeType プラグマによって変更することができます。それは、すべてのバリアントメンバ（プラグマのスコープ外）の型を他の何かに変更します。もっと正確に言うと、開発者はバリアント変数が特別な VB6Variant タイプを使ってレンダリングされるように決めることができます。次のコードを見てください。

```
'## ChangeType Variant, VB6Variant
Dim v As Variant
Dim arr() As Variant
```

which is translated to:

変換されると次のようになります。

```
Dim v As VB6Variant
Dim arr() As VB6Variant
```

The VB6Variant type (defined in the language support library) mimics the behavior of the VB6 Variant type as closely as possible, for example by providing support for the special Null and Empty values.

VB6Variant タイプ（Language Support Library に定義）は可能な限り厳密に VB6 の Variant タイプの機能を踏襲します。たとえば、特別な NULL や Empty 値のサポートを提供していることなどです。

VB6Variant values can be tested by means of the IsEmpty6 and IsNull6 methods, and are recognized by the VarType6 method. Optional parameters of type Variant can be tested with the IsMissing6 function, similarly to what VB6 apps can do.

VB6Variant 値は IsEmpty6、IsNull6 メソッドによりテストすることができます。それらは VarType6 メソッドとして認識されています。タイプ Variant のオプションパラメータは同様に VB6 のアプリケーションでもできたように、IsMissing6 機能でテストすることができます。

The VB6Variant class provides a limited support for null propagation in math and string expressions. This ability is achieved by overloading all math and strings operators. The degree of support offered is enough complete for most common cases, but there might be cases when the result differs from VB6.

VB6Variant のクラスは計算式と文字列式での Null 伝播の限定的なサポートを提供します。この能力はすべての計算と文字列操作をオーバーロードすることによって実現されています。提供されたサポートの度合いは、ほとんどの一般的なケースにおいて十分完全の状態です。しかし、VB6 の結果と異なる場合があるかもしれません。

By default VB Migration Partner translates variables and parameters of type Controls to Object variables and parameters. We opted for this approach because the VB6 Control is actually an IDispatch object and inherently requires late binding, as in this example:

デフォルトでは VB Migration Partner は変数とタイプコントロールのパラメータをオブジェクト変数とパラメータに変換します。なぜなら VB6 コントロールは実際には IDispatch オブジェクトであり、本質的に遅延バインディングが必要なので、この手法を選択しました。次にサンプルを示します。

```
' Make all the textboxes on form read-only
フォーム ReadOnly のすべてのテキストボックスを作ります

Dim ctrl As Control
For Each ctrl In Me.Controls
    If TypeOf ctrl Is TextBox Then ctrl.Locked = True
Next
```

If the *ctrl* variable were rendered as a System.Windows.Forms.Control object, the code wouldn't compile because the Control class doesn't expose a Locked property. By contrast, VB Migration Partner renders the variable as an Object variable and produces VB.NET code that compiles and executes correctly:

Ctrl 変数が System.Windows.Forms.Control オブジェクトとしてレンダリングされていた場合、コントロールクラスは Locked プロパティがないので、コンパイルできません。それに反して、VB Migration Partner は変数をオブジェクト変数としてレンダリングし、コンパイルと、正確に実行する VB.NET コードを実現します。

```
' Make all the textboxes on form read-only
フォーム ReadOnly のすべてのテキストボックスを作ります

Dim ctrl As Object
For Each ctrl In Me.Controls6
    If TypeOf ctrl Is VB6TextBox Then ctrl.Locked = True
Next
```

In other circumstances, however, changing the default behavior might deliver more efficient code. For example, consider this VB6 code:

その他の事情では、デフォルト動作を変えることで、より効率的なコードを提供することになるかも知れません。次の VB6 コードを見てください。

```
'## ctrl.SetType Control
Dim ctrl As Control
For Each ctrl In Me.Controls
    If TypeOf ctrl Is TextBox Or TypeOf ctrl Is ComboBox Then
        ctrl.Text = ""
    End If
Next
```

In this case, you can leverage the fact that the System.Windows.Forms.Control class exposes the Text property, thus you can add a SetType pragma that changes the type for the *ctrl* variable. This is the resulting VB.NET code:

このケースでは、System.Windows.Forms.Control クラスが Text プロパティを持つという事実を利用することができます。よって、ctrl 変数の型を変更する SetType プラグマを追加します。次の VB.NET の結果コードを見てください。

```
Dim ctrl As Control
For Each ctrl In Me.Controls6
    If TypeOf ctrl Is VB6TextBox Or TypeOf ctrl Is VB6ComboBox Then
        ctrl.Text = ""
    End If
Next
```

The *ctrl* variable is now strong-typed and the VB.NET code runs faster.

Ctrl 変数は現在強い型付けになり、VB.NET のコードはより早く動作します。

Please notice the difference between the ChangeType pragma (which affects all the variables and parameters of a given type) and the SetType pragma (which affects only a specific variable or parameter).

ChangeType プラグマ(すべての変数と与えられた型のパラメータに影響します)と SetType プラグマ(特定の変数またはパラメータにのみ影響します)の違いに注意してください。

3.9 Classes and Interfaces／クラスとインターフェース

VB Migration Partner deals with VB6 classes and interfaces in a manner that resembles the way interfaces and coclasses work in COM. More specifically, if a VB6 class named *XYZ* appears in an Implements statement, anywhere in the current solution, then VB Migration Partner generates an Interface named *XYZ* and renames the original class as *XYZ*Class. For example, assume that you have the following IPlugIn class:

VB Migration Partner は COM で動作するインターフェースと coclass の手法に類似している作法で VB6 クラスとインターフェースは取り扱われます。より明確にいうと、XYZ という名前の VB6 クラスが Implements ステートメントや現在の Solution のどこかにあるのであれば、VB Migration Partner は XYZ という名前のインターフェースと XYZClass としてオリジナルのクラスを改名します。IPlugIn クラスの例を参照ください。

```
' the IPlugin class
Sub Execute()
    ' execute the task ...
End Sub

Property Get Name() As String
    ' return Name here ...
End Property
```

Next, assume that the IPlugIn class is referenced by an Implements statement in the SamplePlugIn class, defined elsewhere in the current project or solution:

次に、IPlugIn クラスが現在の Projectまたはソリューションのどこかで定義された SamplePlugIn クラスのImplements ステートメントで参照されているとみなしてください。

```
' inside the SamplePlugIn class
SamplePlugIn class の中で定義

Implements IPlugIn
```

Under these assumptions, this is the code that VB Migration Partner generates:

これらの仮定で、VB Migration Partner は次のようにコードを生成します。

```
Public Class IPlugInClass
    Implements IPlugIn

    Sub Execute() Implements IPlugIn.Execute
        ' execute the task ...
    End Sub

    ReadOnly Property Name() As String Implements IPlugIn.Name
        Get
            ' return Name here ...
        End Get
    End Property
End Class

Public Interface IPlugIn
    Sub Execute()
    ReadOnly Property Name() As String
End Interface
```

This rendering style minimizes the impact on code that references the ISomething class. For example, the following VB6 code:

このレンダリングスタイルは ISomething クラスを参照しているコードの影響を最小限にします。

```
Sub CreatePlugIn(itf As IPlugIn)
    Set itf = New SamplePlugIn
End Sub
```

is converted to a piece of VB.NET code that is virtually identical, except for the Set keyword being dropped:

次は Set キーワードが削除されていることを除けば、ほとんど同一の VB.NET コードに変換されます。

```
Sub CreatePlugIn(ByRef itf As IPlugIn)
```

```
        itf = New SamplePlugIn()
    End Sub
```

References to the IPlugIn type are replaced by references to the IPlugInClass name only when the class name follows the New keyword, as in this VB6 code:

IPlugIn タイプの参照はクラス名が新しいキーワードに従うときのみ、IPlugInClass という名前の参照に置き換えられます。次の VB6 コードをご覧ください。

```
    Dim itf As New IPlugIn
```

which translates to

変換されると、

```
    Dim itf As New IPlugInClass
```

You've seen so far that when a VB6 class appears in an Implements statement, by default VB Migration Partner takes a conservative approach and creates a both a VB.NET class and an interface. This approach ensures that the migrated app works correctly in all cases, including when the VB6 class is actually instantiated. In most real cases, however, a type used in an Implements statement never appears as an operand for the New keyword; therefore generating the class is of no practical use. You can tell VB Migration Partner not to generate the class by means of a ClassRenderMode pragma:

今までに VB6 クラスが Implements ステートメントにある場合、デフォルトで VB Migration Partner が保守的な手法をとり、VB.NET のクラスとインターフェースの両方を作ることを見ていただきました。この手法は VB6 クラスが実際にインスタンス化されているのを含み、移行されたアプリケーションがすべてのケースにて正確に動作することを保証します。しかしながら、実際のケースでは Implements ステートメントで使われる型は New キーワードとしてのオペランドとして現れることはありません。したがってクラスを生成することは実用性のないものです。ClassRenderMode プラグマを使用してクラスを生成しないように VB Migration Partner にて設定することができます。

```
    ' render the current class only as an interface
    このクラスをインターフェースとしてのみレンダリングする
    '## ClassRenderMode Interface
```

The ClassRenderMode pragma can't be applied at the project level and has to be specified for each distinct class.

ClassRenderMode プラグマはプロジェクトレベルでは適用することはできません。それぞれの異なるクラスに対して指定されなければなりません。

3.10 Finalization and disposable classes／終了処理と disposable クラス

All VB6 and COM objects internally manage a *reference counter*: this counter is incremented each time a reference to the object is created and is decremented when the reference is set to Nothing. When the counter reaches zero it's time to fire the Class_Terminate event and destroy the object. This mechanism is known as *deterministic finalization*, because the instant when the object is destroyed can be precisely determined.

すべての VB6 と COM オブジェクトは内部的に参照カウンタを管理します。このカウンタはオブジェクトへの参照が作成されるたびに増加し、オブジェクト参照が Nothing にセットされるごとに減少します。そのカウンタがゼロに達するときに、Class_Terminate イベントが発動し、オブジェクトを破棄します。このメカニズムは確定終了処理として知られています。なぜならオブジェクトが破棄される瞬間が正確に決定できるからです。

.NET objects don't manage a reference counter and objects are physically destroyed only *some time* after all references to them have been set to Nothing, more precisely when a garbage collection is started. One of the biggest challenges in writing a VB6 code converter is the lack of support for deterministic finalization in the .NET Framework.

.NET オブジェクトは参照カウンタを管理せず、オブジェクトはそれらのすべての参照が Nothing でセットされた後の少しの時間で物理的に破棄されます。もっと正確にいえば、ガーベージコレクションが開始されたときです。VB6 コード変換を書くことの最も大きい挑戦のひとつは、.NET Framework の確定終了処理のサポートの不足です。

VB.NET objects that need to execute code when they are destroyed implement the IDisposable interface. Such objects rely on the collaboration from client code, in the sense that the developer who instantiates and uses the object is responsible for disposing of the object – by calling the IDisposable.Dispose method – before setting the object variable to Nothing or letting it go out of scope. In general, any .NET class that defines one or more class-level field of a disposable type should be marked as disposable and implement the IDisposable interface. The code in the Dispose method should orderly dispose of all the objects referenced by the class-level fields.

それらが破棄される際にコードを実行する必要がある VB.NET オブジェクトは IDisposable インターフェースを実装します。そのようなオブジェクトはクライアントコードからのコラボレーションに依存します。開発者はオブジェクト変数を Nothing にセットするか、範囲外にする前に IDsiposable.Dispose メソッドを呼びながらオブジェクトの破棄のためにオブジェクトをインスタンス化し、そして使用することに責任があるということに気がつきます。一般には disposable 型のひとつもしくはそれ以上のクラスレベルフィールドを定義するどんな.NET クラスも disposable としてマークされ、IDisposable インターフェースを実装すべきです。Dispose メソッドのコードはクラスレベルフィールで参照される全てのオブジェクトを正しく処理すべきです。

As just noted, the code that instantiates the class is also responsible for calling the Dispose method as soon as the object isn't necessary any longer, so that referenced disposable objects are disposed as soon as possible. For example, if the class defines and opens one or more database connections (e.g. an SqlConnection object), calling the Dispose method ensures that the connection is closed as quickly as possible. If the call to the Dispose method is omitted, the connection will be closed only later, at the first garbage collection.

注意点としてクラスをインスタンス化するコードはまた、オブジェクトがもう必要でないとすぐに Dispose メソッドを呼ぶ責任があります。よって参照された disposable オブジェクトはできるだけ早く処理されます。たとえば、クラスがひ

とつまたはそれ以上のデータベースコネクション（例 SqlConnection オブジェクト）を定義し、開くなら、Dispose メソッドはコネクションができるだけ素早く閉じられるということ保証します。Dispose メソッドの呼び出しが省略される場合、コネクションは最初のガーベージコレクションで後に閉じられるでしょう。

The .NET Framework also supports finalizable classes. A *finalizable* class is a class that overrides the Finalize method and defines one or more fields that contain Windows handles or other values related to *unmanaged* resources. For example, a class that opens a file by means of the CreateFile Windows API method must be implemented as a finalizable class. The method in the Finalize method is guaranteed to run when the object is being removed from memory during a garbage collection. The code in the Finalize method is expected to close all handles and orderly release all unmanaged resources. Failing to do so would create a resource leak.

.NET Framework は終了処理のクラスをサポートします。Finalizable クラスは終了処理メソッドを無効にし、非管理リソースに関連する他の値と Windows ハンドルを含む1つ以上のフィールドを定義するクラスです。例として、CreateFile Windows API メソッドによるファイルを開くクラスは終了処理のクラスとして実装されなければなりません。終了処理メソッドによるメソッドはガーベージコレクションの中のメモリからオブジェクトが削除されているときに、動作することを保証します。終了処理のメソッドのコードは全てのハンドルが閉じられと全ての非管理リソースが順番に解放することを予想します。そうしないとリソース漏れを引き起こすでしょう。

VB Migration Partner supports both disposable and finalizable classes. However, you might need to insert one or more pragmas to help it to generate the same quality code that an experienced .NET developer would write. Let's start with a VB6 class that handles the Class_Terminate event

VB Migration Partner は終了処理クラスと disposable クラスの両方をサポートします。しかし、ひとつもしくはそれ以上の Pragma を経験豊富な.NET 開発者が書いたものと同じ品質のコードを生成するために挿入する必要があるかもしれません。では VB6 クラスが Class_Terminate イベントをハンドルするのを見てみましょう。

```
    Private fileHandle As Long

    Private Sub Class_Terminate()
        ' CloseHandle is a Windows API method defined elsewhere by a Declare statement
        CloseHandle は Declare ステートメントでどこかに定義された Windows API メソッドです

        CloseHandle fileHandle
    End Sub
```

VB6 classes that include a Class_Terminate are converted to disposable classes that implement the recommended Dispose-Finalize pattern. The generated code ensures that the code inside the original Class_Terminate event runs when either a client invokes the Dispose method or when the garbage collection invokes the Finalize method:

Class_Terminate を含む VB6 クラスが推奨される Dispose－Finalize パターンを実行する disposable クラスに変換されます。生成されたコードは、クライアントが Dispose メソッドを呼び出すか、またはガーベージコレクションが

Finalize メソッドを呼び出す際に、オリジナルの Class_Terminate イベントにおけるコードが動作するのを保証します。

```
Public Class Widget
    Implements IDisposable

    Private fileHandle As Integer

    Private Sub Class_Terminate_VB6()
        CloseHandle(fileHandle)
    End Sub

    Protected Overrides Sub Finalize()
        Dispose(False)
    End Sub

    Public Sub Dispose() Implements System.IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        Class_Terminate_VB6()
    End Sub
End Class
```

If the Terminate event is defined inside a Form or a UserControl class, the Dispose method isn't emitted (because the base class is already disposable); instead, the Form_Terminate or UserControl_Terminate protected method is overridden:

終了処理イベントが Form またはユーザーコントロールクラス内で定義されている場合は、Dispose メソッドは発令されません。(なぜなら基本クラスはすでに処分されているので)、その代り、Form_Terminate または UserControl_Terminate がメソッドが無効されることを保護します。

```
Protected Overrides Sub Form_Terminate_VB6()
    CloseHandle(fileHandle)
End Sub
```

VB Migration Partner can take additional steps to ensure that, if a class uses one or more disposable objects, such objects are correctly disposed of when an instance of the class goes out of scope. In other words, not only does the code generator mark classes with a finalizer as IDisposable classes (as explained above) but it also marks classes using other disposable objects as IDisposable.

VB Migration Partner はそれを保証するための追加ステップをとることができます。もしクラスが1つ以上の Disposable オブジェクトを使用しているのであれば、そのようなオブジェクトはクラスのインスタンスが範囲外にするときに、正確に破棄されます。言い換えればコード生成が IDisposable クラス（上記で説明しています）としての Finalizer をもつクラスをマークするだけではなく、IDisposable としての他の Disposable オブジェクトを使用しているクラスをもマークします。

To explain how this feature works, a few clarifications are in order. As far VB Migration Partner is concerned, a disposable type is one of the following:

この特徴がどのように機能しているか、順番に明確に説明していきます。VB Migration Partner としては Disposable 型としては次に示す通りです。

a. a VB6 class that has a Class_Terminate event (as seen above)
   VB6 クラスは Class_Terminate イベントを持っています。（上記で指摘）
b. a COM type known to be as disposable (e.g. ADODB.Connection)
   COM 型は disposable として知られています。（例　ADODB.Connection）
c. a COM type that is explicitly marked as disposable by means of an AddDisposableType pragma, as in this example: `'## AddDisposableType CALib.DBUtils`
   AddDisposableType プラグマによる disposable として明らかにマークされる COM タイプ。
   例として `'## AddDisposableType CALib.DBUtils`
d. a VB6 class that has one or more class-level fields of a disposable type
   VB6 クラスは disposable 型の 1 つ以上のクラスレベルフィールドを持っています。

VB Migration Partner applies these definition in a recursive way. For example, assuming that class C1 has a field of type ADODB.Connection, class C2 has a field of type C1, and class C3 has a field of class C2, then all the C1, C2, and C3 classes are all marked as IDisposable.

VB Migration Partner は再帰的な方法でこれらの定義を適用します。例えば、クラス C1 にはタイプ ADODB.Connection のフィールドがあって、クラス C2 にタイプ C1 のフィールドがあって、クラス C3 にクラス C2 のフィールドがあると仮定する場合、すべての C1、C2、および C3 のクラスは皆、IDisposable としてマークされます。

If a type is found to be disposable, the exact VB.NET code that VB Migration Partner generates depends on whether it's under the scope of an AutoDispose pragma. This pragma takes an argument that can have the following values:

型が disposable であることが分かっているのであれば、VB Migration Partner が生成する正確な VB.NET コードは AutoDispose プラグマの範囲内にあるかどうかに依存します。このプラグマは次に示す値をもてる引数をとります。

No

Variables of disposable types aren't handled in any special way. (This is the default behavior.)

Disposable 型の変数はどのような方法でもハンドリングされません。（これはデフォルト機能です）

Yes

If X is a variable of a disposable type, the Set X = Nothing statement is converted as follows:

もし X が Disposable 型の変数であれば、Set X=Nothing ステートメントは次のように変換されます。

```
SetNothing6(X)
```

The SetNothing6 method (defined in CodeArchitects.VBLibrary) ensures that the object is cleaned-up correctly. If the object implements IDisposable then SetNothing6 calls its Dispose method. If the object is a COM object, SetNothing6 ensures that the object's RCW is correctly released.

SetNothing6 メソッド（CodeArchitects.VBLibrary に定義）はオブジェクトが正確に除去されることを保証します。オブジェクトが IDisposable を実装している場合、SetNothing6 はその Dispose メソッドを呼びます。オブジェクトが COM オブジェクトの場合は、SetNothing6 はオブジェクトの RCW は正確にリリースされることを保証します。

Force

In addition to converting explicit Set X = Nothing statements for disposable objects, VB Migration Partner ensures that if a VB6 class uses one or more disposable objects, the corresponding VB.NET class implements the IDisposable interface and all the disposable objects are correctly disposed of in the class's Dispose method.

Disposable オブジェクトの Set X=Nothing ステートメントを明確に変換することに加え、VB Migration Partner は VB6 クラスが 1 つ以上の Disposable オブジェクトを使用している場合、対応する VB.NET のクラスは IDisposable インタフェースを実装します、そして、すべての Disposable オブジェクトがクラスの Dispose メソッドで正しく処分されるということを保証します。

Let's see in practice how to use the AutoDispose pragma, starting with the Yes option:

Yes オプションでの AutoDispose Pragma をどのように使用するか、例をご覧ください。

```
'## AutoDispose Yes

Sub Test()
    Dim cn As New ADODB.Connection
    Dim rs As New ADODB.Recordset
    ' opens the connection and the recordset (omitted)
    コネクションをオープンし、レコードセット（省略）
    ' ...

    Set rs = Nothing
    Set cn = Nothing
End Sub
```

The resulting VB.NET code is identical, except for the SetNothing6 method:

SetNothing6 メソッドを除いて、結果としてできた VB.NET コードは同じです。

```
Sub Test()
    Dim cn As New ADODB.Connection
```

```
        Dim rs As New ADODB.Recordset
        ' opens the connection and the recordset (omitted)
        コネクションをオープンし、レコードセット（省略）
        ' ...

        SetNothing6(rs)
        SetNothing6(cn)
    End Sub
```

Let's see now the effects of the Force option, and let's assume that the following VB6 code is contained in the Widget class:

次に Force オプションの効果を見てみましょう。次の VB6 コードが Widget クラスに含まれていると仮定してご覧ください。

```
    '## AutoDispose Force
    '## AddDisposableType CALib.DBUtils
    Dim cn As ADODB.Connection
    Dim utils As CALib.DBUtils
    ...
```

The ADODB.Connection type is known to be disposable, whereas CALib.DBUtils is marked a disposable by the AddDisposableType pragma. (Such a pragma implicitly has a project-level scope.) Because of rule d) above, the Widget class is considered to be disposable, which makes VB Migration Partner generate the following code:

ADODB.Connection 型は Disposable になることが知られていますが、CALib.DBUtils が AddDisposableType プラグマによって、Disposable をマークされました。(そのようなプラグマは暗黙的に ProjectLevel のスコープを持っています) 上記のルール d のために、Widget クラスは Disposable であると考えられています。それで VB Migration Partner は次のコードを作ります。

```
    Public Class Widget
        Implements System.IDisposable

        Dim cn As ADODB.Connection
        Dim utils As CALib.DBUtils
        ...
        Public Sub Dispose() Implements System.IDisposable.Dispose
            SetNothing6(cn)
            SetNothing6(utils)
        End Sub
    End Class
```

If the Widget class has a Class_Terminate event handler, the code in the Dispose method is slightly different:

Widget クラスが Class_Terminate イベントハンドラを持っている場合、Dispose メソッドのコードは多少異なっています。

```
Public Sub Dispose() Implements System.IDisposable.Dispose
    Try
        SetNothing6(cn)
        SetNothing6(utils)
    Finally
        Class_Terminate_VB6()
        GC.SupporessFinalize(Me)
    End Try
End Sub
```

Notice that a class that uses disposable objects doesn't necessarily implement the Finalize method, as per .NET guidelines. Only VB6 classes that have a Class_Terminate event are migrated to VB.NET classes with the Finalize method.

Disposable オブジェクトを使うクラスが.NET のガイドラインに従って、必ず Finalize メソッドを実装する必要がないということに注意してください。Class_Terminate イベントをもつ唯一の VB6 クラスは、Finalize メソッドと共に VB.NET クラスに移行されます。

VB Migration Partner ensures that disposable objects are correctly cleaned-up also when they are assigned to local variables, if a proper AutoDispose pragma is used. For example, consider the following method inside the TestClass class:

適切な AutoDispose プラグマが使われているのであれば、VB Migration Partner は Disposable オブジェクトがローカル変数に割り当てられるときに正確に処分されるということを保証します。

```
'## AutoDispose Force
Sub Execute()
    Dim conn As New ADODB.Connection
    If condition Then
        Dim wid As Widget
        ...
    End If
End Sub
```

In such a case VB Migration Partner moves variables declarations to the top of the method, puts the method's body inside a Try block, and ensures that disposable objects are cleaned-up in the Finally block. Notice that the *wid* variable is cleaned-up as well, because Widget has found it to be disposable:

このようなケースにおいて、VB Migration Partner は変数の宣言をメソッドの上部に移動し、Try ブロックをメソッドのボディに配置します。そして最後に Disposable オブジェクトを Finallyブロックにて処分されることを保証します。Widget が Disposable になると分かったので、wid 変数は処分されるということに注意してください。

```
Sub Execute()
    Dim conn As New ADODB.Connection
    Dim wid As Widget
    Try
        If condition Then
            ...
        End If
    Finally
        SetNothing6(conn)
        SetNothing6(wid)
    End Try
End Sub
```

However, if the method contains one or more On Error statements (which can't coexist with Try blocks) or GoSub statements (which would produce a forbidden GoTo that jumps inside the Try-Catch block), the code generator emits a warning that reminds the developer that a manual fix is needed:

しかし、もしメソッドが 1 つ以上の On Error ステートメント(Try ブロックと共存できない)や GoSub ステートメント(Try-Catch ブロック内に移動する、禁じられている GoTo を提供する)を含んでいる場合、コード生成プログラムは手動による修正が必要になるということを開発者にお知らせする警告を出力します。

```
' UPGRADE_INFO (#0201): An On Error or GoSub statement prevents from generating
' Try-Finally block that clears IDisposable local variables.
```

The approach VB Migration Partner uses to ensure that disposable variables are cleaned-up correctly resolves most of the problems related to undeterministic finalization in .NET. One of the few cases VB Migration Partner can't handle correctly is when a class field or a local variable points to an object that is referenced by fields in another class, as in this case:

VB Migration Partner が disposable 変数を正確に処理するのを保証するのに使用するアプローチは、.NET におけるはっきりしない終了処理に関連したさまざまな問題を解決します。VB Migration Partner が取り扱うことができないわずかなケースのひとつが、クラスフィールドまたはローカル変数が他のクラスのフィールドによって参照されているオブジェクトを示す場合です。次のようなケースです。

```
Sub Execute()
    Dim conn As New ADODB.Connection
    ' GlobalConn is a public variable defined in a BAS module
    GlobalConn は BAS モジュールで定義されているパブリック変数です

    Set GlobalConn = conn
```

```
    ...
End Sub
```

In this specific case, invoking the Dispose method on the conn variable would close the connection referenced by the GlobalConn variable, which in turn may cause the app to malfunction. Developers can avoid this problem by disabling the AutoDispose feature for a given variable or for all the variables in a method:

この具体的なケースで、conn 変数で Dispose メソッドを起動することは GlobalConn 変数により参照されているコネクションをクローズします。そうするとアプリケーションを誤動作することになるかもしれません。開発者は与えられた変数や、メソッド内のすべての変数に AutoDispose 機能を無効にすることによってこの問題を回避することができます。

```
Sub Execute()
    '## conn.AutoDispose No
    Dim conn As New ADODB.Connection
    ...
End Sub
```

3.11 ActiveX Components／ActiveX コンポーネント

VB Migration Partner supports most of the kinds of COM classes that you can create with VB6. This section explains how you can fine-tune the VB.NET code being generated.

VB Migration Partner は VB6 で作ることができる COM クラスのほとんどの種類をサポートします。このセクションでは生成されていく VB.NET コードをどのように微調整していくかを説明します。

ActiveX EXE projectsActiveX EXE プロジェクト

ActiveX EXE projects aren't supported in VB.NET and, by default, VB Migration Partner converts them to standard EXE projects. Developers can change this behavior by means of the ProjectKind pragma:

ActiveX EXE プロジェクトは VB.NET ではサポートされません。そしてデフォルトで VB Migration Partner は標準 EX プロジェクトにそれらを変換します。開発者はこの機能を ProjectKind プラグマで変更することができます。

```
'## ProjectKind dll
```

MultiUse, SingleUse, and PublicNotCreatable classesMultiUse、SingleUse、および PublicNotCreatable クラス

MultiUse and SingleUse classes are converted to public VB.NET classes with a public constructor, so that they can be instantiated from a different assembly. PublicNotCreatable classes are converted to public VB.NET classes whose constructor has Friend scope, so that the class can't be instantiated from outside the current project.

MultiUse、SingleUse クラスは異なったアセンブリから例示できるように Public コンストラクタと共に、PublicVB.NET クラスに変換されます。PublicNotCreatable クラスは友好的な範囲をもつ Public の VB.NET クラスに変換されます。よって、クラスは現在のプロジェクトの外側からインスタンス化することはできません。

Notice that the .NET Framework doesn't support the behavior implied by the SingleUse instancing attribute, therefore SingleUse and MultiUse classes are converted in the same way.

.NET Framework は SingleUse のインスタンス化された属性による暗黙的な動作をサポートしません。よって、SingleUse や MultiUse のクラスは同様に変換されるということに注意してください。

In all three cases, the class is marked with a System.Runtime.InteropServices.ProgID attribute, so that it is visible to COM clients. If the VB6 class was associated to a description, it appears as an XML comment at the top of the VB.NET class:

すべての3つのケースにおいて、クラスは COM クライアントにとって見えるように System.Runtime.InteropServices.ProgID 属性にマークされます。もし VB6 クラスが説明に関連付けられているのであれば、VB.NET クラスの先頭に XML コメントとして現れます。

```
''' <summary>
''' description for the Widget class
''' </summary>
<System.Runtime.InteropServices.ProgID("Project1.Widget")> _
Public Class Widget
    ' A public default constructor
    パブリックデフォルトコンストラクタ

    Public Sub New()
        ' Add initialization code here
        イニシャライズコードをここに記述します

    End Sub

    ' other class members here ...
    その他のクラスのメンバをここに記述します・・・

End Class
```

GlobalMultiUse and GlobalSingleUse classes／GlobalMultiUse と GlobalSingleUse クラス

By default, GlobalMultiUse and GlobalSingleUse classes are translated to standard VB.NET classes. However, when a client accesses a method or property of such classes, VB Migration Partner generates a call to a method of a default instance named *ProjectName_ClassName*_DefInstance, as in:

デフォルトでは GlobalMultiUse と GlobalSingleUse クラスは標準の VB.NET クラスに変換されます。しかし、クライアントがそのようなクラスのメソッドやプロパティにアクセスする場合、VB Migration Partner は Project 名_クラス名_DefInstance という名前のデフォルトインスタンスのメソッドへの呼び出しを生成します。

```
' EvalArea is a method of the Geometry global multiuse class
' defined in an ActiveX DLL project named CALib
```

EvalArea は CALib という名前の ActiveX DLL プロジェクトに定義された GeometryGlobalMultiUse クラスのメソッドです。

```
res = CALib_Geometry_DefInstance.EvalArea(12, 23)
```

All the *_DefInstance variables are defined and instantiated in the VisualBasic6.Support.vb module, in the MyProject folder.

全ての*_DefInstance 変数は MyProject フォルダの VisualBasic6.Support.vb モジュールにて定義もしくはインスタンス化されています。

In most cases, a global class is used as a singleton class and is never instantiated explicitly. In other words, a client typically never uses a global class with the New keyword and uses only the one instance that is instantiated implicitly. If you are sure that all clients abide by this constraint, it is safe to translate the class to a VB.NET module instead of a class, which you do by means of the ClassRenderMode pragma:

ほとんどのケースにおいて Global クラスはシングルトンクラスとして使用され、決して明確にインスタンス化されません。言い換えれば、クライアントが New キーワードをもつ Global クラスを通常使用することはありません。暗黙的にインスタンス化されたたったひとつのインスタンスを使用します。すべてのクライアントがこの制約を守るのを確信しているなら、ClassRenderMode プラグマによってクラスの代わりに VB.NET モジュールにクラスを変換するのは、安全です。

```
' (add inside the Geometry class...) (Geometry クラス内に追加...)
'## ClassRenderMode Module
```

If such a pragma is used, the current class is rendered as a VB.NET Module and no default instance variable is defined in the client project. When a Module is used, methods can be invoked directly, the VB.NET code is more readable, and the method call is slightly faster. Notice that the project name is included in all references, to avoid ambiguities:

そのようなプラグマが使用された場合は、現在のクラスは VB.NET モジュールにレンダリングされ、デフォルトのないインスタンス変数はクライアントのプロジェクトに定義されます。モジュールが使用された際に、メソッドを直接呼び出すことができます。VB.NET コードはより読みやすいコードになり、メソッドの呼び出しも若干早くなります。プロジェクト名があいまいさを避けるためにすべての参照に含まれるということに注意してください。

```
res = CALib.EvalArea(12, 23)
```

Notice that you shouldn't use the ClassRenderMode pragma with global classes that have a Class_Terminate event, because VB Migration Partner automatically renders them as classes that implement the IDisposable interface, and the Implements keyword inside a VB.NET module would cause a compilation error.

Class_Terminate イベントを持っているグローバルなクラスがある ClassRenderMode プラグマを使用するべきでないことに注意してください、VB Migration Partner が IDisposable インタフェースを実行するクラスとして自動的にそれらをレンダリングし、VB.NET モジュールにおける Implements キーワードがコンパイルエラーを引き起こすことになります。

Component initialization／コンポーネントの初期化

If an ActiveX DLL includes a Sub Main method, then the VB6 runtime ensures that this method is invoked before any component in the DLL is instantiated. This mechanism allows VB6 developers to use the Sub Main method to initialize global variables, read configuration files, open database connections, and so forth.

ActiveX DLL が Sub Main メソッドを含んでいる場合は、VB6 のランタイムはこのメソッドが DLL のいくつかのコンポーネントがインスタンス化される前に呼び出されるということを保証します。VB6 開発者はこのメカニズムでグローバル変数を初期化するための Sub Main メソッドを使用できます、構成ファイルの読み込み、データベースコネクションのオープンなど。

This mechanism isn't supported by VB.NET and the .NET Framework in general, therefore VB Migration Partner emits additional code to ensure that the Sub Main is executed exactly once, before any class of the DLL is instantiated.

このメカニズムは一般に.NET Framework と VB.NET ではサポートされていません。したがって、VB Migration Partner は、DLL のいくつかのクラスがインスタンス化される前に、Sub Main が厳密に一度実行されるのを保証するために追加コードを生成します。

```
Public Class Widget
    ' This static constructor ensures that the VB6 support library
    ' be initialized before using current class.
    この静的なコンストラクタは、現在のクラスを使用する前に VB6 サポートライブラリが初期化されるのを保証します。


    Shared Sub New()
        EnsureVB6LibraryInitialization()
        ' Ensure that code in Sub Main be executed before using this class
        Sub Main のコードがこのクラスを使用する前に実行されることを保証します


        EnsureVB6ComponentInitialization()
    End Sub
```

```
    ' other class members here ...
    他のクラスのメンバをここに記述します・・・

End Class
```

The EnsureVB6LibraryInitialization method checks that the language support library is initialized correctly, whereas the EnsureVB6ComponentInitialization method invokes the Sub Main if it hasn't been already executed.

EnsureVB6LibraryInitialization メソッドは、言語サポートライブラリが正しく初期化されるのをチェックしますが、それが既に実行されていないなら、EnsureVB6ComponentInitialization メソッドは Sub Main を呼び出します。

3.12 Persistable classes／持続性(Persistable)クラス

VB Migration Partner fully supports VB6 persistable classes. To illustrate exactly what happens, assume that you have a VB6 class marked as persistable and that handles the InitProperties, ReadProperties, and WriteProperties to implement persistence:

VB Migration Partner は VB6 の持続性クラスを完全にサポートします。何が起きているかを正確に例証するために、持続性としてマークされた VB6 クラスに持続性を実装するために InitProperties、ReadProperties、WriteProperties を扱います。

```
Const ID_DEF As Integer = 0
Const NAME_DEF As String = ""

Public ID As Integer
Public Name As String

' initialize property values
プロパティの値を初期化

Private Sub Class_InitProperties()
    ID = 123
    Name = "widget name"
End Sub

' read property values when the class is deserialized
クラスがデシリアライズされたらプロパティの値を読む

Private Sub Class_ReadProperties(PropBag As PropertyBag)
    ID = PropBag.ReadProperty("ID", ID_DEF)
    Name = PropBag.WriteProperty("Name", NAME_DEF)
```

```
End Sub

' write property values when the object is serialized
オブジェクトがシリアライズされたら、プロパティの値を書き込む

Private Sub Class_WriteProperties(PropBag As PropertyBag)
    PropBag.WriteProperty "ID", ID, ID_DEF
    PropBag.WriteProperty "Name", Name, NAME_DEF
End Sub
```

The resulting VB.NET class is marked with the Serializable attribute and implements the System.Runtime.Serialization.ISerializable interface. The class constructor invokes the Class_InitProperty handler:

結果としてできる VB.NET のクラスは、Serializable 属性でマークされ、System.Runtime.Serialization.ISerializable インタフェースを実装します。クラスのコンストラクタは Class_InitProperty ハンドラを呼び出します。

```
Imports System.Runtime.Serialization

<System.Runtime.InteropServices.ProgID("Project1.Widget")> _
<Serializable()> _
Public Class Widget
    Implements ISerializable

    'A public default constructor
    Public デフォルトコンストラクタ

    Public Sub New()
        Class_InitProperties()
    End Sub
```

Event handlers are converted as standard private methods:

イベントハンドラはスタンダードプライベートメソッドとして変換されます。

```
    Private Const ID_DEF As Short = 0
    Private Const NAME_DEF As String = ""

    Public ID As Short
    Public Name As String = ""

    ' initialize property values
    プロパティの値を初期化

    Private Sub Class_InitProperties()
```

```
        ID = 123
        Name = "widget name"
    End Sub

    ' read property values when the class is deserialized
    クラスがデシリアライズされたらプロパティの値を読む

    Private Sub Class_ReadProperties(ByRef PropBag As VB6PropertyBag)
        ID = PropBag.ReadProperty("ID", ID_DEF)
        Name = PropBag.WriteProperty("Name", NAME_DEF)
    End Sub

    ' write property values when the object is serialized
    オブジェクトがシリアライズされたら、プロパティの値を書き込む

    Private Sub Class_WriteProperties(ByRef PropBag As VB6PropertyBag)
        PropBag.WriteProperty("ID", ID, ID_DEF)
        PropBag.WriteProperty("Name", Name, NAME_DEF)
    End Sub
```

The code in the GetObjectData and the constructor implied by the ISerializable interface invoke the InitProperties, ReadProperties, and WriteProperties handlers:

GetObjectData のコードと ISerializable インタフェースで暗黙的に定義されたコンストラクタは InitProperties、ReadProperties、および WriteProperties ハンドラを呼び出します。

```
    Private Sub GetObjectData(ByVal info As SerializationInfo, _
        ByVal context As StreamingContext) Implements
ISerializable.GetObjectData
        Dim propBag As New VB6PropertyBag
        Class_WriteProperties(propBag)
        info.AddValue("Contents", propBag.Contents)
    End Sub

    Private Sub New(ByVal info As SerializationInfo, ByVal context As
StreamingContext)
        Dim propBag As New VB6PropertyBag
        Class_InitProperties()
        propBag.Contents = info.GetValue("Contents", GetType(Object))
        Class_ReadProperties(propBag)
    End Sub
End Class
```

All references to the VB6's PropertyBag object are replaced by references to VB6PropertyBag, a class with similar interface and behavior defined in the language support library. It is important to bear in mind, however, that binary files created by persisting a VB6 object can't be deserialized into a VB.NET object, and vice versa.

VB6 の PropertyBag オブジェクトのすべての参照は同様のインターフェースと、言語サポートライブラリで定義されている動作をもつクラス VB6PropertyBag への参照に置き換えられます。しかしながら、持続性の VB6 オブジェクトによって作成されたバイナリファイルが VB.NET オブジェクトにデシリアライズすることができないということを覚えておくことが重要であり、その逆も同様です。

3.13 Resources／リソース

VB6 resource files are converted to standard .resx files and can be viewed and modified by means of the My Project designer. More precisely, resources are converted to My.Resources.*prefixNNN*, where *prefix* is "str" for string resources, "bmp" for bitmaps, "cur" for cursors, and "ico" for icons.

VB6 リソースファイルはスタンダード.resx ファイルに変換されます。また My Project デザイナによって表示させたり、修正することができます。より詳細に説明すると、リソースは My.Resources.prefixNNN に変換されます。そこでは、接頭語は、ストリングリソースのための「str」と、ビットマップへの「bmp」と、カーソルのための「cur」と、アイコンのための「ico」です。

VB Migration Partner attempts to convert all occurrences of LoadResString, LoadResPicture, and LoadResData methods into references to My.Resource.*prefixNNN* elements. This is possible, however, only if the arguments passed to these method are constant values or constant expressions, as in the following VB6 example:

VBMP は、LoadResString、LoadResPicture、および LoadResData メソッドのすべてのオカレンスを My.Resource.prefixNNN 要素の参照に変換するのを試みます。しかしながら、これらのメソッドに渡された引数が、コンスタント値かコンスタント式である場合にだけ可能です。以下の VB6 の例を御参照ください。

```
Const RESBASE As Integer = 100
Const STRINGRES As Integer = RESBASE + 1
MsgBox LoadResString(STRINGRES)
Image1.Picture = LoadResPicture(RESBASE + 7, vbResBitmap)
```

which is correctly translated into:

正確に変換されます。

```
Const RESBASE As Short = 100
Const STRINGRES As Short = RESBASE + 1
MsgBox6(My.Resources.str101)
Image1.Picture = My.Resources.bmp107
```

If the first or the second argument isn't a constant, then VB Migration Partner falls back to the LoadResString6, LoadResPicture6, and LoadResData6 support methods. These methods rely on the same ResourceManager instance used by the My.Resources class and therefore return the same resource data. This approach ensures that all .NET localization features can be used on the converted project, including satellite resource-only DLLs.

もし最初か 2 番目の引数がコンスタントではない場合は、VBMP はメソッドをサポートする、LoadResString6、LoadResPicture6、および LoadResData6 に戻ります。これらのメソッドは、My.Resources のクラスによって使用された同じ ResourceManager インスタンスに依存します、したがって、同じリソースデータを返します。このアプローチは、サテライト DLL(リソースだけの)を含む変換されたプロジェクトで、すべての.NET ローカライズ機能を使用できるのを確実にします。

Interestingly, if an icon resource is being assigned to a VB6 icon property that has been translated to a bitmap property under VB.NET, then VB Migration Partner automatically generates the code that manages the conversion, as in this code:

興味深いことに、アイコンリソースが VB.NET 配下でビットマッププロパティに変換された VB6 アイコンプロパティに割り当てられているのであれば、VB Migration Partner は自動的に変換を管理する次のようなコードを生成します。

```
Image1.Picture = My.Resources.ico108.ToBitmap()
```

3.14 Minor language differences／マイナーな言語の違い

VB Migration Partner generates code that accounts also for minor differences between VB6 and VB.NET.

VB Migration Partner は VB6 と VB.NET 間でのマイナーな違いを説明するコードを生成します。

Font objects／Font オブジェクト

VB6's Font and StdFont objects are converted to .NET Font objects. The main difference between these two objects is that the .NET Font object is immutable. Consider the following VB6 code:

VB6 の Font と StdFont オブジェクトは.NET の Font オブジェクトに変換されます。これら二つのオブジェクトの主な違いは、.NET Font オブジェクトは不変であるということです。次の VB6 コードを参照ください。

```
    Dim fnt As StdFont                      ' a StdFont objectStdFont オブジェクト
    Set fnt = Text1.Font
    fnt.Bold = True
    Text2.Font.Name = "Arial"               ' a control's Font property コントロールの Font プロパティ
```

Assignments to font properties are translated to FontChange*Xxxx*6 methods in the language support library:

Font プロパティへの割り当ては言語サポートライブラリの FontChangeXXXX6 メソッドに変換されます。

```
    Dim fnt As Font                         ' a StdFont objectStdFont オブジェクト
    fnt = Text1.Font
    FontChangeBold6(fnt, True)
    FontChangeName6(Text2.Font, "Arial")    ' a control's Font property コントロールの Font プロパティ
```

VB Migration Partner provides support also for the StdFont.Weight property. For example, this VB6 code:

VB Migration Partner は StdFont.Weight プロパティも同様にサポートを提供します。次の VB6 コード例をご覧ください。

```
    Dim x As Integer
    x = Text1.Font.Weight
    Text2.Font.Weight = x * 2
```

translates to:

変換されると

```
    Dim x As Integer
    x = GetFontWeight6(Text1.Font)
    SetFontWeight6(Text2.Font, x * 2)
```

The GetFontWeight6 and SetFontWeight6 helper functions map the Weight property to the Bold property. They are marked as obsolete, so that the developer can easily spot and get rid of them after the migration has completed.

GetFontWeight6 と SetFontWeight6 ヘルパー機能は Weight プロパティを Bold プロパティにマップします。それらはサポート外とマークされていますので、開発者は移行が完了した後で、それらを簡単に見つけ取り除くことができます。

VB Migration Partner emits a warning if the original VB6 program handles the FontChanged event exposed by the StdFont object. In this case no automatic workaround exists and code must be fixed manually.

オリジナルの VB6 プログラムが StdFont オブジェクトによって露呈された FontChanged イベントを処理するなら、VB Migration Partner は警告を出力します。この場合、どのような自動回避策も存在しません、そして、手動でコードを修正しなければなりません。

For Each loop on multi-dimensional arrays／多次元配列における For Each ループ

For Each loops visit multi-dimensional arrays in column-wise order under VB6, and in row-wise order under VB.NET. When such a loop is detected, VB Migration Partner emits the following warning just before the loop:

For Each ループは VB6 では列方向の順で、.NET では行方向の順で多次元配列を巡回します。そのようなループが検出されるとき、VBMP はループののすぐ前で以下の警告を出力します。

```
    ' UPGRADE WARNING (#01C4): Insert a call to TransposeArray6 method if process order
    ' in following For Each loop is significant
    For Each o As Object In myArr
        ...
    Next
```

The TransposeArray6 helper function returns an array whose rows and columns are transposed, so that the For Each loop works exactly as in the VB6 program:

TransposeArray6 ヘルパー機能は行と列を入れ替えた配列を戻します。よって For Each ループは VB6 プログラムと同様に正確に動作します。

```
    For Each o As Object In TransposeArray6(myArr)
        ...
    Next
```

You can add the call to the TransposeArray6 method at each migration cycle, by means of an ReplaceStatement pragma, while using a DisableMessage pragma to suppress the warning:

それぞれの移行サイクルにて、警告メッセージを抑制する DisableMessage プラグマを使用しながら、ReplaceStatement プラグマで TransposeArray6 メソッドの呼び出しを追加することができます。

```
    '## DisableMessage 01C4
    '## ReplaceStatement For Each o As Object In TransposeArray6(myArr)
    For Each o As Object In myArr
        ...
    Next
```

Fields passed to ByRef arguments／ByRef 引数に渡されたフィールド

VB6 fields are internally implemented as properties and can't be modified by a method even if they are passed to a ByRef argument; in the same circumstances, a VB.NET field *can* be modified. For this reason, converting such calls to VB.NET as-is can introduce subtle and hard-to-find bugs. (being ByRef the default passing mechanism in VB6, such bugs can be rather frequent.)

VB6 フィールドは内部でプロパティとして実装されます。それらが、ByRef 引数に渡されたとしてもメソッドによって変更されることはありません。同じ事情で VB.NET フィールドは変更されます。この理由としては、そのままで VB.NET へのそのような呼び出しを変換すると捉えにくく、見つけることが難しいバグを導入することになります。(VB6 のデフォルト引き渡しメカニズムである ByRef になることは、そのようなバグがかなり頻繁になります。)

VB Migration Partner handles this issue by wrapping the field in a call to the ByVal6 helper method, as in this example:

VB Migration Partner は ByVal6 ヘルパーメソッドを呼ぶフィールドをラッピングすることでこの問題を制御することができます。次のサンプルコードをご覧ください。

```
' Value is a field of the object held in the obj variable
値は、obj 変数に格納されていたオブジェクトのフィールドです

MethodWithByrefParam( ByVal6(obj.Value) )
```

ByVal6 is a do-nothing method that simply returns its argument, ensuring that the original VB6 semantics are preserved. The ByVal6 method is marked as obsolete and produces a compilation warning that encourages the developer to double-check the intended behavior and modify either the calling code or the syntax of the called method.

ByVal6 はとにかくその引数を戻すという何もしないメソッドです。オリジナルの VB6 意味論が失われないようにするのを確実にします。ByBal6 メソッドはサポート外とマークされており、コンパイルの警告を発行し、呼ばれたメソッドの構文やコードの呼び出しを修正するように、そして開発者に意図された動作をダブルチェックするように奨励します。

TypeOf keyword／TypeOf キーワード

VB Migration Partner accounts for minor differences in how the TypeOf keyword behaves in VB6 and VB.NET, more precisely:

VB Migration Partner は TypeOf キーワードが VB6 と VB.NET でどのように動作するかという点でのわずかな違いを説明します。より正確には。

a. TypeOf can’t by applied to VB.NET Structures.
   TypeOf は VB.NET 構造体には適用できません。
b. using TypeOf to test a Nothing value raises an error in VB6, but not in VB.NET.
   Nothing 値をテストするのに TypeOf を使用することは VB6 ではエラーになりますが、VB.NET ではなりません。
c. testing against the Object type never fails in VB.NET, except when testing an interface variable.
   インターフェース変数をテストする時以外で、Object タイプに対するテストは VB.NET において決して失敗しません。

Consider the following VB6 code:

次の VB6 コードを見てください。

```
If TypeOf obj Is TestUDT Then
    ' obj is a UDT
     obj はユーザー定義型です

ElseIf TypeOf obj Is Widget Then
    ' obj holds an instance of a specific class
```

```
        obj は固有のクラスのインスタンスを保持します

    ElseIf TypeOf obj Is Object Then
        ' obj holds an instance of any class
        obj はいくつかのクラスのインスタンスを保持します

    ElseIf TypeOf obj Is ITestInterface Then
        ' obj is an interface variable
        obj はインターフェース変数です

    End If
```

To account for these differences, VB Migration Partner performs the test using reflection, except when the second operand is an interface type. This is the converted VB.NET code:

これらの違いを説明するために、VB Migration Partner は2番目のオペランドがインターフェース型であることを除き、反射を使用したテストを実行します。次は変換された VB.NET のコードです。

```
    If obj.GetType() Is GetType(TestUDT) Then
        ' this test works if obj is a Structure
        obj が構造体ならこのテストは動作します

    ElseIf obj.GetType() Is GetType(Widget) Then
        ' this test throws if obj is Nothing (as in VB6)
        obj が Nothing (VB6 での) であれば、このテストはスローします

    ElseIf (Not TypeOf obj Is String AndAlso Not obj.GetType().IsValueType) Then
        ' obj contains an instance of a reference type
        obj は参照型のインスタンスを含みます

    ElseIf TypeOf obj Is ITestInterface Then
        ' no need to modify tests for interfaces
        インターフェースの為のテストを修正する必要はありません

    End If
```

If the variable being tested is of type VB6Variant, the TypeOf operator is translated as a call to the special IsTypeOf6 method:

テストされている変数が VB6Variant タイプであるならば、TypeOf オペレータは特別な IsTypeOf6 メソッドへの呼び出しとして変換されます。

```
    If IsTypeOf6(myVariant, GetType(Widget)) Then
```

Mod operator／Mod オペレータ

The VB6's Mod operator automatically converts its operands to integer values and returns the remainder of integer division:

VB6 の Mod オペレータは自動的に integer 値へのそのオペランドを変換します、そして整数部の残りの部分を返します

```
Dim d As Double, i As Integer
d = 1.8: i = 11
Debug.Print i Mod d     ' Displays 1, because it rounds up 1.8 to 2 1を表示します。なぜなら1.8は2に丸められるからです
```

VB.NET doesn't convert to Integer and returns the remainder of floating-point division if any of the two operands is of type Single or Double. VB Migration Partner ensures that the operator behaves exactly as in VB6 by forcing a conversion to Integer where needed:

VB.NET は Integer に変換しません。2つのオペランドのどちらかが Single または Double 型の場合は、浮動小数点除算の残りを返します。VB Migration Partner はオペレータが必要であるところで Integer に変換を強制することにより VB6 のように正確に動作することを保証します。

```
Debug.WriteLine(i Mod CInt(d))
```

Eqv and Imp operators／Eqv と Imp オペレータ

Both these operators aren't supported by VB.NET. VB Migration Partner translates them by generating the bitwise operation that they perform internally. More precisely, the following VB6 code:

これらのオペレータの両方とも VB.NET ではサポートされません。VB Migration Partner はそれらが内部的に実行されるビット単位のオペレーションを生成することによってそれらを変換します。もっと正確に述べると次のような VB6 コードを見てください。

```
x = y Eqv z
x = y Imp z
```

translates to

変換すると

```
x = (Not y And Not z)
x = (Not y Or z)
```

Strings to Byte array conversions／Strings を Byte 配列に変換

VB Migration Partner correctly converts assignments between string and Byte arrays. The following VB6 code:

VB Migration Partner は String と Byte 配列の間の割り当てを正しく変換します。次の VB6 コードをご覧ください。

```
Dim s As String, b() As Byte
s = "abcde"
b = s
s = b
```

translates to:

変換すると

```
Dim s As String, b() As Byte
s = "abcde"
b = StringToByteArray6(s)
s = ByteArrayToString6(b)
```

where the StringToByteArray6 and ByteArrayToString6 helper methods perform the actual conversion. If necessary, this transformation is performed also when a string or a Byte array is passed to a method's argument or returned by a function or property.

ここでは StringToByteArray6 と ByteArrayToString6 のヘルパーメソッドが実際の変換を実施します。また、必要であれば、String か Byte 配列がメソッドの引数に引き渡されるか、または Function かプロパティによって戻されるとき、この変換は実行されます。

Note: the ByteArrayToString6 method internally uses the UnicodeEncoding.Unicode.GetString method, but adds a dummy null byte if the argument has an odd number of bytes.

注記: ByteArrayToString6 メソッドは内部的に UnicodeEncoding.Unicode.GetString メソッドを使用します。しかし、もし引数が奇数のバイトを持っている場合はダミーの Null バイトを追加します。

Date to Double conversions／Date 型を Double 型に変換

VB Migration Partner correctly converts assignments between Date and Double values. The following VB6 code:

VB Migration Partner は Date 値と Double 値の間の割り当てを正しく変換します。次の VB6 コードをご覧ください。

```
Dim d As Date, v As Double
d = #11/1/2006#
v = d
d = v
```

translates to:

変換すると

```
Dim d As Date, v As Double
d = #11/1/2006#
```

```
v = DateToDouble6(d)
d = DoubleToDate6(v)
```

where the DateToDouble6 and DoubleToDate6 helper methods internally map to Double.ToOADate and Date.FromOADate methods.

ここでは DateToDouble6 と DoubleToDate6 ヘルパーメソッドは内部的に Double.ToOADate と Date.FromOADate メソッドにマップします。

For loops with Date variables／Date 変数における For Loops

For…Next loops that use a Date control variable are allowed in VB6 but not in VB.NET. For example, consider the following VB6 code:

Date コントロール変数を使う For･･･Next loops は VB6 では許可されますが、VB.NET ではされません。次の VB6 コードを見て下さい。

```
Dim dt As Date
For dt = startDate To endDate
    Debug.Print dt
Next
```

This is how VB Migration Partner converts the code to VB.NET:

VB Migration Partner がどのように VB.NET コードに変換するか、次の結果を見てください。

```
Dim dt As Date
For dt_Alias As Double = DateToDouble6(startDate) To DateToDouble6(endDate)
    dt = DoubleToDate6(dt_Alias)
    Debug.WriteLine(dt)
Next
```

3.15 Unsupported features and controls／サポートされていない機能とコントロール

Here's a list of VB6 features that VB Migration Partner doesn't support:

ここに VB Migration Partner でサポートされない VB6 の機能を一覧します。

- ActiveX Documents
  ActiveX ドキュメント
- Property Pages
  プロパティページ

- Web classes
  Web クラス
- DHTML classes
  DHTML クラス
- DataReport designer
  DataReport デザイナー
- OLE and Repeater controls
  OLE と Repeater コントロール
- a few graphic-related properties common to several controls, including DrawMode, ClipControls, Palette, PaletteMode
  DrawMode、ClipControls、Palette、PaletteMode を含む数個のコントロールに共通のいくつかのグラフィック関連のプロパティ
- VarPtr, ObjPtr, StrPtr undocumented keywords (the VarPtr keyword is partially supported, though)
  VarPtr、ObjPtr、StrPtr 言語仕様がないキーワード（VarPtr キーワードは部分的にサポートされます）
- a small number of control features, including:
  少数のコントロール機能、以下を含みます

  - The ability to customize, save, and restore the appearance of the Toolbar control
    Toolbar コントロールの外観をカスタマイズ、保存、復元する機能
  - The MultiSelect property of the TabStrip control
    TabStrip コントロールの MultiSelect プロパティ
  - Vertical orientation for the ProgressBar control
    ProgressBar コントロールの垂直配向
  - The DropHighlight property of TreeView and ListView controls
    TreeView と ListView コントロールの DropHighlight プロパティ

Even if a feature isn't supported, VB Migration Partner always attempts to emit VB.NET that has no compilation errors. For example, ActiveX Documents and Property Pages are converted into VB.NET user controls. Also, the support library exposes do-nothing properties and methods named after the original VB6 unsupported member.

機能がサポートされない場合、VB Migration Partner は常にコンパイルエラーのない VB.NET を生成するのを試みます。例えば、ActiveX ドキュメントとプロパティページは VB.NET のユーザコントロールに変換されます。またサポートライブラリはオリジナルの VB6 でサポートされないメンバにちなんで名づけられた何も動作しないプロパティとメソッドを露呈します。

Unsupported properties and methods／サポート外のプロパティとメソッド

In general, unsupported members in controls are marked as obsolete and return a default reasonable value. For example, the DrawMode property of the Form, PictureBox, UserControl, Line, and Shape classes always return 1-vbBlackness.

一般に、コントロールのサポートされないメンバはサポートされないと位置付けられ、デフォルトの妥当性のある値を戻します。例えば、Form の DrawMode プロパティ、PictureBox、UserControl、Line、Shape クラスは常に vbBlackness の1を戻します。

By default, assigning an unsupported property or invoking an unsupported method is silently ignored. If the property or the method affects the control's appearance you can't modify such a default behavior.

デフォルトでは、サポートされないプロパティを割り当てるか、サポートされないメソッドを呼び出すことは無視されます。プロパティかメソッドがコントロールの外観に影響するのであれば、そのようなデフォルト動作を修正することはできません。

If the property or the method affects the control's behavior (as opposed to appearance), however, you can force the control to throw an error when the property is assigned a value other than its default value or when the method is invoked, by setting the VB6Config.ThrowOnUnsupportedMember property to True:

プロパティまたはメソッドがコントロールの動作に影響を与えるのであれば(外観とは対照的に)、デフォルト値以外の値がプロパティに割り当てられるか、またはメソッドが呼び出されるとき、VB6Config.ThrowOnUnsupportedMember のプロパティを True に設定することによって、コントロールにエラーを強制的に投げることができます。

```
VB6Config.ThrowOnUnsupportedMember = True
```

This setting is global and affects all the controls and classes in control support library.

この設定はグローバル設定で、コントロールサポートライブラリの全てのコントロールとクラスに影響を与えます。

Unsupported Controls／サポート外のコントロール

Occurrences of unsupported controls – including OLE instances and all controls that aren't included in the Visual Basic 6 package – are replaced by a VB6Placeholder control, which is nothing but a rectangular empty control with a red background.

サポート外のコントロールのオカレンス(VisualBasic6 パッケージにない、全てのコントロールと OLE インスタンスを含む)はただの赤の背景色をもつ長方形の空のコントロールである VB6Placeholder コントロールによって置き換えられます。

For each unsupported control, VB Migration Partner generates a form-level Object variable named after the original control:

それぞれのサポート外のコントロールのために、VB Migration Patner はオリジナルコントロールにちなんだ名前の Form レベルオブジェクト変数を生成します。

```
' UPGRADE_ISSUE (#02C8): Controls of type VB.OLE aren't supported. This control
' was replaced by a dummy variable to make the VB.NET compiler happy
Friend OLE1 As Object
```

The variable is never assigned an object reference, therefore any attempt to use it causes a NullReferenceException to be thrown at runtime. This variable has the only purpose of avoiding one compilation error for each statement that references the control.

変数はオブジェクト参照を割り当てることはありません。そのため、それを使用するどんな試みも NullReferenceException をランタイムに投げさせます。この変数には、コントロールを参照する各ステートメントのコンパイルエラーを回避する唯一の効果があります。

3.16 The VB6Config class／VB6 コンフィグクラス

Some facets of the runtime behavior of VB.NET applications converted with VB Migration Partner can be modified by means of the VB6Config class. This class exposes only static members, therefore you never need to instantiate an object of the class. Here is the list of exposed members:

VB6Config クラスによって VB Migration Partner と共に変換された VB.NET アプリケーションのランタイム動作のいくつかの一面を変更できます。このクラスは Static なメンバのみを出力します。よってクラスのオブジェクトをインスタンス化する必要はありません。以下は出力されたメンバのリストです。

VB6Config.ThrowOnUnsupportedMembers

This property specify whether unsupported properties of controls throw an exception when they are assigned an invalid value. For example, converted VB.NET forms and PictureBox controls don't support the DrawMode and ClipControls properties, even though they expose two properties named DrawMode and ClipControls. By default ThrowOnUnsupportedMembers is False, therefore assigning a value other than 1-Blackness to the DrawMode property or the True value to the ClipControls property doesn't raise an error and the assignment is just ignored. However, if you set the ThrowOnUnsupportedMembers property to True, the same action raises a runtime error whose code is 999 and whose message is "Unsupported member":

このプロパティは、無効の値がそれらに割り当てられるとき、コントロールのサポートされないプロパティが例外を投げるかどうかを指示します。例えば、変換された VB.NET の Form と PictureBox コントロールは DrawMode と ClipControls プロパティをサポートしておりませんが、DrawMode と ClipControls という名前の2つのプロパティを出力します。デフォルトでは ThrowOnUnsupportedMembers は False です。したがって、DrawMode プロパティに 1-Blackness 以外の値を割り当てる、または ClipControls プロパティに True 値を割り当てることはエラーを起こしません。割り当てはただ無視されます。しかしながら、ThrowOnUnsupportedMembers を True に設定する場合、同じ動作をすると RunTimeError を引き起こします。その時のエラーコードは"999"でメッセージは"Unsupported member"です。

```
VB6Config.ThrowOnUnsupportedMembers = True
```

The ThrowOnUnsupportedMembers also affects the behavior of unsupported methods, such as RichTextBox.SelPrint or SSTab.ShowFocusRect. Usually, calls to these unsupported methods are simply ignored. However, if ThrowOnUnsupportedMembers is True than the method throws an exception.

ThrowOnUnsupportedMembers はまた RichTextBox.SelPrint や SSTab.ShowFocusRect といったサポートされないメソッドの動作にも影響します。通常では、これらのサポートされないメソッドを呼ぶこと単純に無視されます。しかし ThrowOnUnsupportedMembers を True に設定するとそのメソッドは例外をスローします。

Please notice that there are a few unsupported language keywords – namely VarPtr, StrPtr, and ObjPtr – that always raise an error, regardless of the value of ThrowOnUnsupportedMembers. We implemented this behavior because typically the value of these functions is passed to a Windows API, in which case passing an invalid value might compromise the entire system.

サポートされない言語キーワードがいくつかあることを覚えておいてください。すなわち、VarPtr、StrPtr、ObjPtr です。それらは、ThrowOnUnsupportedMembers の設定値に関係なく常にエラーを引き起こします。この動作を実装しましたが、これらのファンクションの値は一般的に、WindowsAPI に渡されます。その場合、システム全体の動作に影響をきたすことになるかもしれません。

VB6Config.LogFatalErrors

All .NET applications immediately stop when an unhandled error is raised (unless they run inside a debugger). The VB.NET applications converted by VB Migration Partner are no exception, however they have the added ability to display a message box containing the complete stack dump of where the error occurred. This information can be very important when debugging an application that runs fine on the development machine and fails elsewhere. You enable this feature by setting the LogFatalErrors to True:

全ての.NET アプリケーションはハンドルされないエラーが引き起こされた場合、直ちにストップします。(デバッガで起動している場合を除く)VB Migration Partner によって変換された VB.NET アプリケーションは例外エラーはありません。しかし、どこでエラーが発生したか完全なスタックダンプを含んでいるメッセージボックスを表示する追加された機能を持っています。問題なく開発環境マシンで動いて、ほかの場所で失敗するアプリケーションをデバッグする際に、この情報は非常に重要である場合があります。LogFatalErrors を True に設定することによってこの機能を有効にできます。

```
VB6Config.LogFatalErrors = True
```

VB6Config.FocusEventSupport

VB6 and VB.NET controls slightly differ in the exact sequence of events that fire when a control receives or loses the input focus. For example, when you click on a VB6 control, the control raises a MouseDown event followed by a GotFocus event, whereas a VB.NET control fires the GotFocus event first, followed by the MouseDown event.

VB6 と VB.NET ではコントロールがフォーカスの入力を受け取ったり、放したりする際の正確なイベントの順番がわずかに異なります。例えば、VB6 コントロールをクリックすると、コントロールは MouseDown イベントを先に、続いて GotFocus イベントに移ります。VB.NET コントロールでは GotFocus イベントを先に、続いて MouseDown イベントに移ります。

Another difference is the event sequence that is fired when the focus is moved away from the control. VB6 controls fire the Validate event first, then fire the LostFocus event only if validation succeeded. (If validation failed, the LostFocus event is never raised). Conversely, the behavior of standard VB.NET depends on whether the focus

is moved away by means of the keyboard or the mouse. If the keyboard is used, a Validate event is raised, followed by a LostFocus event. If the mouse, the event order is just the opposite. Worse, a GotFocus event is fired if validation fails.

他の異なる点はフォーカスがコントロールから外部に移動された際に、起こるイベントの順番です。VB6 コントロールは最初に Validate イベントを発生させ、Validation が成功した場合にのみ LostFocus イベントを発生させます（もし Validation が失敗した場合は、LostFocus イベントは発生しません）。逆に、スタンダードの VB.NET の動作はキーボードかマウスによるフォーカスの移動によります。キーボードを使用した場合は、Validate イベントが起動され、次に LostFocus イベントに移ります。マウスの場合は、イベント順は反対です。より悪いことに、Validation が失敗しても、GotFocus イベントが発生します。

In most cases, these differences are negligible and don't affect the application's overall behavior. For this reason, therefore, by default converted applications follow the .NET standard behavior. If you experience a problem related to these events, however, you can have VB.NET controls behave exactly like the original VB6 controls, simply by setting the FocusEventSupport to True:

ほとんどのケースにおいて、これらの違いはとるに足らないことで、アプリケーションの全体の動作に影響はありません。その理由としては、デフォルトで変換されたアプリケーションは.NET スタンダード動作に従います。これらのイベントに関連する問題を経験するなら、オリジナルの VB6 コントロールのような正確な動作を VB.NET コントロールに持たせることができます。それは簡単に FocusEventSupport を True に設定するだけです。

```
VB6Config.FocusEventSupport = True
```

VB6Config.ReturnedNullValue

Some VB6 functions return a Null value when the caller passes Null to one of their parameters. This behavior is duplicated under converted VB.NET applications. By default, the .NET equivalent for the Null value is the DBNull.Value, a detail that might cause a problem when the value is passed to a string or used in an expression. For example, consider this VB6 code:

いくつかの VB6 の機能は呼び出し側がパーサに引数のうちのひとつに Null を渡す場合、Null 値が戻されます。この動作は変換された VB.NET のアプリケーションでも再現されます。デフォルトでは Null 値に対する.NET 相当物は DBNull.Value です。詳細に、値が Stringに渡されるか式に使われる場合に問題が起こるかもしれません。例として、次の VB6 コードをご覧ください。

```
' rs is a recordset, txtValue is a TextBox control
rs はレコードセット、txtValue は TextBox コントロール

txtValue.Text = Hex(rs("value").Value)
```

This code fails under VB.NET if the field "value" of the recordset contains Null, because the Hex function would return DBNull.Value and the assignment to the Text property would raise an error.

Recordset のフィールド"値"に Null が含まれているのであれば、VB.NET ではこのコードは失敗します。なぜなら、Hex 関数が DBNull.Value を戻し、Text プロパティへの割り当てがエラーを引き起こすからです。

To work around this issue, you can assign the ReturnNullValue property a value other than DBNull.Value. In most cases, using an empty string solves the problem:

この問題を回避するために、DBNull.Value 以外の値を ReturnNullValue のプロパティに割り当てることができます。ほとんどのケースにおいて、空文字列を使用することで問題を解決できます。

```
VB6Config.ReturnedNullValue = True
```

VB6Config.DragIcon

When a “classic” drag-and-drop operation begins, by default VB6 shows the outline of the control that acts as the source of the drag-and-drop. This behavior isn’t replicated in converted VB.NET apps, which instead display a default image. This default image is the one returned by the VB6Config.DragIcon property, but you can assign this property to change the default image:

クラシックな drag-and-drop 操作を行うとき、デフォルトで、VB6 は drag-and-drop の起点として機能するコントロールのアウトラインを示します。この動作は変換された VB.NET アプリケーションでは模写されておりません。代わりにデフォルトイメージを表示します。このデフォルトイメージは VB6Config.DragIcon プロパティによって戻されるイメージです。デフォルトイメージを変更するためにこのプロパティを割り当てることができます。

```
VB6Config.DragIcon = LoadPicture6("c:¥dragicon.bmp")
```

As in VB6, the default image is used only for controls whose DragIcon is nothing.

VB6 では、デフォルトイメージは DragIcon がないコントロールにおいてのみ使用されます。

VB6Config.DDEAppTitle

By default, a VB6 app that works as DDE server reacts to requests from client controls whose LinkTopic property is formatted as follows:

デフォルトでは、DDE サーバが LinkTopic のプロパティが以下の通りフォーマットされるクライアントコントロールからの要求に反応するとき、VB6 アプリケーションは動作します。

appname|*formlinkopic*

where *appname* is the application title (same as the App.Title property in VB6) and *formlinktopic* is the value of the LinkTopic property of a form whose LinkMode property has been set to 1-vbLinkSource).

appname はアプリケーションのタイトル（VB6 の App.Title と同じ）です。そして formlinkopic は LinkMode プロパティが 1-vbLinkSource でセットされている form の LinkTopic プロパティの値です。

Converted VB.NET forms work much like in the same way, except that the *appname* value is checked against the VB6Config.DDEAppTitle property. The initial value of this property is the same it was in the VB6 project, therefore it most cases the application should work as it used to do under VB6. However, the ability to change the *appname* your DDE server app responds to adds a degree of flexibility to your migrated projects.

変換された VB.NET の form は同じ方法で同じように動作します。appname の値が VB6Config.DDEAppTitle のプロパティに対してチェックされるのを除きます。このプロパティの初期値は VB6 プロジェクトであった時と同様です。よって、ほとんどのケースにおけるアプリケーションが VB6 で動作していたのであれば、機能します。しかし、DDE サーバーアプリケーションの appname を変える能力は移行されたプロジェクトのフレキシビリティの度合いにより追加するために応答します。

4. Advanced Topics／アドバンストトピックス

4. Advanced Topics／アドバンストトピックス

4.1 The VBMigrationPartner_Support module／VB Migration Partner サポートモジュール

While VB Migration Partner can often automatically translate most VB6 constructs, in some cases it is necessary for you to either modify the VB6 code before the conversion or modify the VB.NET code after the conversion process.

VB Migration Partner は多くの場合自動的に VB6 の構造物を変換することができますが、いくつかのケースにおいて変換前に VB6 のコードを修正するか、または変換プロセスの後で、VB.NET のコードを修正する必要があります。

In general, we recommend that you don't alter the original VB6 code in a significant way if the VB6 application is still in production, because relevant edits should be thoroughly tested and validated. If you edit the original VB6 application you should be guaranteed that your edits don't change the application's behavior.

一般的に、VB6 のアプリケーションがまだ稼動中であれば、大きな観点でオリジナルの VB6 コードを修正しないことをお勧めします。なぜなら適切な修正がなされたかどうか徹底的にテストされ、正しい動作が確認されるべきであるからです。もしオリジナルの VB6 アプリケーションを修正したのであれば、修正した箇所がアプリケーションの動作を変えないことを保証されるべきです。

Ideally, you should modify the VB6 code only by means of pragmas. Pragmas are just remarks, therefore they can't affect the original application in any way. For example, you can use the InsertStatement pragma to insert VB.NET statements in the middle of the code to be migrated:

理想的には、Pragma によってのみ VB6 コードを修正すべきです。Pragma はただのコメントです。したがって、それらはどのような方法でもオリジナルの VB6 アプリケーションに影響を及ぼすことはできません。例えば、変換しようとするコードの真ん中に VB.NET のステートメントを挿入するように InsertStatement Pragma を使うことができます。

```
'## InsertStatement  If x < 0 Then Return x
```

The main limitation of the InsertStatement pragma is that it can only insert entire VB.NET statements; it can't, for example, add function calls in the middle of an expression.

InsertStatement Pragma の主要な制限は完全な VB.NET のステートメントのみ挿入することができるということです。例えば、式の真ん中にファンクションコールを追加することはできません。

VB Migration Partner comes with a special VB6 module – stored in the VBMigrationPartner_Support.bas file – that contains many special methods that you can use in your VB6 applications to prepare them for a smooth translation to VB.NET. You need to include this module in the VB6 project being converted, so that you can use these methods; VB Migration Partner never translates this module nor includes it in the migrated VB.NET.

VB Migration Partner は特別な VB6 モジュールを搭載しており、そのモジュールは VBMigrationPartner_Support.bas ファイルに格納されています。VB6 モジュールにはたくさんの特別なメソッドが含まれており、VB.NET へのスムーズな変換を準備できるように VB6 アプリケーションの中で使用することができます。これらのメソッド使用することができるように、変換される VB6 プロジェクトにこのモジュールを含める必要があります。VB Migration Partner はこのモジュールを変換しませんし、変換された VB.NET にもそれを含めません。

All the methods in this support module have names with a trailing "6" character, therefore chances of name clashing with other methods or variables in the original VB6 application are negligible. All these methods are just "do-nothing" or "pass-through" methods and don't alter VB6 behavior. To see how these methods can be useful, consider the following VB6 code fragment:

このサポートモジュールの全てのメソッドは末尾に"6"という文字を付加した名前を持っています。よって、VB6 アプリケーションの中で他のメソッドや変数に名前が衝突することはめったに起こらないことになります。全てのこれらのメソッドはただの"do-nothing"または"pass-through"メソッドで、VB6 の動作を変えることはありません。これらのメソッドがどのように有益なものとなるかを見るために、次の VB6 コード断片を御覧ください。

```
' change fore and back color properties of an object, if possible
可能であれば、オブジェクトの ForeColor と BackColor プロパティをチェンジします

Sub ChangeColor(ByVal obj As Object, ByVal foreColor As Long, ByVal backColor As Long)
    On Error Resume Next
    obj.ForeColor = foreColor
    obj.BackColor = backColor
End Sub
```

The problem with this code is that the target object is accessed through late binding, therefore VB Migration Partner has no way to determine that the ForeColor and BackColor properties take a System.Drawing.Color object and that a conversion from the 32-bit integer is needed.

このコードの問題は遅延バインディングでターゲットのオブジェクトがアクセスされるということです。したがって、VB Migration Partner には、ForeColor と BackColor のプロパティが System.Drawing.Color オブジェクトを使うことや、32 ビットの整数からの変換が必要であることを決定する方法がありません。

One way to solve this problem is by means of the FromOleColor6 method defined in the VBMigrationPartner_Support module. After adding the module to current VB6 project, change the original code as follows:

この問題を解決するひとつの方法としては、VBMigrationPartner_Support モジュールに定義されている FromOleColor6 によって解決することができます。当該 VB6 プロジェクトにそのモジュールを追加した後で、次のようにオリジナルコードを変更してください。

```
' change fore and back color properties of an object, if possible
可能であれば、オブジェクトの ForeColor と BackColor プロパティをチェンジします

Sub ChangeColor(ByVal obj As Object, ByVal foreColor As Long, ByVal backColor As Long)
    On Error Resume Next
    obj.ForeColor = FromOleColor6(foreColor)
    obj.BackColor = FromOleColor6(backColor)
End Sub
```

The FromOleColor6 method is a "pass-through" method: it takes a 32-bit integer and returns it to the caller, without modifying it in any way. This ensures that the VB6 code continues to work as before the edit. However, when the code is translated to VB.NET, the VBMigrationPartner_Support module is discarded and the converted VB.NET code now references the FromOleColor6 method defined in the language support library, which converts a 32-bit value to the corresponding System.Drawing.Color object. The bottom line: the converted VB.NET code works like the original application and no manual fix after the migration is needed.

FromOleColor6 メソッドは"Pass - Through"メソッドです。32 ビットの整数を取り、全く修正せずに Caller に戻します。これはコードを編集する前と同様に VB6 コードが動作を継続するということを保証します。しかし、コードが VB.NET に変換された際に、VBMigrationPartner_Support モジュールは捨てられます。そして変換された VB.NET コードは、次に Language サポートライブラリ内に定義された FromOleColor6 メソッドを参照し、32 ビットの値を対応した System.Drawing.Color オブジェクトに変換します。結論:移行が必要であった後に、手動で修正することなく、変換された VB.NET コードはオリジナルアプリケーションのように動作します。

The VBMigrationPartner_Support module contains several conversion methods. You can use them when VB Migration Partner isn't able to detect the type used in an expression or in an assignment, for example when the value is held in a Variant variable:

VBMigrationPartner_Support モジュールは様々な変換メソッドを含みます。VB Migration Partner が式や代入で使用されている型を検出できない場合にそれらを使用することができます。例えば、値がバリアント変数に保持されている場合です。

FromOleColor6, ToOleColor6:

FromOleColor6 convert a 32-bit integer into a .NET Color value, whereas ToOleColor6 converts a .NET Color value into the corresponding 32-bit integer.

FromOleColor6 は 32 ビットの整数を.NET Color 値に変換する一方で、ToOleColor6 は.NET Color 値を対応する 32 ビットの整数に変換します。

RGB6, QBColor6:

Similar to RGB and QBColor methods, except they return a .NET Color object.

.NET Color オブジェクトを戻す以外は、RGB と QBColor メソッドと同様です。

DateToDouble6, DoubleToDate6:

Explicitly convert a Date value to a Double value, and vice versa.

明確に Date 値を Double 値に変換します、そして、逆もまた同様です。

ByteArrayToString6, StringToByteArray6:

Explicit convert a Byte array to a String value, and vice versa.

明確にバイト配列を String 値に変換します、そして、逆もまた同様です。

Another group of methods force VB Migration Partner to generate the correct code when you access a Font object in late-bound mode. For example, consider this method:

メソッドの別のグループが、Late - Bound モードで Font オブジェクトにアクセスする際、VB Migration Partner は正しいコードを生成します。次のメソッドの例を参照下さい。

```
Sub SetControlFont(ByVal frm As Form)
    Dim ctrl As Control
    For ctrl In frm.Controls
        ctrl.Font.Name = "Arial"
        ctrl.Font.Size = 12
        ctrl.Font.Bold = True
    Next
End Sub
```

Once again, the problem is that *ctrl* is an IDispatch variable: the Font property is accessed in late-bound mode, therefore VB Migration Partner can't determine that it references a System.Drawing.Font object and can't take the appropriate steps to account for the fact that .NET font objects are immutable.

もう一度、問題は、ctrl が IDispatch 変数であるということです。Font プロパティが Late □Bound モードでアクセスされており、したがって、VB Migration Partner は、System.Drawing.Font オブジェクトを参照することを決定できず、.NET フォントオブジェクトが不変である事実を明確にするために正しい措置をとることができません。

The VBMigrationPartner_Support module includes the **FontChangeName6, FontChangeSize6, FontChangeBold6, FontChangeItalic6, FontChangeStrikeout6**, and **FontChangeUnderline6** methods which, as their name suggest, allow you to work around the read-only nature of the corresponding property of the .NET Font object. Here's how you can use these methods to prepare previous code snippet for a smooth translation:

VBMigrationPartner_Support モジュールは **FontChangeName6**、**FontChangeSize6**、**FontChangeBold6**、**FontChangeItalic6**、**FontChangeStrikeout6**、**FontChangeUnderline6** メソッドを含みます。これらの名前が示唆しているように、.NET Font オブジェクトに対応するプロパティで ReadOnly という性質で動作します。ここに、スムーズな変換をするために前述のコード断片を用意して、これらのメソッドの使用方法を記します。

```
Sub SetControlFont(ByVal frm As Form)
    Dim ctrl As Control
    For ctrl In frm.Controls
        ChangeFontName6(ctrl.Font, "Arial")
        ChangeFontSize6(ctrl.Font, 12)
        ChangeFontBold6(ctrl.Font, True)
    Next
End Sub
```

(This is exactly the code that VB Migration Partner would generate if the control variable was accessed in early-bound mode.)

(もしコントロール変数が Early - Bound モードでアクセスされたならば、VB Migration Partner は正しいコードを生成します。)

Read the remarks in the VBMigrationPartner_Support module for a complete list of supported methods, their purpose, and example usages.

VBMigrationPartner_Support モジュールの備考をご覧下さい。そこには、サポートされるメソッドの一覧、それらの目的、およびサンプルの用法などがあります。

4.2 Code analysis features／コード解析機能

VB Migration Partner employs several code analysis techniques to deliver high-quality VB.NET code.

VB Migration Partner は、高品質な VB.NET コードを提供するのにいくつかのコード分析技術を用います。

Implicitly declared local variables／暗黙的に宣言されたローカル変数

VB Migration Partner emits an UPGRADE INFO remark for each local variable that wasn't declared explicitly in the original project. For example, the following VB6 code:

VB Migration Partner はオリジナルプロジェクトで明確に宣言されなかった各ローカル変数のために UPGRADE INFO として所見を出力します。次の VB6 コード例をご覧下さい。

```
Public Sub Test()
    x = 123
```

```
End Sub
```

generates the following VB.NET code

次のような VB.NET コードを生成します。

```
Public Sub Test()
    ' UPGRADE_INFO (#05B1): The 'x' variable wasn't declared explicitly.
    x = 123
End Sub
```

You can force VB Migration Partner to generate an explicit declaration for such local variables by means of the**DeclareImplicitVariables** pragma, therefore this VB6 code:

DeclareImplicitVariables Pragma によってそのようなローカル変数を明確に宣言するように VB Migration Partner に指示することができます。この VB6 コードに適用すると次のようになります。

```
Public Sub Test()
    '## DeclareImplicitVariables
    x = 123
End Sub
```

generates the following VB.NET code

次のような VB.NET コードを生成します。

```
Public Sub Test()
    ' UPGRADE_INFO (#05B1): The 'x' variable wasn't declared explicitly.
    Dim x As Object = Nothing
    x = 123
End Sub
```

Notice that the upgrade remark is emitted even if the variable declaration is now present in the generated code, but you can get rid of this comment by means of a **DisableMessage** pragma.

変数宣言が現在生成されたコードで存在していてもアップグレードされた備考が出力されていることに注意してください。しかし、**DisableMessage** Pragma によってこのコメントを取り除くことができます。

Also, notice that all implicitly-declared variables are Object variable. However, users of VB Migration Partner can use an **InferType** pragma to generate a less-generic data type. For example, the following VB6 code:

また、すべての暗黙的に宣言された変数が Object 変数である場合にも注意してください。しかし、VB Migration Partner Enterprise Edition を使用しているユーザーは、Less - Generic データ型を生成するのに **InferType**Pragma を使用できます。次のコードをご覧下さい。

```
Public Sub Test()
    '## DeclareImplicitVariables
    '## InferType
    '## DisableMessage 05B1
    x = 123
End Sub
```

Is converted to VB.NET as follows:

次のような VB.NET コードに変換されます。

```
Public Sub Test()
    Dim x As Integer
    x = 123
End Sub
```

Unused members／未使用のメンバ

Unused classes, fields, and methods are tagged with a special UPGRADE_INFO comment, which encourages the developer to whether the member is actually unnecessary and possibly delete it from the original VB6 application.

未使用のクラス、フィールド、メソッドについて、UPGRADE_INFO が出力されます。本当に使用していないメンバは、オリジナルの VB6 アプリケーションから削除することを推奨します。

```
Public Sub Test()
    ' UPGRADE_INFO (#0501): 'Test' member isn't used anywhere in current application.
    ' ...
End Sub
```

The code analyzer is smart enough to mark as unused those methods that invoke each other but don't belong to any possible execution path. For example, suppose that method Test references Test2. In this case, VB Migration Partner emits a slightly different message:

コードアナライザーは参照あるいは実行されることのない、未使用のメソッドを、高精度で検出します。例えば、メソッド Test は Test2 を参照すると仮定してください。この場合、VB Migration Partner は異なった表現のメッセージを出力します。

```
Public Sub Test2()
    ' UPGRADE_INFO (#0511): The 'Test' member is referenced only by members that
    ' haven't found to be used in the current project/solution.
    ' ...
End Sub
```

Bear in mind that code analysis can account only for members that are referenced by means of a strongly-typed variable. If the member is referenced exclusively via a Variant, Object, or Control variable, the member is mistakenly

considered to be “unused”. A similar problem occurs if the member is referenced only though a CallByName method. In such cases you can tell VB Migration Partner not to include the field, property, or method in the list of referenced members by means of the MarkAsReference pragma:

コードアナライザーは、厳密に型が指定された変数によって参照されているメンバのみを認識できることを覚えておいてください。もし、メンバがバリアント、オブジェクト、コントロール変数だけに参照されているとしたら、メンバは“未使用”と判断されます。同様の問題は、メンバが CallByName 関数だけに参照されている場合にも起きます。そのような場合、MarkAsReference Pragma を使うことにより、参照されるメンバリストの中にフィールド、プロパティ、メソッドを含めないように VB Migration Partner に設定することが可能です。

```
'## Test.MarkAsReferenced
Public Sub Test()
    '...
End Sub
```

Another case when VB Migration Partner can mistakenly find “unused” members is when migrating an ActiveX DLL or ActiveX EXE project. If you are migrating such a project individually, most of the public properties and methods of its public classes and controls are never referenced elsewhere. Even if you migrate the project as part of a group that contains a client project, chances are that the client project doesn’t reference each and every public member of all public classes of the ActiveX DLL or EXE project.

その他のケースとして、VB Migration Partner は、ActiveX DLL や ActiveX EXE プロジェクトをマイグレーションする時に、誤って“未使用の”メンバを検出してしまいます。それらのプロジェクトを個別にマイグレーションする場合、Public クラスとコントロールのほとんどの Public プロパティとメソッドは、どこからも参照されていません。クライアントプロジェクトを含むグループの一部のプロジェクトをマイグレーションしていたとしても、クライアントプロジェクトは、ActiveX DLL や EXE プロジェクトの全ての Public クラスに存在する Public メンバのどこも参照していない可能性があります。

To cope with this situation, VB Migration Partner supports the MarkPublicAsReferenced pragma, which automatically marks as referenced all the public members in current class. You can apply this pragma at the project level, too, in which case all public members of all public classes in the project are marked as referenced:

この状況に対処するために、VB Migration Partner は MarkPublicAsReferenced Pragma をサポートします。この Pragma は、今のクラスに存在する全ての Public メンバが参照されているものとして、自動的に示します。この Pragma をプロジェクトレベルに当てはめると、プロジェクトの中の全ての Public クラスの全ての Public メンバが参照されているものとして示されます。

```
'## project:AddPublicAsReferenced
```

Because of late-bound references and accesses from external processes, VB Migration Partner doesn’t automatically delete unused members. However, we provide an extender that deletes unused Declare and Const statements, which are never accessed in late-bound mode or from external projects. You can enable this feature in the Extenders tab of the Tools-Options dialog box.

外部プロセスからの Late □Bound 参照とアクセスにより、VB Migration Partner は自動で未使用のメンバを消去することはありません。しかし、我々は Late – Bound モードや外部プロジェクトからアクセスされることのない、未使用の Declare ステートメントや Const ステートメントを消す Extender を提供します。ツール－オプションをクリックすると現れるダイアログボックスのタブで、この Extender の使用が可能です。

Parameters unnecessarily marked as ByRef／ByRef として不必要とみなされたパラメータ

By default, VB6's default passing mechanism is by-reference, and many developers mistakenly omitted the ByVal keyword for parameters that were meant to be passed by value. VB Migration Partner emits an UPGRADE_INFO remark for all parameters that are passed byreference but aren't assigned in the method itself, nor are passed by-reference to methods that modify them. Let's consider the following VB6 code:

VB6 でのパラメータ指定は、デフォルトで参照渡しでした。そして多くの開発者は、値渡しとして定義されるべきパラメータに対しても、誤って ByVal キーワードを省略しました。VB Migration Partner は ByRef で渡され、メソッドの中で割り当てられていない全てのパラメータに対して UPGRADE_INFO を出力しますが、メソッドの中で変更される by-reference 引数は除きます。次のような VB6 コードについて考えてみましょう。

```
Public Sub Test(n1 As Integer, ByRef n2 As Integer, n3 As Integer)
    Test2 n1, n2, n3
End Sub

Public Sub Test2(ByVal n1 As Integer, ByRef n2 As Integer, n3 As Integer)
    Debug.Print n1
    Debug.Print n2
    Get #1, , n3
End Sub
```

Parameter *n1* is never assigned in method Test and can't be modified by Test2 because it is passed into a ByVal parameter. Parameter *n2* is passed to a ByRef parameter of Test2 method, yet Test2 never modifies it. Finally, parameter *n3* is not flagged is passed to Test2 method in a ByRef parameter and it is indeed modified inside that method. Here's is the code that VB Migration Partner emits:

まず、パラメータ n1 はメソッド Test2 の引数指定として ByVal で定義されているため、値を変更することが出来ません。パラメータ n2 はメソッド Test2 で ByRef として渡されますが、メソッド内での値の変更は行われません。最後に、ByRef も ByVal も付いていない（引数指定が省略されている）パラメータ n3 は ByRef として渡され、メソッド Test2 の中で値が変更されます。上記のコードを、VB Migration Partner で変換した場合以下のようなコードになります。

```
Public Sub Test(ByRef n1 As Short, ByRef n2 As Short, ByRef n3 As Short)
    ' UPGRADE_INFO (#0551): The 'n1' parameter is neither assigned in current method nor
    ' is passed to methods that modify it. Consider changing its declaration using the
    ' ByVal keyword.
```

```
        ' UPGRADE_INFO (#0551): The 'n2' parameter is neither assigned in current
method nor
        ' is passed to methods that modify it. Consider changing its declaration
using the
        ' ByVal keyword.
        Test2(n1, n2, n3)
    End Sub

    Public Sub Test2(ByVal n1 As Short, ByRef n2 As Short, ByRef n3 As Short)
        ' UPGRADE_INFO (#0551): The 'n2' parameter is neither assigned in current
method nor
        ' is passed to methods that modify it. Consider changing its declaration
using the
        ' ByVal keyword.
        Debug.WriteLine(n1)
        Debug.WriteLine(n2)
        FileGet6(1, n3)
    End Sub
```

In general, you should consider changing the passing semantics of parameters flagged with the special UPGRADE_INFO remark, because ByVal parameters are often faster than ByRef parameters. The only exception to this rule is structures, and for this reason VB Migration Partner never applies this remark to Structures.

パラメータの意味合いにより UPGRADE INFO が出力されていることを考えてください。例えば、ByVal パラメータは ByRef パラメータより処理されるスピードは早いですが、このルールは構造体の引数には当てはまりません。このため、VB Migration Partner はこのルールを構造体に当てはめません。

Even more important, replacing ByRef parameters with ByVal parameters often avoids hard-to-find bugs in the VB.NET code. 更に重要なことですが、ByRef パラメータを ByVal パラメータに取り替えることで、VB.NET コード上での、見つけることが難しいバグを避けられることがあります。

To help you in applying the ByVal keyword where needed, VB Migration Partner supports the UseByVal pragma, which can be applied at the project, file, or method level. This pragma takes one argument, chosen among the following values:

VB Migration Partner は、必要に応じて ByVal キーワードを当てはめられるように、UseByVal Pragma をサポートします。この Pragma はプロジェクト、ファイル、メソッドレベルで適用でき、下記に示す値から選んだ1つの引数を持ちます。

Yes:
VB Migration Partner applies the ByVal keyword with unassigned parameters that implicitly use by-reference semantics (i.e. have no explicit ByRef or ByVal keyword).

VB Migration Partner は、無条件に by-reference として使用されている割り当てられていないパラメータを ByVal キーワードとして使います。(例えば、ByRef や ByVal キーワードが指定されていない場合)

Force:

VB Migration Partner applies the ByVal keyword with all unassigned parameters that use by-reference semantics, even if an explicit ByRef keyword is found.

VB Migration Partner は、明示的な ByRef キーワードが見つかったとしても、by-reference として使用されている全ての割り当てられていないパラメータを ByVal キーワードとして使います。

No:

VB Migration Partner never applies the ByVal keyword. (This is the default behavior, yet this option can be useful to override another pragma at a broader scope.)

VB Migration Partner は、決して ByVal キーワードを適用しません。(これはデフォルトの動作で、このオプションはより広いスコープで他の Pragma を無視するのに役立ちます。)

Let's see how this pragma can affect the Test and Test2 methods seen in previous example:

この Pragma が上記で検証した Test と Test2 メソッドにどのような影響を与えるかについて見てみましょう。

```
    Public Sub Test(n1 As Integer, ByRef n2 As Integer, n3 As Integer)
        '## UseByVal Yes
        Test2 n1, n2, n3
    End Sub

    Public Sub Test2(ByVal n1 As Integer, ByRef n2 As Integer, n3 As Integer)
        '## UseByVal Force
        Debug.Print n1
        Debug.Print n2
        Get #1, , n3
    End Sub
```

Here's the result from VB Migration Partner:

こちらは VB Migration Partner の結果です。

```
    Public Sub Test(ByVal n1 As Short, ByRef n2 As Short, ByVal n3 As Short)
        ' UPGRADE_INFO (#0551): The 'n2' parameter is neither assigned in current
method nor
        ' is passed to methods that modify it. Consider changing its declaration
using the
        ' ByVal keyword.
        Test2(n1, n2, n3)
    End Sub
```

```
Public Sub Test2(ByVal n1 As Short, ByVal n2 As Short, ByRef n3 As Short)
    Debug.WriteLine(n1)
    Debug.WriteLine(n2)
    FileGet6(1, n3)
End Sub
```

All parameters that were flagged by the special remark in previous example are now defined with the ByVal keyword (and therefore have no remark associated with them), except the *n2* parameter of Test method, because it was defined with an explicit ByRef keyword and we didn't use the "Force" option in that method.

前述の変換例で定義が省略されていた全てのパラメータは、今度は ByVal キーワードを付けて定義されています。（メソッド Test1 は、引数に関連した記述はありませんので、このメソッドでは"Force"オプションを使いませんでした。）

Unreachable code／Unreachable code（どこからも呼ばれないコード）

Sections of unreachable code inside methods are tagged with a special UPGRADE_INFO comment; examples of such sections are those that immediately follow Exit Sub, Exit Function, and Exit Property statements:

メソッドの中でどこからも呼ばれないコードのセクションに対し、特別な UPGRADE INFO コメントが出力されます。そのようなセクションは、Exit Sub、Exit Function、Exit Property ステートメントの後ろに記述されます。

```
Public Sub Test()
    ' ...
    Exit Sub
    ' UPGRADE_INFO (#0521): Unreachable code detected
    Test2()
End Sub
```

VB Migration Partner correctly recognizes as unreachable code the region that follows a label that isn't referenced by any GoTo, GoSub, On GoTo, On GoSub, On Error Goto, or Resume statement.

VB Migration Partner は、GoTo、GoSub、On GoTo、On GoSub、On Error GoTo、On Resume Statement によって参照されない Unreachable code の範囲を正確に認識しています。

Unreachable code detection in current version of VB Migration Partner accounts neither for conditional structures (e.g. If blocks) nor for unhandled errors caused by an Error or Err.Raise statement. For example, in both these examples the code analyzer fails to detect that the call to Test3 is unreachable:

VB Migration Partner の現バージョンでは、条件文（例えば、If 文）とエラーや Err.Raise ステートメントによって起きるハンドルされていないエラーの両方共に、Unreachable code を検出できません。例えば、これら 2 つの例で、コードアナライザーは Test3 が呼ばれないことを検出できません。

```
Public Sub Test(ByVal x As Integer)
```

```
    If x < 0 Then
        ' ...
        Exit Sub
    Else
        ' ...
        Exit Sub
    End If

    Test3
End Sub

Public Sub Test2(ByVal x As Integer)
    ' ...
    Err.Raise 5

    Test3
End Sub
```

The reason why VB Migration Partner doesn't recognize as unreachable code the regions that are preceded by an Error or Err.Raise statement is that such error-related statements are ignored if an On Error Resume Next statement is active. You can have VB Migration Partner recognize those regions as unreachable by simply appending an Exit Sub (or Exit Function or Exit Property) immediately after the Error or Err.Raise statement.

VB Migration Partner がエラーや Err.Rise ステートメントが存在することによってどこからも呼ばれないコード範囲が分かるのに認識しない理由は、On Error Resume Next Statement がアクティブな場合、error-related ステートメントが無視されるためです。エラーや Err.Erase ステートメントの後ろに Exit Sub（あるいは Exit Function や Exit Property）を追加するだけで、VB Migration Partner にこれらの範囲を Unreachable として認識させることが可能です。

Unneeded Class_Initialize events／必要でない Class_Initialize イベント

The only way to initialize class fields under VB6 is to assign them in the Class_Initialize event handlers. (This method is named Form_Initialize inside forms, UserControl_Initialize inside user controls.)

VB6 において、クラスの中で定義される変数を初期化する唯一の方法は、それらを Class_Initialize イベントハンドラー割り当てることです。（この方法は、Form では Form_Initialize、ユーザ・コントロールでは UserControl_Initialize と呼ばれています。）

```
Public StartTime As Date
Public NumberOfDays As Integer

Private Sub Class_Initialize()
        StartTime = Now
        NumberOfDays = 10
End Sub
```

VB Migration Partner attempts to move assignments from the Initialize event handler to the field declaration, thus the above VB6 code is translated to VB.NET as follows:

VB Migration Partner は、変数の初期化を Initialize イベントハンドラからフィールド宣言へ割り当てを動かそうと試みますので、結果として、VB6 のコードは以下のように VB.NET に変換されます。

```
' UPGRADE_INFO (#0721): The 'Class_Initialize' method is empty because all
' its statements were rendered as class field initializers.
Private Sub Class_Initialize_VB6()
End Sub
```

You can tell VB Migration Partner not to generate empty Initialize methods by adding a project- or file-level**RemoveUnusedMembers** pragma in the original VB6 source code.

オリジナルの VB6 ソースコードでプロジェクト或いはファイルレベルに **RemoveUnusedMembers** Pragma を追加して、空の Initialize method を発生させないように VB Migration Partner に指定することができます。

Unneeded Class_Terminate events／必要でない Class_Terminate イベント

Many VB6 developers define a Class_Terminate event handler just to set class-level variables to Nothing. Such assignments are superfluous – because the object is released anyway when the "Me" object is destroyed, but this coding style doesn't harm in VB6.

多くの VB6 開発者が、ただクラスレベル変数を Nothing に設定するために Class_Terminate イベントハンドラを定義します。しかし、この定義は本来必要がありません。なぜなら"Me"オブジェクトが破棄されるとき、オブジェクトは必ず開放されるためです。ただし、このコーディングスタイルは VB6 においては害にはなりません。

When you convert this code to VB.NET, however, the Class_Terminate handler is translated into a Finalize method. As you may know, finalizable VB.NET classes are less efficient than standard classes, therefore you should avoid to declare the Finalize method unless it is strictly necessary. To help you follow .NET Framework coding best practices, VB Migration Partner emits an UPDATE_INFO remark when a Class_Terminate event handler can be safely removed, as in this case:

このコードを VB.NET に変換する際に、Class_Terminate ハンドラは Finalize メソッドに変換されます。ご存知の方もいらっしゃるかもしれませんが、VB.NET のファイナライズクラスは一般的なクラスより効率的ではありません。したがって、それが厳密に必要でない場合、あなたが Finalize メソッドを宣言するのを避けるべきです。VB Migration Partner は最適な.NET Framework コーディングの手助けとなるように、下記のように Class_Terminate イベントハンドラが安全に取り除ける場合は、UPGRADE_INFO コメントを出力します。

```
Private col As New Collection

' UPGRADE_INFO (#0541): This 'Class_Terminate' method appears to be useless. Please
' delete it in the original project and restart migration.
Private Sub Class_Terminate_VB6()
```

```
    col = Nothing
End Sub
```

Unneeded On Error Goto 0 statements／必要でない On Error Goto 0 ステートメント

Many VB6 developers like to reset error handling on exiting a method, as in this code:

多くの VB6 開発者は、次のコードのように、存在するメソッド上でエラー処理をリセットする処理を好みます。

```
Public Sub Test(n1 As Integer, ByRef n2 As Integer, n3 As Integer)
    On Error GoTo ErrorHandler
    Debug.Print n1, n2, 3

ErrorHandler:
    ' ...
    ' reset error handling before exiting
    On Error GoTo 0
End Sub
```

It turns out that the On Error GoTo 0 statement is useless in this case, because the VB error handling is always reset on exiting a method. For this reason, VB Migration Partner remarks out the statement:

この場合、On Error Goto 0 ステートメントは意味のないことが明らかです。なぜなら、VB のエラー処理が、メソッドを抜けるときに常にリセットを行うからです。このために、VB Migration Partner はステートメントをコメントアウトします。

```
Public Sub Test(ByRef n1 As Short, ByRef n2 As Short, ByRef n3 As Short)
    On Error GoTo ErrorHandler
    DebugPrintLine6(n1, TAB(), n2, TAB(), 3)
ErrorHandler:
    ' ...
    ' IGNORED: On Error GoTo 0
End Sub
```

Members without “As” clause／「As」を省略したメンバ

Some VB6 developers – especially those who have written code in other languages, such as C or Pascal – incorrectly assume that the following statement defines three 32-bit variables:

以前 C や Pascal のような他の言語でコードを書いていた VB6 開発者の中には、下記の記述が 3 つの 32 ビット変数を定義していると思い込んでいる人たちがいます。

```
Dim x, y, z As Long
```

Instead, the statement defines two Variant variables and one Long variable. (More precisely, the type of x and y variables depends on the active Def*Xxx* directive and is Variant only if no such a directive exist in current file.) VB

Migration Partner translates the variable declarations correctly, but adds a warning so that you can spot the probable mistake:

そうではなく、この記述は 2 つのバリアント型変数と 1 つの Long 型の変数を定義します。(より正確に言うと、x と y の変数の型は DefXxx ディレクティブに依存し、そのようなディレクティブが存在しない Variant 型のみ現在のファイルに依存しています。)VB Migration Partner は変数の宣言を正しく変換しますが、警告を発します。その結果、存在する可能性のある間違いに気付くことが出来ます。

```
    ' UPGRADE_INFO (#0561): The 'x' symbol was defined without an explicit "As" clause.
    Dim x As Object = Nothing
    ' UPGRADE_INFO (#0561): The 'y' symbol was defined without an explicit "As" clause.
    Dim y As Object = Nothing
    Dim z As Integer
```

A similar warning is emitted for functions, properties, and parameters that lack an explicit “As” clause.

同様な警告が関数、プロパティ、そして明確に「As」が付いていないパラメータに対して発行されます。

String concatenation inside loops／Loop の中における String の連結

In general, VB.NET is slower than VB6 at concatenating strings. As a matter of fact, you should avoid string concatenations inside a time-critical loop, as in this case:

一般的に、VB.NET は String の連結が VB6 より遅いです。実際、この場合のように、時間がかかる Loop の中では String の連結を避けるべきです。

```
    Dim s As String
    Dim n As Integer
    For n = 1 To 10000
        s = s & Str(n) & ","
    Next
```

VB Migration Partner helps you optimize your code by marking string concatenations inside a For, While, or Do loop with a special migration warning. This is how the previous code is converted to VB.NET:

VB Migration Partner は、For、While、あるいは Do Loop の中で行われている String の連結に、特別なマイグレーション警告を伴った目印を付けることにより、コードを最適化する手助けをします。下記のコードは、上記のコードが VB.NET にどのように変換されたかを示すものです。

```
    Dim s As String = ""
    Dim n As Short
    For n = 1 To 10000
        ' UPGRADE_INFO (#0571): String concatenation inside a loop.
```

```
        ' Consider declaring the 's' variable as a StringBuilder6 object.
        s = s & Str6(n) & ","
    Next
```

You should scrutinize all upgrade infos #0571 and decide whether you should change the type of the string variable into System.Text.StringBuilder or the special StringBuilder6 type exposed by VB Migration Partner's support library. The latter type is usually preferable because it minimizes the code edits that are necessary to deliver working code:

全ての#0571 アップグレードインフォメーションを吟味して、System.Text.StringBuilder クラス あるいは VB Migration Partner のサポートライブラリで公表されている特別な StringBuilder6 タイプを使って、String 変数の型を変えるべきか判断してください。 後者を使う方が望ましいです。なぜなら、後者は動いているコードに対して加えるコードの編集量を最小限にするからです。

```
    Dim s As StringBuilder6 = ""
    ' (remainder of code as before)
```

You can also use a SetType pragma to replace the data type from inside the VB6 code:

また、VB6 コードの中からデータ型を置き換えるのに SetType Pragma を使用できます。

```
    '## s.SetType StringBuilder6
    Dim s As String
```

Unused type libraries／未使用のタイプライブラリ

VB6 projects often contain references to type libraries that aren't directly used by the current project. For example, if you create a new VB6 project by selecting the "Data Project" or the "VB Enterprise Edition Controls" template, Visual Basic 6 creates a project that references a number of type libraries that you don't actually use. VB Migration Partner detects all unreferenced type libraries and lists them at the top of one of the VB.NET source files:

VB6 プロジェクトは、現在のプロジェクトでは直接使われていないタイプライブラリの参照をよく含みます。例えば、"Data Project"や"VB Enterprice Edition コントロール"テンプレートを選択することによって、新しい VB6 プロジェクトを生成した場合、Visual Basic 6 は実際には使用しない、いくつかのタイプライブラリを参照するプロジェクトを生成します。VB Migration Partner は全ての参照していないタイプライブラリを検知し、VB.NET ソースファイルの一つとして、一番上にリストアップします。

```
    ' UPGRADE_INFO (#0571): The 'DDSLibrary' type library is never used in current project.
    ' Consider deleting it from VB6 project references.
```

Notice that the unused reference is *not* removed automatically from the converted VB.NET project. It's up to the developer to decide whether the type library is truly unnecessary and remove it manually from the original VB6 project. This manual fix ensures that the VB.NET project contains only the references that are strictly necessary and speeds up the conversion process.

変換した VB.NET プロジェクトから、使われていない参照が自動的に除かれていないことに注目してください。タイプライブラリが本当に必要でないか、そしてオリジナルの VB6 プロジェクトから手動で取り除くかを決めるのは開発者です。このマニュアルの対処通りにすると、VB.NET プロジェクトは厳密に必要な参照だけを含み、変換プロセスのスピードを上げることを保証します。

4.3 Refactoring features／リファクタリング機能

VB Migration Partner leverages code analysis techniques to refactor the VB.NET to make faster and more readable.

VB Migration Partner は、コード分析技術を用いて、VB.NET 上でより早く、より読みやすいコードにリファクタリングします。

Return values／戻り値

VB6's Function and Property Get procedures return a value to their caller by assigning the value to a local variable named after the procedure itself, as in this code:

VB6 の関数とプロパティを取得するプロシージャは、プロシージャの名前をとって付けられたローカル変数に値を割り当てることによって、呼び出し元に値を戻します。

```
Function GetData() As Integer
    ...
    If x > 0 Then
        GetData = x
        Exit Function
    End If
    ...
End Function
```

VB Migration Partner is able to collapse the assignment and the Exit statement into a VB.NET Return statement if it's safe to do so:

VB Migration Partner は上記のように変数に値を割り当てないこともできます。VB.NET の中では、VB6 の Exit ステートメントは下記のように安全な Return ステートメントに変わります。

```
Function GetData() As Short
    ...
    If x > 0 Then
        Return x
    End If
    ...
End Function
```

This feature is quite sophisticated and works as expected even in more intricate cases, as in the following example:

この特徴はとても洗練されていて、下記のように、より複雑な場合でも期待された通りに動きます。

```
Function GetData() As Integer
    ...
    If x > 0 Then
        If y > 0 Then
            GetData = y
        Else
            GetData = x
        End If
        Exit Function
    End If
    ...
    GetData = 0
End Function
```

which translates to:

これを移行すると、

```
Function GetData() As Short
    ...
    If x > 0 Then
        If y > 0 Then
            Return y
        Else
            Return x
        End If
        Exit Function
    End If
    ...
    Return 0
End Function
```

Notice that the Exit Function keyword is left in code and must be removed manually.

Exit Function キーワードがコードの中に残されていて、手で取り除かねばいけないことに注意してください。

Variable initialization／変数の初期化

VB Migration Partner attempts to merge a local variable's declaration with its initialization. For example, the following VB6 code:

VB Migration Partner はローカル変数の宣言時に初期化を一緒に行おうとします。次の VB6 コードをご覧ください。

```
Dim d As Double, i As Integer, v As Long, o As Object
d = 1.8
i = 11
...
If d = 0 Then v = 111
```

is translated to VB.NET as

このコードを VB.NET に移行すると、

```
Dim d As Double = 1.8
Dim i As Integer = 11
Dim v As Long
Dim o As Object = Nothing
...
If d = 0 Then v = 111
```

In this example, the *d* and *i* variables can be safely declared and initialized in a single statement, whereas the *v* variable can't. This feature is enabled only for strings and scalar variables; Object and other reference type variables are always explicitly initialized to Nothing to prevent the VB.NET compiler from complaining about uninitialized variables that might throw a NullReference exception.

この例では、変数 d と変数 i が 1 行で安全に宣言・初期化されているのに、変数 v はそうではありません。これはストリングスやスカラー変数で見られる特徴です。オブジェクトや他の参照型変数の規定値は常に Nothing に設定され、VB.NET コンパイラーが NullReference Exception を投げる可能性のある、初期化されていない変数によるエラーが出るのを避けています。

VB Migration Partner takes a conservative approach and doesn't merge a variable declaration with the first assignment to that variable if the code block between these two statements includes keywords that change execution flow, for example method calls or Select Case blocks, or if the value being assigned isn't constant.

VB Migration Partner は時には変数の初期化と値の代入を同時に行わないことがあります。それは、これら 2 つのステートメントの間のコードブロックに、関数の呼び出しや Select Case ブロック、一定でない値の代入のような change 実行フローキーワードを含む場合です。

You can force variable initialization by adding a MergeInitialization pragma for that specific variable. Consider this code fragment:

特別な変数に MergeInitialization Pragma を用いることにより、変数の初期化を強力に行うことができます。このコードの断片を考えてみてください。

```
'## n1.MergeInitialization Force
Dim n1 As Integer, n2 As Integer
```

```
n1 = 10 + GetValue()
n2 = 11 + n1
```

In this case the MergeInitialization Force pragma informs VB Migration Partner that the call to the GetValue method can be safely included in the variable initialization. The code generator merges the declaration and initialization of *n1* variable, because of the MergeInitialization pragma, but not of *n2* variable:

この場合、VB Migration Partner は MergeInitialization Force Pragma を使うことで、GetValue メソッドが変数の初期化で安全に使用されていることを認識します。コード生成プログラムは、MergeInitialization Pragma によって n1 変数の宣言と初期化を同時に行いますが、n2 変数に対しては行いません。

```
Dim n1 As Short = 10 + GetValue()
Dim n2 As Short
n2 = 11 + n1
```

You can disable this optimization at the project, file, method, or variable level by means of the following pragma:

次に記す Pramga を使うことによって、プロジェクト、ファイル、メソッド、変数レベルの最適化を無効にすることができます。

```
'## MergeInitialization No
```

VB Migration Partner is also able to merge class field declarations and assignments found in the Class_Initialize (or Form_Initialize) methods. However, the MergeInitialization pragma has no effect on class fields.

VB Migration Partner はクラスフィールド宣言とクラスの初期化（あるいは Form_Initialize）メソッドを一緒に行うこともできます。けれども、MergeInitialization Pragma はクラスフィールドに対して効果がありません。

Note:
in a method that contains one or more Gosub statements that are converted to separate methods, because of a ConvertGosubs pragma, the variable initialization feature is disabled.

別のメソッドに変換される 1 つかそれ以上の GoSub ステートメントを含むメソッドにおいて、ConvertGosubs Pragma を用いると、変数初期化の特徴は無効になります。

Compound assignment operators／複合代入演算子

VB Migration Partner is able to replace plain assignment symbols with compound assignment operators, such as += or /=. For example, the following code:

VB Migration Partner は簡単な代入記号を複合代入演算子（例えば、+= や /=）に変換することができます。

```
sum = sum + 1.8
value = value ¥ 2
text = text & "abc"
```

```
Label1.Caption = Label1.Caption & "."
Label2 = Label2 + "< end > "
```

is translated to VB.NET as follows:

VB.NET に変換すると、

```
sum += 1.8
value ¥= 2
text & = "abc"
Label1.Caption & = "."
Label2.Caption & = " < end > "
' notice that default property has been resolved
デフォルトプロパティが解決されていることに注目してください。
```

Wrapping fields in properties／プロパティのラッピングフィールド

VB Migration Partner can automatically wrap a specific public field in a class (or all public fields in the current class or project) in an equivalent Property. This feature is enabled by means of the AutoProperty pragma. For example, the following VB6 code:

VB Migration Partner Enterprise Edition は、クラスの中の特定の Public フィールド(或いは、今のクラスやプロジェクトの中の全ての Public フィールド)を、同等なプロパティとして、自動的にラップすることが出来ます。 この特徴は AutoProperty Pragma によって実現されます。下記の VB6 コードをご覧ください。

```
'## AutoProperty

Public Name As String
Public Widget As New Widget
```

is rendered as follows:

は下のように変換されます。

```
Public Property Name() as String
    Get
        Return Name_InnerField
    End Get
       Set(ByVal value As String)
        Name_InnerField = value
    End Set
End Property
```

```
Private Name_InnerField As String = ""

Public Property Widget() As Widget
    Get
        If Widget_InnerField Is Nothing Then Widget_InnerField = New Widget()
        Return Widget_InnerField
    End Get
    Set(ByVal value As Widget)
        Widget_InnerField = value
    End Set
End Property

Private Widget_InnerField As Widget
```

Notice that the AutoProperty pragma automatically enforces the auto-instancing (As New) semantics if possible, regardless of whether the variable is under the scope of an AutoNew pragma.

変数が AutoNew Pragma のスコープの影響を受けているかどうかに関係なく、AutoProperty Pragma は自動でインスタンスの生成(As New)を行うことに注意してください。

Appending to the Text property／テキストプロパティへの追加

When the original VB6 code appends a string to the Text property of a TextBox, MaskEdBox, RichTextBox, or WLText control, VB Migration Partner can translate this operation into a more efficient call to the AppendText method. This is a special case of the pattern described at previous point:

オリジナルの VB6 コードは、テキストボックス、MaskEdBox、リッチテキストボックス、WLText コントロールのテキストプロパティに string を付け加えるのに対し、VB Migration Partner はこの機能をより有効な AppendText メソッドに変換します。このことは以前のポイントで記した場合の特別なケースです。

```
Me.Text1.Text = Me.Text1.Text & "< end > "
Text2 = Text2 + "< end > "
```

becomes

は、下記のように変換されます。

```
Me.Text1.AppendText("< end > ")
Text2.AppendText("< end > ")
```

If … Else simplification／If … Else の簡略化

VB Migration partner is able to drop If blocks if the Then and the Else blocks contain the same code. Such blocks can be produced, for example, by a conversion of a VB6 property. A VB6 Variant propery can have both the Property Let and Property Set blocks, if the property can be assigned either a scalar or an object value, as in this example:

VB Migration Partner は、Then や Else ステートメントに同じコードを含む場合、If 節を削除します。そのような例は、VB6 プロパティを変換した時に起こります。VB6 バリアントプロパティには Property Let と Property Set の両方が存在します。プロパティがスカラー値かオブジェクト値のどちらかなら、プロパティは次のように割り当てられます。

```
Private m_Tag As Variant

Property Get Tag() As Variant
    If IsObject(m_Tag) Then
        Set Tag = m_Tag
    Else
        Tag = m_Tag
    End If
End Property

Property Let Tag(ByVal newValue As Variant)
    m_Tag = newValue
End Property

Property Set Tag(ByVal newValue As Variant)
    Set m_Tag = newValue
End Property
```

Converting this property literally delivers the following VB.NET code:

このプロパティをありのままに変換すると、下記のような VB.NET コードになります。

```
Private m_Tag As Object

Public Property Tag() As Object
    Get
        If IsObject6(m_Tag) Then
            Return m_Tag
        Else
            Return m_Tag
        End If
    End Get
    Set(ByVal newValue As Object)
        If IsObject6(newValue) Then
            m_Tag = newValue
        Else
            m_Tag = newValue
        End If
    End Set
```

```
End Property
```

VB Migration Partner detects that the Then and the Else portions of the If blocks contain the same executable statements and can simplify them as follows:

VB Migration Partner は、If ブロックの Then と Else に同じ実行文が含まれていて、下記のように簡略化できることを検知します。

```
Private m_Tag As Object

Public Property Tag() As Object
    Get
        Return m_Tag
    End Get
    Set(ByVal newValue As Object)
        m_Tag = newValue
    End Set
End Property
```

The IsNot operator／IsNot 演算子

VB Migration Partner can automatically to adopt the IsNot operator in all expressions that test whether two objects are equal or if an object is equal to Nothing. For example, the following VB6 code:

VB Migration Partner は 2 つのオブジェクトが等しいか、それともオブジェクトが Nothing かどうか調べる全ての式の中で、自動的に IsNot 演算子を選んで使います。下記の VB6 コードをご覧ください。

```
If Not obj Is Nothing And Not obj Is obj2 Then
```

Is translated as follows:

これは下のように変換されます。

```
If obj IsNot Nothing And obj IsNot obj2 Then
```

Nested If merging／If 文の入れ子

VB Migration Partner can detect whether the original VB6 code contains two or more nested If blocks and automatically merge them into a single If condition that contains the AndAlso operator. The key to this refactoring feature is the **MergeIfs** pragma, which can have project-, file-, or method-scope. For example, consider the following VB6 code:

VB Migration Partner Enterprise Edition は、オリジナルの VB6 コードに、2 つ以上の入れ子にされた If ブロックを含んでいるかを調べ、自動的に AndAlso 演算子を含んだ状態の一つの If 文に統合します。このリファクタリングの機能の鍵は **MergeIfs** Pragma です。(プロジェクト、ファイル、メソッドで範囲指定が出来ます)例えば、下記の VB6 コードについて考えてみてください。

```
Sub Test(ByVal obj As Object, ByVal obj2 As Object)
    '## MergeIfs True
    If Not obj Is Nothing Then
        If obj.Text = "" Then Debug.Print "Empty Text"
    End If

    If Not obj Is Nothing Then
        If Not obj2 Is Nothing Then
            If obj.Text = obj2.Text Then Debug.Print "Same text"
        End If
    End If
End Sub
```

This is how VB Migration Partner translates it to VB.NET

VB Migration Partner は VB.NET に次のように変換します。

```
Sub Test(ByVal obj As Object, ByVal obj2 As Object)
    ' UPGRADE_INFO (#0581): Two nested If...End If blocks have been merged.
    If (obj IsNot Nothing) AndAlso (obj.Text = "") Then
        Debug.WriteLine("Empty Text")
    End If

    ' UPGRADE_INFO (#0581): Three nested If...End If blocks have been merged.
    If (obj IsNot Nothing) AndAlso (obj2 IsNot Nothing) AndAlso (obj.Text =
obj2.Text) Then
        Debug.WriteLine("Same text")
    End If
End Sub
```

You can further simplify the expression by dropping the parenthesis around each subexpressions, by using the following pragma:

各部分式の周りの()を削除すると、より式を簡単にできます。それを実現するために、下記に記す Pragma を使ってください。

```
'## MergeIfs True, False
```

Structured Exception Handling／構造化例外処理

VB Migration Partner can replace old-styled On Error Goto statements into structured exception handling (SEH) based on the Try-Catch block. You can activate this feature by means of the UseTryCatch pragma, which can have project-, file-, and method-level scope. For example, the following VB6 code:

VB Migration Partner Enterprise edition は古い形式の On Error Goto ステートメントを Try-Catch ブロックに基づいた構造化例外処理(SEH)へ替えます。UseTryCatch pragma を使うことによって、この特色を実現することができます。UseTryCatch Pragma はプロジェクト、ファイル、メソッドレベルで範囲指定が出来ます。下記の VB6 コードをご覧ください。

```
Sub Test()
    '## UseTryCatch
    On Error Goto ErrorHandler
    ' method body
 ErrorHandler:
    ' error handler
 End Sub
```

is translated to the following VB.NET code:

上で挙げた例は、次のような VB.NET コードに変換されます。

```
Sub Test()
    Try
        ' IGNORED: On Error Goto ErrorHandler
        ' method body
    Catch _ex As Exception
        ' IGNORED: ErrorHandler:
        ' error handler
    End Try
End Sub
```

The Try…Catch block can be inserted only if the following conditions are all true:

Try. . .Catch ブロックは下記に挙げる状態が true の場合に挿入されます。

a. The method contains only a single On Error GoTo <label> method.
   メソッドの中に On Error GoTo＜ラベル＞メソッドが一つ存在する。
b. The On Error GoTo <label> statement doesn't appear inside a conditional block such as If, For, Select Case, and isn't preceded by a GoTo statement.
   If、For、Select Case のような条件節の中で On Error GoTo＜ラベル＞ステートメントが現れず、GoTo ステートメントの前にも現れない場合
c. The method doesn't contain GoSub, Resume, or Resume Next statements. (Resume <label> statements are acceptable, though)
   GoSub、Resume、Resume Next ステートメントがメソッドに存在しない場合
d. There is no Goto statement in the method body that points to a label that is defined in the error handler. (Notice that it is ok if a GoTo in the error handler points to a label defined in the method body.)
   エラーハンドラーが定義されているラベルへジャンプする Goto ステートメントがメソッド本体に存在

しない場合（エラーハンドラーの中に存在する GoTo が、メソッドに定義されたラベルへジャンプすることは大丈夫なことに注意してください。）

e. If the method contains one or more On Error Goto 0 statements, such statements must immediately precede a statement that causes exiting from the current method, e.g. Exit Sub or End Function.
メソッドに 1 つ以上の On Error GoTo 0 ステートメントを含む場合は、Exit Sub や Exit Function のような、現在のメソッドをすぐに終了させるコードの手前になくてはならない。

f. If the current method is a Property Let procedure, there must not be a Property Set procedure for the same property. Likewise, If the current method is a Property Set procedure, there must not be a Property Let procedure for the same property.
メソッドに Property Let プロシージャが存在するときに、Property Set プロシージャは絶対に存在しません。同じように、メソッドに Property Set プロシージャが存在するときに、Property Let プロシージャは存在しません。

For example, the following VB6 code:

例えば、次の VB6 コードは、

```
Sub Test()
    '## UseTryCatch
    On Error Goto ErrorHandler
    ' method body
    If x > 0 Then GoTo ErrorHandler2

ErrorHandler:
    ' error handler
ErrorHandler2:
    ' do something here
End Sub
```

can't be converted to VB.NET using a Try-Catch block because a GoTo statement in the method body points to a label that is defined in the error handler.

Try-Catch ブロックを使って VB.NET には変換されません。理由は、メソッドに存在している GoTo ステートメントが、エラーハンドラとして定義されている行にジャンプするからです。

By default, VB Migration Partner inserts a Try-Catch also if the method contains the On Error Resume Next statement plus another executable statement. For example, the following VB6 code:

デフォルトで、メソッドが On Error Resume Next ステートメントと別の実行文を含んでいるなら、VB Migration Partner は Try- Catch を挿入します。 例えば、以下の VB6 コードを御覧下さい。

```
Function Reciprocal(ByVal x As Double) As Double
    '## UseTryCatch True
```

```
        On Error Resume Next
        Reciprocal = 1 / x
        ' returns zero if an error occurs (e.g. X is zero)
    End Function
```

contains only two executable statements (including On Error Resume Next) and is converted to:

上に挙げた例では、On Error Resume Next を含むと 2 つの実行可能なステートメントを含みます。そして、次のように変換されます。

```
    Function Reciprocal(ByVal x As Double) As Double
        Try
            Return 1 / x
            ' returns zero if an error occurs (e.g. X is zero)
        Catch
            ' Do nothing if an error occurs
        End Try
    End Function
```

Methods with three or more executable statements can be converted using Try Catch if you specify a value higher than 2 in the second argument for the UseTryCatch pragma. For example, all methods with up to 5 executable statements under the scope of the following pragma:

3 つ以上の実行可能なステートメントを持つメソッドは、UseTryCatch Pragma の 2 つ目の引数に 2 より大きい値を指定することによって Try Catch を使って変換できます。例えば、下に記すような Pragma を用いると、5 つまで実行可能なステートメントを持つメソッドが、

```
    '## UseTryCatch True, 5
```

are converted using a Try-Catch block.

Try-Catch ブロックを使って変換されます。

Notice that the UseTryCatch pragma can be used together with the AutoDispose Force pragma, in which case VB Migration Partner generates a complete Try-Catch-Finally block.

UseTryCatch pragma は AutoDispose Force Pragma と一緒に使われることに注目してください。この 2 つの Pragma を一緒に使うことで、VB Migration Partner は Try-catch-Finally ブロックを生成します。

GoSub refactoring／GoSub リファクタリング

VB Migration Partner is able, in many circumstances, to automatically refactor old-styled Gosub keywords into calls to a separate method. This feature can be applied at the project-, file-, or method-level by means of the**ConvertGosubs** pragma. For example, consider the following VB6 code:

VB Migration Partner Enterprise は多くの場合、古い形式の GoSub キーワードを別のメソッドを呼び出すように自動的にリファクタリングしています。**ConvertGosubs** Pragma を使うことによって、この特徴をプロジェクト、ファイル、メソッドレベルで適用可能です。下のような VB6 のコードについて考えてみましょう。

```
Function GetValue(x As Integer) As Integer
    '## ConvertGosubs True
    On Error Resume Next

    Dim s As String, name As String
    s = "ABC"
    name = "Code Architects"
    GoSub BuildResult
    Exit Function
BuildResult:
    GetValue = x + Len(s) + Len(name)
    Return
End Function
```

VB Migration Partner detects that the code that begins at the BuildResult label (a.k.a. the target label) can be safely refactored to a separate method, that receives four arguments. The result of the conversion is therefore:

VB Migration Partner は BuildResult ラベル（ターゲットラベル）で始まるコードは安全に別のメソッドへリファクタリングされ、4 つの引数を取ることを見つけます。変換すると、次のようになります。

```
Public Function GetValue(ByRef x As Short) As Short
    Dim s As String
    Dim name As String
    On Error Resume Next

    s = "ABC"
    name = "Code Architects"
    Gosub_GetValue_BuildResult(x, GetValue, s, name)
    Exit Function
End Function

Private Sub Gosub_GetValue_BuildResult(ByRef x As Short, ByRef GetValue As
Short, _
    ByRef s As String, ByRef name As String)
    On Error Resume Next
    GetValue = x + Len6(s) + Len6(name)
End Sub
```

Notice that the external method contains an On Error Resume Next statement, because the original GetValue method also contains this statement.

形式上、メソッドは On Error Resume Next ステートメントを含むことに注目してください。それは、オリジナルの GetValue メソッドが On Error Resume Next ステートメントを含むからです。

All arguments to the new *Gosub_GetValue_BuildResult* method are passed by reference, so that the caller can receive any value that has been modified inside the method (as is the case with the *GetValue* parameter in previous example.) You can have VB Migration Partner optimize this passing mechanism and use ByVal if possible by passing True as the second argument to the ConvertGosubs pragma:

新しい Gosub_GetValue_BuildResult メソッドの全ての引数は参照渡しなので、呼び元はメソッドの中でどんな値に変更されたとしても値を受け取ることができます。(以前例に挙げた GetValue パラメータと同じケースです。) VB Migration Partner でこの引数渡しのメカニズムを最大限に利用することが可能であり、ConvertGosubs Pragma の 2 番目の引数に True を設定することによって ByVal を使うことも可能となります。

```
'## ConvertGosubs True, True
```

If this optimization is used in the above code, the separate method is rendered as follows:

上記のコードにこの Pragma を使うと、分割されるメソッドは下記のように変わります。

```
    Private Sub Gosub_GetValue_BuildResult(ByVal x As Short, ByRef GetValue As
Short, _
        ByVal s As String, ByVal name As String)
        On Error Resume Next
        GetValue = x + Len6(s) + Len6(name)
    End Sub
```

If you don't like the name that VB Migration Partner assigns to the automatically-generated method, you can easily change it by means of a **PostProcess** pragma:

VB Migration Partner が自動的に生成するメソッド名が好みでないならば、**PostProcess** Pragma を使うことによって間単に名前を変えることができます。

```
'## PostProcess "Gosub_GetValue_BuildResult", "AssignValueResult"
```

It is important to keep in mind that the conversion of a Gosub block of code into a separate method isn't always possible. More precisely, VB Migration Partner can perform this conversions only if all the following conditions are met:

GoSub ブロックコードを別のメソッドに変換することがいつも可能でないことを覚えておくことは、とても重要です。もっと詳しく言うと、VB Migration Partner は下に記すような条件が満たされた場合にのみ、この変換を行います。

a. The method doesn't contain any Resume statement. (On Error Resume Next is OK, though).
   メソッドに Resume ステートメントを含んでいない。(On Error Resume Next は OK です。)
b. The target label isn't located inside a If, Select, For, Do, or Loop block.
   ターゲットラベルが If、Select、For、Do、Loop ブロックの中に存在しない。

c. The target label isn't referenced by a Goto, On Goto/Gosub, or On Error statement.
ターゲットラベルが Goto、On Goto/GoSub、On Error Statement に参照されない。
d. The target label must be preceded by a Goto, Return, End, Or Exit Sub/Function/Property statement.
ターゲットラベルが Goto、Return、End、Exit Sub/Function/Property ステートメントの前に存在している。
e. The code block must terminate with an unconditional Return, or an End Sub/Function/Property statement. (Blocks that terminate with Exit Sub/Function/Property statements aren't converted.)
コードブロックが Return、End Sub/Function/Property ステートメントで終わる。（Exit Sub/Function/Property ステートメントで終わるブロックは変換されません。）
f. The block of code between the target label and the closing Return/End statement doesn't contain another label.
ターゲットラベルと「終了」を示す Return/End ステートメント間のコードブロックに別のラベルを含まない。

If a method contains one or more labels that can't be converted to a separate method, VB Migration Partner still manages to convert the remaining labels. In general, a label can be converted to a separate method if it neither references nor is referenced by a label that can't be converted (for example, a label that is defined inside an If block or a label that doesn't end with an unconditional Return statement.)

VB Migration Partner は、変換出来なかったラベルの他に、1 つ以上のラベルがメソッドの中に含まれている場合、引き続き残りのラベルをなんとか変換しようと試みます。それらは、変換できなかったラベルとの参照関係がなければ、一般的に、別のメソッドに変換されます。（例えば、If 文の中で定義されたラベルや、Return ステートメントで終わらなかったラベルなどです。）

Note:
in a method that contains one or more Gosub statements that are converted to separate methods, the variable initialization merging feature is disabled.

別のメソッドに変換された 1 つ以上の GoSub ステートメントを含んだメソッドにおいて、変数の初期化と宣言は同時には起こりません。

Type inference／型の推測

VB Migration Partner is capable, in most cases, to correctly deduce a less generic data type for Variant local variables, parameters, class fields, functions and properties. This applies both to members that were explicitly declared as Variant and to members that lacked an explicit As clause. To see how this feature works, consider the following VB6 code:

VB Migration Partner Enterprise edition は、たくさんのバリアント型のローカル変数、パラメータ、クラスフィールド、関数とプロパティのデータ型を正しく推測できます。これは、バリアント型として明確に定義されたメンバと明確に定義されていないメンバの両方にあてはまります。この特徴の動作を確認するために、以下の VB6 コードを考察してください。

```
'## DeclareImplicitVariables True

Private m_Width As Single

Property Get Width ()
    Width = m_Width
End Property

Property Let Width (value)
    m_Width = value
End Property

Sub Test(x As Integer, frm)
    Dim v1 As Variant, v2
    v1 = x * 10
    v2 = Width + x
    Set frm = New frmMain
    res = frm.Caption
End Sub
```

By default this code is converted to VB.NET as follows:

デフォルトで、このコードは下記のような VB.NET に変換されます。

```
    Private m_Width As Single

    Public Property Width() As Object
        Get
            ' UPGRADE_WARNING (#0364): Unable to assign default member of symbol
'Width'.
            ' Consider using the SetDefaultMember6 helper method.
            Return m_Width
        End Get
        Set(ByVal value As Object)
            ' UPGRADE_WARNING (#0354): Unable to read default member of symbol
'value'.
            ' Consider using the GetDefaultMember6 helper method.
            m_Width = value
        End Set
    End Property

    Public Sub Test(ByRef x As Short, ByRef frm As Object)
        ' UPGRADE_INFO (#05B1): The 'res' variable wasn't declared explicitly.
```

```
        Dim res As Object = Nothing
        ' UPGRADE_INFO (#0561): The 'frm' symbol was defined without an explicit
"As" clause.
        Dim v1 As Object = Nothing
        ' UPGRADE_INFO (#0561): The 'v2' symbol was defined without an explicit "As"
clause.
        Dim v2 As Object = Nothing
        ' UPGRADE_WARNING (#0364): Unable to assign default member of symbol 'v1'.
        ' Consider using the SetDefaultMember6 helper method.
        v1 = x * 10
        ' UPGRADE_WARNING (#0364): Unable to assign default member of symbol 'v2'.
        ' Consider using the SetDefaultMember6 helper method.
        v2 = Width + x
        frm = New Form1()
        ' UPGRADE_WARNING (#0354): Unable to read default member of symbol
'frm.Caption'.
        ' Consider using the GetDefaultMember6 helper method.
        ' UPGRADE_WARNING (#0364): Unable to assign default member of symbol 'res'.
        ' Consider using the SetDefaultMember6 helper method.
        res = frm.Caption
    End Sub
```

Variant variables – either implicitly declared or not – are translated into Object variables, which causes several warning to be emitted and, more important, adds an overhead at runtime. This overhead can be often avoided by declaring the variable of a more defined scalar type. You can have VB Migration Partner change the data type for a given member by means of the **SetType** pragma, but this operation requires a careful analysis of which values are assigned to the variable.

バリアント型の変数(暗黙の宣言を含む)はオブジェクト型に変換されますが、それが原因でいくつかの警告が出され、より重要なこととして、処理に時間がかかってしまうようになります。このオーバーヘッドは、より厳密に定義されたスカラー型の変数で宣言することで避けることができます。**SetType** Pragma を使うことによって、メンバ変数のデータ型が変わったことを VB Migration Partner に知らせることができます。けれども、この機能は変数に割り当てられた値に対して慎重な分析を必要とします。

Users of VB Migration Partner can have this analysis performed automatically, by inserting an **InferType** pragma at the project-, file-, method- or variable-level. For example, let's assume that the previous VB6 code contains the following pragma at the file level:

VB Migration Partner Enterprise edition の利用者は、プロジェクト、ファイル、メソッド、変数レベルで **InferType**Pragma を挿入することで、この分析を自動で行わせることができます。ファイルレベルにこのコードを使った VB6 コードの例を見てみることにしましょう。

```
    '## InferType Yes
```

The **Yes** argument in this pragma makes VB Migration Partner attempt to infer the type of al the implicitly-declared local variables and of all the local variables, class fields, functions and properties that lack an explicit As clause:

このPragmaに引数**Yes**を指定することで、VB Migration Partnerは全ての厳密に宣言されたローカル変数と、厳密に定義されていない全てのローカル変数、クラスフィールド、関数、プロパティの型の推測を試みることができます。

```
    Private m_Width As Single

    Public Property Width() As Single
        Get
            Return m_Width
        End Get
        Set(ByVal value As Single)
            m_Width = value
        End Set
    End Property

    Public Sub Test(ByRef x As Short, ByRef frm As Object)
        ' UPGRADE_INFO (#0561): The 'frm' symbol was defined without an explicit
"As" clause.
        ' UPGRADE_INFO (#05B1): The 'res' variable wasn't declared explicitly.
        Dim res As Object = Nothing
        Dim v1 As Object = Nothing
        ' UPGRADE_INFO (#0561): The 'v2' symbol was defined without an explicit "As"
clause.
        Dim v2 As Single
        ' UPGRADE_WARNING (#0364): Unable to assign default member of symbol 'v1'.
        ' Consider using the SetDefaultMember6 helper method.
        v1 = x * 10
        v2 = Height + x
        frm = New Form1()
        ' UPGRADE_WARNING (#0354): Unable to read default member of symbol
'frm.Caption'.
        ' Consider using the GetDefaultMember6 helper method.
        ' UPGRADE_WARNING (#0364): Unable to assign default member of symbol 'res'.
        ' Consider using the SetDefaultMember6 helper method.
        res = frm.Caption
    End Sub
```

In this new version, VB Migration Partner can infer the type of the *Width* property from the type of the *m_Width*variable, which in turn permits to infer the type of the v2 local variable. VB Migration Partner attempts to infer neither the type of the *v1* local variable (because it is explicitly declared As Variant) nor the type of the *frm*parameter (because by default method parameters aren't under the scope of the InferType pragma).

今度の新しいバージョンでは、VB Migration Partner は m_Width 変数の型から Width プロパティの型を推測することができ、同様に v2 ローカル変数の型を推測します。VB Migration Partner は、v1 ローカル変数の型（v1 ははっきりとバリアント型で宣言されています。）と frm パラメータ（デフォルトで、メソッドのパラメータは InferType pragma のスコープ範囲にありません。）の両方ともに推測しようとしません。

You can extend type inference to members that were declared with an explicit As Variant by using the **Force**argument in the pragma. In addition, passing True in the second (optional) argument extends the scope to method parameters:

InferType Pragma に引数 **Force** を指定することで、明示的にバリアント型で宣言されたメンバに対しても型の推理が可能となります。さらに 2 つ目の任意の引数に True を指定することで、メソッドパラメータにスコープを広げられます。

```
'## InferType Force, True
```

The result of the conversion is now the following:

変換した結果は以下のようになります。

```
Private m_Width As Single

Public Property Width() As Single
    Get
        Return m_Width
    End Get
    Set(ByVal value As Single)
        m_Width = value
    End Set
End Property

Public Sub Test(ByRef x As Short, ByRef frm As Form1)
    ' UPGRADE_INFO (#0561): The 'frm' symbol was defined without an explicit
"As" clause.
    ' UPGRADE_INFO (#05B1): The 'res' variable wasn't declared explicitly.
    Dim res As String = Nothing
    Dim v1 As Integer
    ' UPGRADE_INFO (#0561): The 'v2' symbol was defined without an explicit "As"
clause.
    Dim v2 As Single
    v1 = x * 10
    v2 = Width + x
    frm = New Form1()
    res = frm.Caption
End Sub
```

Notice that in this new version the type of *frm* parameter is inferred correctly, which in turn permits to infer the type of the *res* local variable.

今度のバージョンでは、frm パラメータの型が正しく推測され、同様に res ローカル変数の型を推測することに注目してください。型の推測はスカラ型とオブジェクト型の両方で行われます。次のような VB6 コードについて考えてみましょう。

Type inference works both with scalar types and object types. For example, consider the following VB6 code:

VB Migration Partner は、変数 tb が TextBox 型であることを推測でき、デフォルトプロパティを正しく拡張できます。

```
'## InferType
Public Sub SetAddress(ByVal address As String)
    Dim tb
    Set tb = Me.txtAddress    ' where txtAddress is a TextBox
    tb = address
End Sub
```

VB Migration Partner can infer that the *tb* variable is of type TextBox and, consequently, it can correctly expand its default property:

メソッドパラメータへ Pragma スコープを広げていくことは、常に良策であるとは限りません。それは、VB Migration Partner の現バージョンでは、メソッドの中での割り当てを見ることによりパラメータの型を推測しており、メソッドの呼び出しによって生じる暗黙の割り当てについては評価しないからです。このため、マイグレーションで推測される型は、ときには正しくないかもしれません。

```
Public Sub SetAddress(ByVal address As String)
    Dim tb As VB6TextBox = Me.txtAddress
    tb.Text = address
End Sub
```

Extending the pragma scope to method parameters isn't always a good idea, because the current version of VB Migration Partner infers the type of parameters by looking at assignments *inside* the method and doesn't account for implicit assignments that result from method calls. For this reason, the type inferred during the migration process might not be correct in some cases.

重要事項:型の推測は正確な技術ではないので、VB Migration Partner によって推測されたデータ型が正しいかどうかを、いつもダブルチェックすべきです。よりよい方法としては、マイグレーションの早い段階だけに InferType Pragma を使い、その後は元の VB6 コードを編集するか、もしくは SetType Pragma を使うことによってメンバに特別な型を割り当てることです。

Important note:
Type inference isn't an exact science and you should always double-check that the data type inferred by VB Migration Partner is correct. Even better, we recommend that you use the InferType pragma only during early

migration attempts and then you later assign a specific type to members by editing the original VB6 code or by means of the SetType pragma.

タイプの推定は科学的に正確ではないので、VBMP によって推定されたデータタイプが正しいことを常に 2 重チェックするべきです。さらに良い方法として、私たちはマイグレーションの初期に InferType プラグマを使い、後で、元の VB6 ソースコードを編集するか SetType を使用してメンバーに特定のタイプを割り当てることを推奨します。

Unreachable code removal／Unreachable code（どこからも呼ばれないコード）の撤去

All versions of VB Migration Partner insert an upgrade comment at the beginning of an unreachable code region. Unreachable code is, for example, the code that follows a Return or Exit Sub statement:

VB Migration Partner の全てのバージョンは、Unreachable code（どこからも呼ばれないコード）範囲の前にアップグレードコメントを入れます。Unreachable code とは、例えば Return や Exit Sub ステートメントの後ろにあるコードのことです。

```
Public Sub Test(ByRef x As Integer, ByRef y As Integer)
    x = 123
    Exit Sub
    ' UPGRADE INFO (#0521): Unreachable code detected
    x = 456
    y = 789
End Sub
```

Users of VB Migration Partner can use the **RemoveUnrechableCode** pragma to either remove or comment out all the statements in the unreachable code block. This pragma takes one argument which can be equal to **Off**(do nothing, the default behavior), On (remove all statements in the unreachable block), or **Remarks** (comment out all statements in the unreachable block). For example, assuming that the previous code snippet is under the scope of the following pragma:

VB Migration Partner Enterprise edition の利用者は、**RemoveUnrechableCode** Pragma を使って Unreachable code ブロックの中にある全てのステートメントを撤去するか、コメントアウトするかのどちらかができます。この Pragma は下のような 1 つの引数を取ります。**OFF**（デフォルト値で何もしません。）ON（Unreachable ブロックの中の全てのステートメントを取り去ります。）**Remarks**（Unreachable ブロックの中の全てのステートメントをコメントアウトします。）先ほど挙げたコードの断片が次の Pragma のスコープ範囲にあると仮定してください。

```
'## RemoveUnreachableCode Remarks
```

then the code generated by VB Migration Partner would be as follows:

VB Migration Partner によって生成されたコードは次のようになります。

```
Public Sub Test(ByRef x As Integer)
    x = 123
    Exit Sub
```

```
        ' UPGRADE INFO (#06F1): Unreachable code removed ' EXCLUDED: x = 456
        ' EXCLUDED: y = 789
    End Sub
```

Unused members removal／未使用のメンバの撤去

All versions of VB Migration Partner insert an upgrade comment just before unused fields, constants, variables, properties, and methods. The actual message depends on whether the member is *never* used in the application or it is referenced by a method that, in turn, appears to be unused. The following piece of generated VB.NET code shows both kinds of messages:

VB Migration Partner の全てのバージョンは、未使用のフィールド、定数、変数、プロパティ、メソッドの直前にアップグレードコメントを入れます。実際のメッセージは、メンバがアプリケーションの中で使われていないかどうか、未使用と思われるメソッドに参照されているかどうかによって決まります。VB.NET が生成した下記のコードの断片は、上で述べた両方のメッセージの例を示しています。

```
    ' UPGRADE_INFO (#0501): the 'UnusedField' member isn't used anywhere in current application
    Public UnusedField As String
    ' UPGRADE_INFO (#0511): the 'UnusedConst' member is referenced only by members that
    ' haven't found to be used in current application
    Public Const UnusedConst As String = "Foobar"
```

VB Migration Partner comes with an extender that automatically removes unused constants and Declare statements.

VB Migration Partner は自動で未使用の定数と Declare ステートメントを取り去ります。

In addition, users of VB Migration Partner can use a **RemoveUnusedMembers** pragma to automatically remove (or remark out) all sorts of unused members. For example, if the previous code snippet were under the scope of the following pragma:

VB Migration Partner Enterprise edition の利用者は、**RemoveUnusedMembers** Pragma を使って全ての未使用のメンバを取り去ります。(または、コメントアウトします。) 先ほど挙げたコードの断片が次の Pragma のスコープ範囲にあると仮定すると、

```
    '## RemoveUnusedMembers Remarks
```

then the result would be as follows:

変換結果は次のようになります。

```
    ' UPGRADE_INFO (#0701): the 'UnusedField' member has been removed because
    ' it isn't used anywhere in current application
    Public UnusedField As String
```

```
    ' UPGRADE_INFO (#0711): the 'UnusedConst' member has been removed because
    ' it is referenced only by members that haven't found to be used in current
application
    Public Const UnusedConst As String = "Foobar"
```

Please notice that the RemoveUnusedMembers pragma used with the Remarks disables a few other refactoring features, for example the ConvertGosubs pragma. In other words, if the following pragmas are active:

RemoveUnusedMembers Pragma に Remarks を引数として使用した場合、他のいくつかのリファクタリング機能(例えば、ConvertGosubsPragma)が無効になることに注意してください。言い換えると、下記の Pragma がアクティブで有った場合、

```
'## RemoveUnusedMembers Remarks
'## ConvertGosubs
'## UseTryCatch
```

then VB Migration Partner generates (remarked out) nonoptimized VB.NET code, where Gosub keywords are not rendered as separate methods and On Error statements are not converted into Try-Catch blocks.

VB Migration Partner は最適化されていない VB.NET コードを生成します。生成されたコードは、GoSub キーワードが別のメソッドに変換されませんし、On Error Statements は Try-Catch ブロックに変換されません。

Important note:
the RemoveUnusedMembers pragma deletes or remarks all members that haven't found to be referenced by any other member in the current project or other projects in the current solution. However, this mechanism can lead to unwanted removal if the member is part of a public class that is accessed by another project (not in the solution) or if the member is accessed via late-binding. You can use
the **MarkAsReferenced** or**MarkPublicAsReferenced** pragmas to account for the first case.

RemoveUnusedMembers Pragma を使うと、現ソリューションの現プロジェクトや他のプロジェクトの中でどのメンバにも参照されていないように見える全てのメンバを消したりメンバの存在に気付くことができます。けれども、この手法を用いることで、メンバが(ソリューションではなく)他のプロジェクトからアクセスされている Public クラスに属している場合や、遅延バインディングでアクセスされているときに、望まない撤去が行われる可能性があるということです。一番目のケースを解明するのに **MarkAsReferenced** や **MarkPublicAsReferenced** Pragma を使うことができます。

However, there is no simple way to detect whether a member is accessed via late-binding. For this reason the RemoveUnusedMembers pragma supports a second *safeMode* parameter. If this parameter is True then the pragma affects only members that can't be reached via late-binding, e.g. constants, Declare statements, and members with a scope other than public.

メンバが遅延バインディングでアクセスされているかどうか検出する簡単な方法はありません。このため、RemoveUnusedMembers Pragma が 2 番目のセーフモードパラメータをサポートします。このパラメータが True ならば、Pragma は遅延バインディングで呼ばれていないメンバ(例えば、定数、Declare ステートメント、パブリック以外のスコープを持つメンバ)だけに影響します。

Renaming program members／プログラムのメンバの名前を変更する

VB Migration Partner supports enforcement of renaming guidelines by means of declarative rules that can be specified in an XML file. Thanks to this feature you can – for example – rename all push button controls so that their name begins with “btn” and possibly delete the “cmd” prefix that appeared in their VB6 name.

VB Migration Partner Enterprise edition は、XML ファイルの中で記載されている宣言の規則に基づいてリネームの指針が実行されるのをサポートします。これにより、“btn”で始まる全てのプッシュボタンコントロールのリネームが可能となり、VB6 で見られた“cmd”の接頭辞を消すことができます。

You enable the XML-based renaming engine by means of the **ApplyRenameRules** pragma. This pragma takes an optional argument that, if specified, must be equal to the path of the XML file that contains the renaming rules. If the argument is omitted, VB Migration Partner uses the RenameRules.xml file in its install folder. (It is recommended that you make a copy of this file and have the pragma point to the copy rather than the original file created by the setup procedure.) Here’s an example of this pragma:

ApplyRenameRules Pragma によって XML ベースのリネームエンジンを動かせます。この Pragma はオプションで 1 つの引数を持ち、リネーム規則を含む XML ファイルと同じパスに具体的に明記されています。引数をうっかり忘れると、VB Migration Partner はインストールフォルダに置かれた RenameRules.xml ファイルを使います。（このファイルをコピーして、setup プロシージャによって生成されたオリジナルファイルではなく、コピーしたファイルを Pragma に使わせることをお勧めします。） Pragma の例をご覧ください。

```
'## ApplyRenameRules c:¥apps¥myrules.xml
```

An important note: this pragma has an implicit project scope and affects all the projects in the project group being migrated. (This behavior is an exception to the usual scoping rules for pragmas.)

重要な注記：この Pragma は暗黙のプロジェクトスコープを持っていて、プロジェクトグループでマイグレーションされた全てのプロジェクトに影響を与えます。（この振る舞いは、Pragma の普通のスコープ規則に対する例外です。）

The RenameRules.xml file provided with VB Migration Partner includes all the common naming rules for controls and forms. Here’s an excerpt of the file:

VB Migration Partner が提供した RenameRules.xml ファイルには、コントロールとフォームに関する共通の全命名規則が書かれています。ファイルから抜粋したものを示します。

```
< ?xml version="1.0" encoding="utf-8" ?>
<NamingRules>
    < Symbol kind="Form">
        < Rule pattern="^frm" replace="" />
        < Rule pattern="(Form)?(? < num > ¥d*)$" replace="Form${num}" />
    < /Symbol>

    < Symbol kind="Component" type="VB.CommandButton" >
        < Rule pattern="^(cmd)?" replace="btn" />
```

```
            < Rule pattern="Button$" replace="" changeCase="pascal"
isFinal="true" />
        < /Symbol>

        < !-- more controls are dealt with here -->
    </NamingRules>
```

The **kind** attribute is used first to determine the kind of symbol the rule applies to. Valid names are Form, Class, Module, Type, UserControl, Enum, Component, Method (includes subs and functions), Field, Property, Event, EnumItem (the members of an Enum declaration), Constant, and Variable (includes, local variables and parameters.)

Kind 属性は、最初に、規則を当てはまるシンボルの種類を決定するのに使用されます。有効な名前としては、Form、Class、Module、Type、ユーザーコントロール、Enum、コンポーネント、メソッド（Subs と Functions を含みます。）、フィールド、プロパティ、イベント、EnumItem（Enum 宣言のメンバ）、定数と変数（ローカル変数とパラメータを含む）があります。

If kind is equal to Component or Variable, you can optionally include a **type** attribute, that specifies the VB6 type of the member to which the rule applies. For example, the following rule renames all Integer variables named “i” into “index”

もし kind がコンポーネントや変数なら、任意で **Type** 属性を含めることができ、ルールを適用したい VB6 型のメンバの指定ができます。下記の例では、“i”として定義された Integer 型の変数を“index”にリネームしています。

```
    < Symbol kind="Variable" type="Integer" >
        < Rule pattern="^i$" replace="index" ignoreCase="false" />
    </Symbol>
```

Renaming rules are based on regular expressions. The **pattern** attribute is used to decide whether the rule applies to a given member, and the **replace** attribute specifies how the member should be renamed.
The**ignoreCase** optional attribute should be false if you want the pattern to be applied in case-sensitive mode (by default comparisons are case-insensitive). The **changeCase** optional attribute allows you to change the case of the result and can be equal to “lower”, “upper”, “pascal” (first char is uppercase), or “camel” (first char is lowercase).

リネーム規則は正規表現に基づいています。**pattern** 属性は規則が特定のメンバに適用されるかどうか決めるのに使用され、**replace** 属性はメンバがどう改名されるべきであるかを指定します。大文字と小文字を区別してパターンを適用したい場合は、**ignoreCase** オプション属性を false に設定してください。（デフォルトでは、大文字と小文字を区別しません） **changeCase** オプション属性は、結果の活字ケースを“lower”, “upper”, “pascal”, “camel”と同等に変更することが可能です。

Being Visual Basic a case-insensitive language, in most cases the default behavior is fine. However, consider the following renaming requirement: many VB6 developers preferred to use “C” as a prefix for all class names, but this guideline has been deprecated in VB.NET. Here’s a rule that drops the leading “C”, but only if it is followed by another uppercase character:

Visual Basic は大文字と小文字を区別しない言語なので、たいていはデフォルトの振る舞いで問題ありません。しかしながら、次のようなリネームが必要なものについて考えてみてください。多くの VB6 開発者が、すべてのクラス名に接頭語として「C」を使用するのを好みましが、この方針は VB.NET では推奨されません。ここに、それが別の大文字キャラクタが後に続いている場合にだけ、先頭の「C」を削除するルールがあります。

```
< Symbol kind="Class">
    < Rule pattern="^C(?=[A-Z])" replace="" ignoreCase="false" />
</Symbol>
```

The changeCase attribute is useful to enforce naming rules that involve the case of identifiers. For example, a few developers like all-uppercase constant names; there a rule that enforce this guideline:

changeCase 属性は識別子を含んだときのネーミング規則を強化するのに役立ちます。例えば、定数の名前を全て大文字にする開発者が中にはいますが、この方針を強化するのに次のような規則があります。

```
< Symbol kind="Constant" >
    < Rule pattern="^.+$" replace="${0}" changeCase="upper" />
</Symbol>
```

You can restrict a rule to one or more project items by means of the **projectItem** optional attribute. This attribute is considered as a regular expression applied to the name of the parent project item where the symbol is defined. (The project item name is in the form “projectname.itemname”. For example, let’s say that you want to drop any “str” prefix that appears in string variables defined inside the frmMain form and in Helpers module of the TestApp project:

projectItem オプション属性によって、変換ルールを複数のプロジェクトに制限することが出来ます。この属性はシンボルが定義された親プロジェクトに適用された正規表現としてみなされます。フォームの中に“projectname.itemname”というプロジェクトアイテム名があります。例えば、TestApp プロジェクトの frmMain と Helpers モジュール内で定義された、文字変数の str 接頭語を削除したい場合、

```
    < Symbol kind="Variable" type="String"
projectItem="^TestApp¥.(frmForm|Helpers)$" >
        < Rule pattern="^str" replace="" />
    < /Symbol>
```

It is essential that the projectItem regular expression accounts for the project name; if you want to apply the rename rule to any project – as it’s often the case – you must specify the .+ regular expression for the project name part. For example, the following rule changes the name of the ChangeID method in the Helpers module in all converted projects

プロジェクト名を表す、projectItem 正規表現は不可欠です。よくあるケースですが、リネーム規則を全てのプロジェクトに当てはめたい場合、正規表現のプロジェクト名パートに” .+”を指定する必要があります。例えば以下のルールでは、全ての変換されたプロジェクトの中の、Helpers モジュールの中の ChangeID メソッドの名前を変更します。

```
< Symbol kind="Method" projectItem="^.+¥.Helpers$" >
    < Rule pattern="^ChangeID" replace="ChangeIdentifier" />
```

```
</Symbol>
```

The projectItem attribute is especially useful if your VB6 adopted the naming convention according to which all form names begin with “frm”, all classes with “cls” and so forth. In such a case, in fact, you can easily restrict a rule by relying on these prefixes:

projectItem 属性は、全てのフォーム名が“frm”で始まったり、全てのクラス名が“cls”で始まったりなど、VB6 のネーミング変換の際に特に役立ちます。このような場合、これらの接頭語を使うことによって、簡単に規則を制限することができます。

```
<!-- convert x,y into xPoint and yPoint, but only inside classes -->
<Symbol kind="Variable"  type="Double" projectItem="^cls" >
    < Rule pattern="^(x|y)$"  replace="${0}Point" />
</Symbol>
```

By default, <Symbol> tags that have a non-empty projectItem attribute are evaluated before rules where this attribute is omitted. All the <Rule> tags inside each block are evaluated in order, from the first one to the last one. However, if a rule has the **isFinal** attribute set to “true” then all the rules that follow are ignored.

デフォルトでは、空でない projectItem 属性を持っている <Symbol> タグは、この属性が省略されたルールの前に判断されます。各々のブロックの中にある全ての <Rule> タグは、最初から最後まで順番に判断されます。ただし、**isFinal** 属性が“true”に設定された変換ルールでは、全てのルールは以後無効となります。

Important note:
VB Migration Partner renaming engine doesn’t check whether the new name assigned to symbols is a valid VB.NET identifier and doesn’t check whether the new name is unique. It’s up to the developer ensuring that the generated VB.NET code doesn’t include invalid or duplicated identifier names. In practice you’ll notice this kind of errors as soon as you compile the generated VB.NET, therefore we don’t consider it a serious issue.

VB Migration Partner リネーミングエンジンは、シンボルに割り当てられた新しい名前が VB.NET のものと比べて妥当であるかどうかをチェックしませんし、新しい名前が唯一の名前かどうかもチェックしません。生成された VB.NET コードが無効でまったく同じ名前を含んでいないかどうかを確認するのは開発者の役目です。実際は、生成された VB.NET コードをコンパイルした際にすぐにこの種のエラーに気付きます。なので、それほど深刻に考えなくても大丈夫です。

When applying this pragma, however, you should be aware that there is margin for other issues that do *not*cause a compilation error and that may go unnoticed if you don’t perform accurate tests on the generated VB.NET application. Consider the following VB6 code:

しかし、この Pragma を適用する際に、ご承知のとおり、生成された VB.NET アプリケーションに対し緻密なテストを実行しないなら、コンパイルエラーではない別の問題が潜んでいる余地があることに十分に配慮してください。以下の VB6 コードを参照下さい。

```
Const Name As String  = "Code Architects"
```

```
    Public Sub Test()
        Const strName As String = "NAME"
        Debug.Print strName & " = " & Name    ' displays "NAME = Code Architects"
「NAME = Code Architects」と示されます。
    End Sub
```

Next, let's assume that you apply a rename rule that prefixes all string constants with the "str" prefix, if the prefix is missing. This causes the first constant to be assigned the same name of the local constant. The neat effect is that the local constant shadows the first constant and that the method displays a different string:

次に接頭語がない文字定数の前に、"str"を接頭語として置くようなリネームルールを適用したと仮定してください。それは、最初の定数に、ローカル変数と同じ名前を割り当てることを引き起こします。適切な対処をするには、ローカル変数が最初の変数を示し、メソッドが別の string を示すようにすることです。

```
    Const strName As String  = "Code Architects"

    Public Sub Test()
        Const strName As String = "NAME"
        Debug.Print strName & " = " & strName     ' displays "NAME = NAME""NAME =
NAME"と示されます。
    End Sub
```

This sort of issues are quite subtle because they don't cause any compilation error. The current version of VB Migration Partner when a rename rule causes this problem, therefore it's your responsibility ensuring that functional equivalence isn't compromised by rename rules.

この種類の問題は、コンパイルエラーの原因とはならないため、かなりとらえにくい問題です。現在の VB Migration Partner のバージョンで、リネームルールがこの問題を引き起こした時、機能等価がリネームルールにより悪い結果にならないように保証するのは、お客様側の責任になります。

4.4 Extenders／機能拡張

Developers can write extenders to receive notifications from VB Migration Partner during the parse and code generation process. An extender class can access VB Migration Partner's object model, read pragmas (and dynamically add new ones), edit the VB6 code before it is being parsed, modify the VB.NET code after it has been generated, and so forth.

開発者は構文解析プログラムやコード生成プロセスが動いている間、VB Migration Partner から通知を受け取って Extender を書くことができます。Extender クラスは VB Migration Partner のオブジェクトモデルにアクセスして Pragma を読んだり、新しい Pragma を追加します。そして、構文解析プログラムが動き出す前に VB6 コードを編集し、生成された VB.NET コードを修正します。

The following extender class inserts a remark at the top of all VB.NET code files produced during the migration process:

次の Extender クラスはマイグレーションプロセスで生成された VB.NET コードファイルの先頭に注記として挿入されます。

```
' this attribute marks the current DLL as an extender
この属性は現在の DLL を拡張としてマークします

<Assembly: CodeArchitects.VB6Library.VB6SupportLibrary(True)>

<VBMigrationExtender("My first extender", _
    "A demo extender for CodeArchitect's VB Migration Partner")> _
Public Class DemoExtender
    Implements IVBMigrationExtender

    Public Sub Connect(ByVal data As ExtenderData) _
            Implements IVBMigrationExtender.Connect
        ' do nothing at connect time
        接続時間で何もしません。

    End Sub

    Sub Notify(ByVal data As ExtenderData, ByVal info As _
            OperationInfo) Implements IVBMigrationExtender.Notify
        If info.Operation = OperationType.ProjectItemConverted Then
            Dim remark As String = "Generated by VB Migration Partner" & vbCrLf

            info.ProjectItem.NetCodeText = _
                remark & info.ProjectItem.NetCodeText
        End If
    End Sub
End Class
```

All pragmas are visible to extenders. Even better, an extender can programmatically create new instances of pragmas and associate them with code entities, as if they were embedded in the VB6 code being migrated. This feature enables developers to overcome that static nature of pragmas hard-coded in VB6 code. For example, the decision to treat a variable as an auto-instancing variable might be taken after checking how the variable is used, as in this code:

全ての Pragma は Extender に見えます。さらによいことは、まるで VB6 のコードがマイグレーションされたかのように、Extender は Pragma の新しいインスタンスを生成でき、それらをコードの実体と関連づけます。この特性は、開発者にとって、Pragma の静的な性質が VB6 で難しかったコーディングに勝ることを可能にします。例えば、自動イン

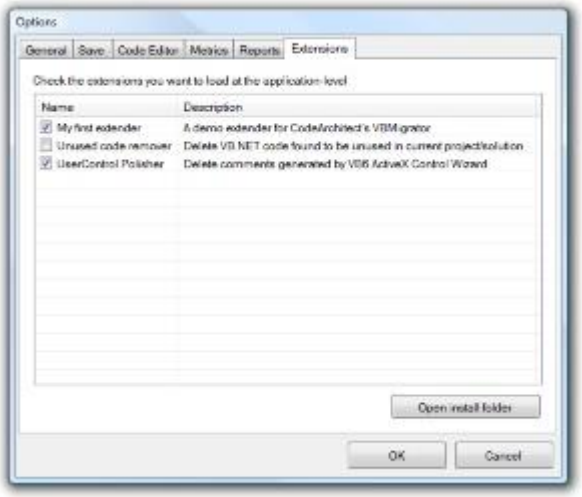
スタンス変数として変数を扱う決定は、次に示すコードのように、変数がどのように使われているかをチェックした後でなされます。

```
' (this code runs inside an extender DLL)
If someCondition Then
    ' treat the "cn" field as a true auto-instancing variable

    Dim cnSymb As VBSymbol = info.ProjectItem.Symbol.Symbols("cn")
    Dim pragma As New VBPragma("AutoNew True", Nothing)
    cnSymb.Pragmas.Add(pragma)
End If
```

For VB Migration Partner to recognize an extender DLL, developers need to deploy the DLL in VB Migration Partner's main directory. The interactive user can then enable and disable extensions from the Extensions tab of the Tools-Options dialog box.

VB Migration Partner が extender DLL を認識するためには、開発者は VB Migration Partner のメインディレクトリに DLL を置く必要があります。Tools → Options の順番にクリックすると現れるダイアログの中にある Extensions タブで、Extension を有効にしたり、無効にしたりすることができます。



The sample extenders that are provided with VB Migration Partner perform quite generic task, for example removing unused Const and Declare statements or polishing the code of a VB6 user control class after the conversion to VB.NET.

VB Migration Partner から提供された Extender のサンプルは、かなり包括的な仕事をします。例えば、未使用の定数や宣言のステートメントを削除したり、VB.NET に変換された後のユーザコントロールクラスの VB6 コードを洗練します。

However, the main purpose of extenders is to customize VB Migration Partner's behavior and output for specific projects and coding styles. For example, a software company might decide to create a base class for all their forms and therefore all the forms in the VB.NET application should derive from this base class rather than

CodeArchitects.VB6Library.VB6Form. Implementing an extender that changes the base class for all forms is quite simple:

けれども、Extender の主な目的は、VB Migration Partner の振る舞いをカスタマイズして、一定のプロジェクトやコーディング形式を出力することです。ソフトウェア会社は全ての Form の基になるクラスを生成しようと試み、そのような訳で VB.NET アプリケーションの全ての Form は CodeArchitects.VB6Library.VB6Form よりこの基になるクラスから派生すべきです。全フォームがかなり単純となるように、基になるクラスを変更する Extender を実行すると下記のようになります。

```
    Sub Notify(ByVal data As ExtenderData, ByVal info As OperationInfo) _
            Implements IVBMigrationExtender.Notify
        If info.Operation = OperationType.ProjectItemConverted AndAlso
            info.ProjectItem.Symbol.Kind = SymbolKind.Form Then
            ' VB Migration Partner has completed the conversion of a form
            info.ProjectItem.NetDesignerText =
info.ProjectItem.NetDesignerText.Replace( _
                "Inherits System.Windows.Forms.Form", "Inherits
CompanyName.CompanyFormBase")
        End If
    End Sub
```

(Notice that the CompanyName.CompanyFormBase class must inherit from CodeArchitects.VB6Library.VB6Form for the converted application to work correctly.)

（CompanyName.CompanyFormBase クラスは変換されたアプリケーションが正しく動くために CodeArchitects.VB6Library.VB6Form を継承しなければなりません。）４．５　3rd パーティ ActiveX コントロールのサポートについて

4.5 Support for 3rd-party ActiveX controls／サードパーティ ActiveX コントロール対応

VB Migration Partner supports all the controls included in the Visual Basic 6 package, with the only exception of the OLE and Repeater controls. When migrating a form that contains unrecognized controls, such controls are transformed into “placeholder” controls that stand out on the form because of their red background.

VB Migration Partner は Visual Basic 6 のパッケージに含まれる全てのコントロールをサポートします。ただし、OLE と Repeater コントロールは除きます。認識されないコントロールを含む Form をマイグレーションすると、そのようなコントロールは赤い背景色をしているために Form の上で目立つ“プレースホルダー”コントロールに変換されます。

Once you have a .NET control that mimics or wraps the original ActiveX control, you just need to copy the .NET DLL inside VB Migration Partner’s main directory; no registration is required. The next time you run VB Migration Partner, the new control is recognized as if it were one of the VB6 built-in controls.

Alternatively, you can copy it to a folder and then use an **AddLibraryPath** pragma to have VB Migration Partner recognize all the DLLs in that folder:

独自の ActiveX コントロールに似ているか、またはラップした.NET コントロールが存在した場合、それらは VB Migration Partner のメインディレクトリの中で.NET DLL にコピーされます。レジストリ登録は必要ありません。VB Migration Partner を次回起動すると、新しいコントロールがまるで VB6 で組み込まれているコントロールであるかのように認識されます。あるいは、それをフォルダにコピーし、**AddLibraryPath**Pragma を使うことで、VB Migration Partner がフォルダの中の全ての DLL を認識できるようにすることができます。

```
'## AddLibraryPath "c:¥mydlls"
```

VB Migration Partner can handle additional ActiveX controls, in addition to those that are supported out of the box. The exact sequence of actions that is necessary to implement such support depends on the type of the control you want to support. Three options are available, listed here in order of increased difficulty:

VB Migration Partner はもともとサポートされている ActiveX に加えて、追加した ActiveX コントロールにも対応可能です。そのようなサポートを実行するのに必要な正しい順序は、サポートしたいコントロールのタイプに依存します。3 つのオプションを使うことができますが、後に記されているほど実現困難となります。

a. Use VB Migration Partner to migrate it to VB.NET. (This option requires that the control is authored in VB6 and you have its source code.)
マイグレーションするために VB Migration Partner を使用して、VB.NET に変換します。（このオプションではコントロールが VB6 で作成され、そのソースコードが存在する必要があります。）
b. Create a managed wrapper around the original ActiveX control. VB Migration Partner’s package includes the AxWrapperGen utility that automates this process.
独自の ActiveX コントロールから管理されたラッパーを生成します。VB Migration Partner のパッケージにはこのプロセスを自動で行う AxWrapperGen ユーティリティが含まれています。
c. Create a fully managed control that behaves like the original control.
オリジナルのコントロールと同じように動作する、完全に管理されたコントロールを作成する。

For completeness’s sake, when comparing all the available solutions you should also consider a fourth solution strategy:

また、万全を期するのであれば、利用可能な全ての解決策を比べたうえで、さらに 4 番目の解決方法を考慮するべきです。

d. Migrate the application as usual, then manually replace the “placeholder” control with a .NET control and then manually fix all the references to the control’s properties and methods.
アプリケーションを通常変換して下さい。それから、手動で“プレースホルダー”コントロールを.NET コントロールに置き換え、さらに手動でコントロールのプロパティとメソッドの参照を調整します。

If you have the VB6 source code of the ActiveX control it is recommended that you follow the a) strategy. If source code isn’t available, the choice between remaining strategies depends on a number of considerations:

ActiveX コントロールの VB6 ソースコードがあるなら、上記 a)を行うことをおすすめします。ソースコードが利用できないのであれば、いくつかのことを考慮する必要があります。

- A VB.NET that uses wrappers for ActiveX controls can't benefit from some advantages of the .NET Framework platform, for example XCOPY deployment and side-by-side execution of different versions of the same component.
  ActiveX コントロールにラッパーを使う VB.NET は、.NET　Framework から恩恵を受けられません。例えば、XCOPY デプロイメント、同じコンポーネントの異なる Verison の side-by-side 実行など。
- The COM Interop mechanism isn't perfect and we have noticed that a few complex controls – most notably, the DataGrid control – occasionally crashes when placed on a .NET form. You should carefully test all forms that host one or more ActiveX controls, either directly or wrapped by a .NET class.
  COM　Interop メカニズムは完全ではないので、特に DataGrid コントロールなどのいくつかの複雑なコントロールは、.NET の Form に配置されたときに壊れることがあります。.NET クラスによるラップあるいは直接配置された 1 つ以上の ActiveX コントロールをもったすべての Form について、注意深く動作確認すべきです。
- If the ActiveX control is used massively by the project being migrated – or in other projects that you plan to migrate in the near future – the time required to create a fully managed control that behaves like the original control can pay off nicely.
  ActiveX コントロールがマイグレーションされたプロジェクト、または近い将来マイグレーションしようとしている他のプロジェクトでたくさん使われている場合、オリジナルのコントロールのように動作する完全に管理されたコントロールを作成する時間が要求されます。
- If the application being migrated sparingly uses the control – for example it appears only in one or two forms – running VB Migration Partner and then manually fixing all references is possibly the most cost effective strategy
  マイグレーションされたアプリケーションが 1 つか 2 つの Form の中でだけに貼られているコントロールを使う場合、VB Migration Partner を活用し、全ての参照を手動で修正していくことが、一番費用対効果の高いやりかたです。

The remainder of this section explains how to implement strategy a) and b).

このセクションの残りでは、方針 a）と　b）の実行方法を説明します。

Note:

Details about strategy c) will be made available when the release version is launched.

方針 c）についての詳細はリリースバージョンが起動されたときに利用できるようになります。

Migrating ActiveX Controls authored in VB6／VB6 で署名された ActiveX コントロールの移行

If the control or component is authored with VB6 and you have its source code, in most cases adding support for that control or component is a simple process: just run VB Migration Partner to migrate the ActiveX DLL or

ActiveX Control project that contains the user control, going through the usual convert-test-fix cycle. When the migrated project finally works correctly, you might want to polish the VB.NET code.

コントロールやコンポーネントが VB6 で作られていて、かつそのソースコードがあるとしたら、多くの場合そのコントロールやコンポーネントに対するサポートを追加することは簡単な工程となります。ユーザコントロールを含む ActiveX DLL か ActiveX Control プロジェクトを移行のために VB Migration Partner をただ利用し、通常の「convert-test-fix」サイクルを通して変換するシンプルな工程です。プロジェクトのマイグレーションが終わったときには、VB.NET のコードは洗練され、正確に動作していることでしょう。

For example, most VB6 user controls have been created by means of the ActiveX Control Interface Wizard, which generates tons of VB6 code whose only purpose is remedying the lack of inheritance in VB6. For example, assume that you are migrating a VB6 user control that works like an enhanced TextBox control. Such a control surely includes a Text property that does nothing but wrapping the property with same name of the inner TextBox control:

例えば、ほとんどの VB6 ユーザコントロールは ActiveX Control Interface Wizard によって生成され、VB6 において欠如された継承を改善することのみを目的としたたくさんの VB6 コードを生成しています。例えば、テキストボックスの機能を拡張した、VB6 のユーザコントロールをマイグレーションしていると仮定してください。そのようなコントロールは、TextBox コントロール内にもつ同じ名前のプロパティをラッピングしただけの Text プロパティが含まれます。

```
Public Property Get Text() As String
    Text = Text1.Text
End Property

Public Property Let Text(ByVal New_Text As String)
    Text1.Text() = New_Text
    PropertyChanged "Text"
End Property
```

This property is translated correctly to VB.NET, however it is unnecessary and could be dropped without any negative impact on the migrated project. The simplest way to fix this code is by means of a couple of OutputMode pragmas:

このプロパティは正確に VB.NET に変換されましたが、マイグレーションされたプロジェクトには必要でなく、どんな悪い影響もありません。このコードを修正する最も簡単な方法は、二つの対になる OutputMode Pragma を使うことです。

```
'## OutputMode Off
Public Property Get Text() As String
    Text = Text1.Text
End Property

Public Property Let Text(ByVal New_Text As String)
    Text1.Text() = New_Text
    PropertyChanged "Text"
```

```
End Property

'## OutputMode On
```

Also, you can use PostProcess pragmas to delete special remarks added by the ActiveX Control Interface Wizard, as well as calls to the PropertyChanged method (which are useless under VB.NET).

(VB.NET では使い物にならない)PropertyChanged メソッドの呼び出しと同じように、ActiveX Control Interface Wizard によって付け加えられた特別な備考を消す PostProcess Pragma を使うことができます。

Running the AxWrapperGen tool／AxWrapperGen ツールを実行する

AxWrapperGen is a tool that is part of the VB Migration Partner package. (You can find it in the main installation directory, which by default is C:¥Program Files¥Code Architects¥VB Migration Partner.) It is a command-line utility, therefore you must open a command window and run AxWrapperGen from there.

AxWrapperGen は VB Migration Partner パッケージに含まれているツールです。(AxWrapperGen は VB Migration Partner のメインインストールディレクトリに存在しており、デフォルトのフォルダは、C:¥Program Files¥Code Architects¥VB Migration Partner です。)このツールはコマンドラインユーティリティーなので、コマンドプロンプト上で動かします。

AxWrapperGen's main purpose is allowing VB Migration Partner to support compiled 3rd-party ActiveX controls. AxWrapperGen takes one mandatory argument, that is, the path of the .ocx file that contains the ActiveX control to be wrapped. For example, the following command creates the wrapper class for the Microsoft Calendar control (and of course assumes that such control is installed in the c:¥windows¥system32 folder):

AxWrapperGen の主な目的は、VB Migration Partner に3rd パーティ ActiveX コントロールのサポートをさせることです。AxWrapperGen は 1 つの必須の引数をとり、その引数とはラップされた ActiveX コントロールを含む ocx ファイルのパスです。下記のコマンドは Microsoft カレンダーコントロール(このコントロールはご存知の通り c:¥windows¥system32 フォルダにインストールされています。)のラッパークラスを生成します。

```
AxWrapperGen  c:¥windows¥system32¥mscal.ocx
```

(Notice that we are using the MSCAL.OCX control only for illustration purposes, because VB Migration Partner already supports this control.) If the file name contains spaces, you must enclose the name inside double quotes. AxWrapperGen is able to convert multiple ActiveX controls in one shot

(MSCAL.OCX コントロールを実例として挙げたのは、VB Migration Partner が既にこのコントロールをサポートしていることを覚えておいてください。)ファイル名にスペースを含んでいるならば、ダブルコーテーションでファイル名を囲まなければなりません。AxWrapperGen は多様な ActiveX コントロールを 1 発で変換することができます。

```
AxWrapperGen
c:¥windows¥system32¥mscal.ocx   "c:¥windows¥system32¥threed32.ocx"
```

By default, AxWrapperGen generates a new project and solution named **AxWrapper** in current directory. You can change these default by means of the **/out** switch (to indicate an alternative output directory) and **/project**switch (to set the name of the new project and solution):

デフォルトで、AxWrapperGen は現在のディレクトリに **AxWrapper** と名付けられた新しいプロジェクトとソリューションを生成します。オプションに**/out**(他の出力ディレクトリを指定します。)や**/project**(新しいプロジェクト名やソリューション名を設定します。)を指定することによってデフォルトの設定を変えることが可能です。

```
AxWrapperGen  c:¥windows¥system32¥mscal.ocx  /out:c:¥myapp
/project:NetCalendar
```

By default, AxWrapperGen generates VS2008 projects. You can generate VS2010 projects by adding a **/version**option:

デフォルトで、AxWrapperGen は VS2008 プロジェクトを生成します。**/version** オプションを指定すると、VS2005 プロジェクトを生成することができます。

```
AxWrapperGen c:¥windows¥system32¥mscal.ocx /version:2010
```

or you can generate VS2010 projects that target .NET Framework 4.0 with this command:

```
AxWrapperGen c:¥windows¥system32¥mscal.ocx /version:2010_40
```

AxWrapperGen runs the AxImp tool – which is part of the .NET Framework SDK – behind the scenes, to generate two DLLs that work as the RCW (Runtime Component Wrapper) for the selected control. For example, the mscal.ocx control generates the following two files: msacal.dll and axmsacal.dll. You can specify the following five options, which AxWrapperGen passes to AxImp: /keyfile, /keycontainer, /publickey, /delaysign, and /source. For more information, read .NET Framework SDK documentation.

AxWrapperGen は AxImp ツール上で動きます。AxImp ツールは.NET Framework SDK に含まれており、背後で、選択されたコントロールに RCW(Runtime Component Wrapper)として働く 2 つの DLL を生成します。例えば、mscal.ocx コントロールは msacal.dll と axmsacal.dll.という 2 つのファイルを生成します。また、/keyfile、/keycontainer、/publickey、/delaysign、/source という AxWrapperGen が AxImp に渡す 5 つのオプションがあります。もっと情報が必要であれば、.NET Framework SDK ドキュメントをお読みください。

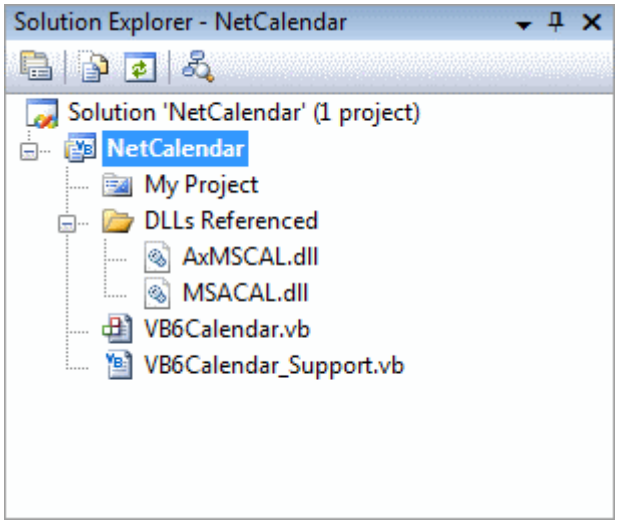
At the end of generation process, AxWrapperGen runs Microsoft Visual Studio and loads a solution that contains one VB.NET class library (DLL) project, containing a pair of classes for each ActiveX control:

生成プロセスの最後で、AxWrapperGen は Microsoft Visual Studio を起動し、1 つの VB.NET クラスライブラリ(DLL)プロジェクトを含むソリューションをロードします。また、それぞれの ActiveX コントロールに対して 1 組のクラスを含んでいます。

- The first class of each pair works as a wrapper for the original ActiveX control.
  1 組のクラスのうち最初のクラスは、オリジナルの ActiveX コントロールのラッパーとして働きます。
- The second class of each pair provides support during the migration process.
  1 組のクラスの 2 番目のクラスは、マイグレーション中にサポートします。

In our example, the MSCAL.OCX file contains only one ActiveX control, therefore only two classes will be created: the class named VB6Calendar inherits from AxMSACAL.AxCalendar and wraps the ActiveX control; the second class is named VB6Calendar_Support And inherits from VB6ControlSupportBase. This class will be instantiated and used by VB Migration Partner during the migration process, but not at runtime during execution.

挙げた例では、MSCAL.OCX ファイルはたった 1 つの ActiveX コントロールを含みます、その場合 2 つのクラスが生成されます。VB6Calendar と名付けられたクラスは AxMSACAL.AxCalendar から継承し、ActiveX コントロールをラップします。2 番目のクラスは VB6Calendar_Support と命名され、VB6ControlSupportBase から継承します。このクラスはインスタンス化されマイグレーションプロセス中は VB Migration Partner によって使用されますが、実行中は使用されません。



Before compiling the solution, check the "Instructions" region at the top of each class. In most cases, the project that AxWrapperGen utility creates compiles correctly at the first attempt. The compilation process creates a DLL that you must manually copy into the VB Migration Partner's main directory.

ソリューションをコンパイルする前に、それぞれのクラスの一番上にある「Instructions」領域をチェックしてください。多くの場合、AxWrapperGen ユーティリティが生成したプロジェクトは、初めてのコンパイル時は正確にコンパイルします。コンパイルプロセスは手動で VB Migration Partner のメインディレクトリにコピーしなければならない DLL を生成します。

In some cases, however, you need to tweak the code that has been created. More precisely, if any property or method has to do with graphic units – in other words, it represents a width or height and is therefore affected by the container's ScaleMode property – you need to shadow the original member and create your own. For example, the DataGrid control exposes a property named RowHeight, which requires this treatment. This is the code that implements the required fix:

いくつかのケースでは、生成されたコードを手直しする必要があります。もっと正確に言うと、プロパティやメソッドがグラフィックユニットに対してすべきことがある場合、他の言葉で言うと、width や height がコンテナの ScaleMode プロパティに影響されている場合、オリジナルのメンバを遮断し新しいメンバを生成する必要があります。例えば、RowHeight という名前のプロパティを持つ DataGrid コントロールは、この処理が必要です。下記は必要な修正を実装するコードです。

```
<Localizable(True)> _
<Category("Appearance")> _
Public Shadows Property RowHeight() As Single
    Get
        Return VB6Utils.FromPixelY(Me, CInt(MyBase.RowHeight), False, True)
    End Get

    Set(ByVal value As Single)
        MyBase.RowHeight = VB6Utils.ToPixelY(Me, value, False, True)
    End Set
End Property
```

The bulk of the work is done inside the **FromPixelY** and **ToPixelY** methods of the VB6Utils class, contained in the CodeArchitects.VBLibrary.dll. These methods check the current ScaleMode value and perform all the required conversions. (Of course, the class exposes also the **FromPixelX** and **ToPixelX** methods, for measures along the X axis.)

この処理の大部分は CodeArchitects.VBLibrary.dll に含まれる VB6Utils クラスの **FromPixelY** および **ToPixelY** メソッドで行われます。これらのメソッドは、現在の ScaleMode 値をチェックし、全ての必要とされる変換を行います。（もちろん、クラスにも X 軸を測る **FromPixelX** と **ToPixelX** メソッドが存在します。）

4.6 Using the VBMP command-line tool／VBMP のコマンドラインツールを利用する

VB Migration Partner supports automated conversions by means of the VBMP.EXE command-line too. You can use this tool inside batch files and in MSBuild projects. Its syntax is quite simple:

VB Migration Partner は VBMP.EXE コマンドラインによる自動変換をサポートします。このツールはバッチファイルや MSBuild プロジェクトで使用することができます。その構文は非常に簡単です。

```
VBMP projectfile  options
```

Where *projectfile* is the complete path of the VB6 project (or project group) file, and *options* is zero or more of the following:

projectfile が VB6 プロジェクト（あるいはプロジェクトグループ）の絶対パスの場合は、options は 0 もしくは下記のようになります。

/out:path
specifies the parent directory of the folder that will be created for the VB.NET solution. This option corresponds to the path you can enter in the Save tab of the Tools-Options dialog box.)

VB.NET ソリューションの為に作成されるフォルダの、親ディレクトリを指定します。(このオプションは、[Tools] ー [Options]ダイアログボックスの[Save]タブで入力したパスに一致します。)

/version:VVVV

specifies the Visual Basic version of the code to be output. VVVV can be either 2005, 2008, 2010 or 2010_40. (The default is 2008). The 2010_40 settings generates projects that target .NET Framework 4.0, all other settings generate projects that target .NET Framework 2.0/3.5.

出力する Visual Basic のコードのバージョンを指定します。VVVV は 2005 もしくは 2008 のどちらかです。(デフォルトは 2005)

/maxissues:NN

specifies the max number of conversions issues. If the conversion process generates more issues than this value, the VBMP utility exits with a non-zero errorlevel and doesn't proceed with the compilation step even if the /compile option is defined. (If omitted, this value is equal to 10.)

変換で出た問題の最大数を指定します。変換プロセスがこの値以上の問題を生成した場合、VBMP ユーティリティは非ゼロのエラーレベルで終了し、/compile オプションが定義されていたとしても変換は続行されません。(入力を省略した場合、値は 10 です。)

/compile

causes the converted VB.NET to be compiled.

変換した VB.NET をコンパイルします。

/clearcache

clears the typelib cache.

タイプライブラリのキャッシュをクリアします。

/quiet

suppresses most messages.

メッセージを制限します。

/help or **/h**

displays a brief explanation of each option.

各々のオプションの簡単な説明を示します。

For example, the following command compiles the c:¥vb6¥widgets.vbg project and creates a VB2010 solution in the c:¥vbnet¥widgets folder, then compiles the result unless the conversion process generated more than 30 migration issues:

例えば、次のコマンドは c:¥vb6 に存在する widgets.vbg プロジェクトをコンパイルして、c:¥vbnet¥widgets フォルダに VB2008 ソリューションを生成し、30 以上のマイグレーションに関する問題が出力されなければ、ソリューションをコンパイルします。

```
VBMP c:¥vb6¥widgets.vbg  /out:c:¥vbnet  /compile /maxissues:30
/version:2010
```

If the specified VB6 project can't be converted, or the conversion was aborted because of a fatal error, or the conversion produced more issues than allowed, or if the compilation step produced one or more errors, then VBMP returns a nonzero errorlevel to the operating system.

指定された VB6 プロジェクトが変換されなかったり、致命的なエラーによって変換が失敗したり、変換によってより問題が増えたり、変換中 1 つ以上の解決困難な問題が発生した場合、VBMP はオペレーティングシステムに非ゼロの値を返します。

4.7 The VB Project Dumper add-in／VB プロジェクトのダンパーアドイン

VB Migration Partner converts all the properties assigned at design-time and whose value is stored in the hidden portion of .frm forms. However, in very few cases, the information stored in these .frm is encoded in a format that we didn't manage to decode.

VB Migration Partner はデザイン時に割り付けられた全てのプロパティを変換し、それらの値は拡張子が.frm である Form の隠れた部分の中に格納されます。これらの.frm に格納された情報は、デコードできない形式でエンコードされることはまれです。

For example, we haven't found a way to decode how properties of the MSChart control are stored in the .frm form. To correctly migrate this control you should load the original project in the VB6 IDE and run the VBMP Project Dumper add-in.

例えば我々は、拡張子が.frm の Form に格納された MSChart コントロールのプロパティをデコードする方法を知りません。このコントロールを正しくマイグレーションするためには、VB6 IDE の独自のプロジェクトをロードし、VBMP プロジェクトのダンパーアドインとして動かすべきです。

The VBMP Project Dumper is a VB6 add-in that adds a new command to the Add-ins menu, which – when invoked – creates an XML file containing all the properties of all the controls in the project. You need to invoke the command just once, before attempting the migration process. (You need to invoke it again only if you modify one or more properties of an MSChart control in the VB6 project or if accidentally delete the XML file.)

VBMP プロジェクトのダンパーは、アドインメニューに新しいコマンドを加える VB6 のアドインで、起動時にはプロジェクトの中に含まれる全てのコントロールの全プロパティを含む XML ファイルを生成します。マイグレーションプロセスを試みる前に、一度だけ起動すればよいです。(VB6 プロジェクトの中の MSChart コントロールのプロパティを一度以上修正したり、間違って XML ファイルを消してしまった時は、再度起動する必要があります。)

If you migrate a form containing an MSChart control or another control that may require decoding – for example the DataTimePicker control – VB Migration Partner searches for this XML file and issues a migration note asking to run the add-in if the file can't be found.

MSChartコントロールやデコードを必要とするその他のコントロール（例えば、DataTimePicker コントロール）を含む Form をマイグレーションする場合、VB Migration Partner はこの XML ファイルを探して、もしファイルが見つからなければ稼動しているアドインに問い合わせ警告を発します。

You install the VBMP Project Dumper by registering the ProjectDumperAddin.dll using the RegSvr32 utility:

RegSvr32 ユーティリティを使って ProjectDumperAddin.dll をレジストリ登録することで、VBMP プロジェクトのダンパーをインストールできます。

```
RegSvr32 ProjectDumperAddin.dll
```

4.8 Support for Dynamic Data Exchange (DDE)／動的データ交換(DDE)対応

Starting with release 1.20, VB Migration Partner supports DDE-related properties (LinkItem, LinkTopic, LinkMode, LinkTimeout), methods (LinkExecute, LinkPoke, LinkSend, LinkRequest) and events (LinkOpen, LinkClose, LinkExecute, LinkError, LinkNotify).

バージョン 1.20 のリリース開始と共に、VB Migration Partner は DDE 関連のプロパティ（LinkItem, LinkTopic, LinkMode, LinkTimeout）、メソッド（LinkExecute, LinkPoke, LinkSend, LinkRequest）、イベント（LinkOpen, LinkClose, LinkExecute, LinkError, LinkNotify）をサポートしてきました。

It's essential to bear in mind that DDE support is implemented by *simulating* the VB6 behavior, but without actually using any native DDE feature offered by Windows. This detail has an important consequence: DDE communications only work between VB.NET applications that has been converted by VB Migration Partner and that use VB Migration Partner's support library. If your original VB6 code uses DDE to communicate with Microsoft Excel or any other compiled DDE server application, it won't be impossible to establish the communication.

DDE サポートが VB6 の動作をシミュレートすることによって実装されますが、実際にどんなネイティブな DDE の特徴も使用せずに Windows によって提供されることを覚えておくのは重要です。この詳細は重要な結果を持っています。DDE との通信は VB Migration Partner によって変換された VB.NET アプリケーションとだけなされ、VB Migration Partner のサポートライブラリを使用します。もしオリジナルの VB6 コードが Microsoft Excel やその他のコンパイルされた DDE サーバーアプリケーションとコミュニケートするのに DDE を使う場合は、コミュニケーションを確立するのは不可能ではありません。

Internally, VB Migration Partner implements DDE-related features by means of inter-process Windows messages. In current release all DDE members are supported except the **LinkTimeout** property and the**LinkError** event. More precisely, DDE communications in converted VB.NET apps never time out and never raise a LinkError event. These features are likely to be implemented in a future release.

内部で、VB Migration Partner はインタープロセス Windows メッセージによって DDE 関連の機能を実装します。現在リリースされているものは、**LinkTimeout** プロパティと **LinkError** イベントを除いた全ての DDE メンバがサポートされ

ます。より正確に言うと、変換された VB.NET における DDE コミュニケートは決してタイムアウトしませんし、リンクエラーイベントも起きません。これらの機能は将来リリースされるバージョンでも実装されます。

When you migrate a VB6 form that works as a DDE server, the corresponding VB.NET form will react to all DDE requests in the form

DDE サーバーとして動く VB6 の Form をマイグレーションする場合、対応する VB.NET の Form はフォームの中の全ての DDE リクエストに反応することでしょう。

appname|*formlinktopic*

where *appname* is the App.Title property of the original VB6 project and *formlinktopic* is the value of the form's LinkTopic property. You can change the former value by means of the **VB6Config.DDEAppTitle**.

appname は元の VB6 プロジェクトの App.Title プロパティで、formlinktopic はフォームの LinkTopic プロパティの値です。**VB6Config.DDEAppTitle** によって、以前の値を変更することができます。

5. Pragma Reference／プラグマリファレンス

5. Pragma Reference／プラグマリファレンス

This section illustrates the purpose and usage of all the pragmas that VB Migration Partner currently supports.

このセクションでは、VB Migration Partner が現在サポートしているすべてのプラグマの目的と使用方法を例証します。

5.1 Project-level pragmas／プロジェクトレベルプラグマ

AddDataFile *filespec*

Copies a file into the output project's main directory and includes the file in the VB.NET project, so that it is automatically copied to the output folder when the project is compiled. It is useful to copy images, Access MDB files, and other data files into the target project. The *filespec* argument is mandatory: it must be a path relative to the VB6 project's folder (can't be an absolute path), can contain wildcards, and must be enclosed in double quotes if it includes spaces:

このプラグマは、出力プロジェクトのメインディレクトリにファイルをコピーして、VB.NET プロジェクトにファイルを含めます。そうすることで、プロジェクトがコンパイルされる際に自動的に出力フォルダにコピーされます。イメージファイル、AccessMDB ファイル、及びその他のデータファイルを対象のプロジェクトにコピーするのに役立ちます。*filespec* 引数は必須です。*filespec* は VB6 プロジェクトフォルダに対する相対パスの指定でなくてはなりません（絶対パスは指定不可）。*filespec* にはワイルドカードを含めることができます。*filespec* にスペースを含む場合は二重引用符で囲まなければなりません。

```
'## Rem Add the books.mdb database and all the files in the Images subfolder
'## AddDataFile books.mdb
'## AddDataFile Images¥*.*
```

AddDisposableType *typename*

Tells VB Migration Partner that a VB6 class is to be considered as a disposable type and be dealt with in a special way when a variable of this type is in the scope of an AutoDispose pragma. It is useful with COM objects that require finalization, for example objects that open a database connection. (Notice that VB Migration Partner recognizes as disposable ADODB objects such as Connection and Recordset.) The *typename* argument is mandatory and must include the type library name:

このプラグマは、指定したタイプの変数が AutoDispose プラグマのスコープにあるとき、VB6 のクラスがディスポーザブル型であるとみなし、特別な方法で対処するように VB Migration Partner に指示します。例えば、データベースのコネクションを開くオブジェクトのように、ファイナライゼーションを要求する COM オブジェクトに役立ちます。(VB Migration Partner は、Connection や Recordset のような ADODB オブジェクトを disposable と認識することに注意してください。)*typename* 引数は、必須であり、以下のようにタイプライブラリ名を含まなければなりません:

```
'## AddDisposableType CALib.DatabaseManager
```

AddImports *namespace [,explicit]*

Imports a .NET namespace at the project- or file-level. It is useful together with an AddReference pragma, to make all types of the referenced library accessible from the current project. The *namespace* argument is mandatory and must be enclosed in double quotes if it includes spaces:

このプラグマは、プロジェクトレベルかファイルレベルで.NET 名前空間をインポートします。参照されるすべてのタイプライブラリを現在のプロジェクトからアクセスできるようにする AddReference プラグマと一緒に使うと便利です。*namespace* 引数は必須であり、空白を含む場合は以下のように二重引用符で囲まれていなければなりません。:

```
'## AddReference "c:¥code architects¥appframework.dll"
'## AddImports CodeArchitects.AppFramework
```

If you specify True in the second argument, the namespace is imported at the file-level by means of an explicit Imports statement. For example the following pragma:

2 番目の引数で True を指定した場合、その namespace は明示的な Imports ステートメントによって、ファイルレベルでインポートされます。 次のプラグマサンプルをご参照下さい。:

```
'## AddImports ADODB, True
```

generates the following statement at the top of the file where the pragma resides:

このプラグマは、プラグマを設定したファイルの先頭に以下のステートメントを生成します:

```
Imports ADODB
```

Even if it is rarely necessary or desirable, you can even generate explicit Imports statements in all the files in the projects by using an explicit project scope:

殆ど必要でない場合でも、明示的にプロジェクトをスコープすることで、プロジェクト内のすべてのファイルに明示的な Imports ステートメントを以下のように生成することができます:

```
'## project:AddImports ADODB, True
```

It's worth noticing that the AddImports pragma has the added effect to drop all the imported namespace off type names, regardless of whether the namespace was imported at the project- or file-level. This feature helps producing more concise and readable code.

namespace がプロジェクトレベルかファイルレベルでインポートされたかどうかにかかわらず、タイプ名ですべてのインポートされた namespace を消すための AddImports プラグマには相加的効果があります。この機能は、より簡潔で読みやすいコードを作成するのに役立ちます。

AddLibraryPath *path, recurse, searchPattern*

Specifies folder where VB Migration Partner can search DLLs and type libraries that have been already converted to VB.NET. The *path* argument is the absolute name of a directory; *recurse* is a Boolean that specifies whether the search must be extended to subfolders (if False or omitted, the search is limited to the directory specified in the first argument); *searchPattern* allows you to restrict the effect of this pragma to certain files.

このプラグマは、VB Migration Partner が検索できる、既に VB.NET に変換された DLL とタイプライブラリのフォルダを指定します。*path* はディレクトリの絶対パスです。*recurse* は Boolean 型で、検索範囲をサブフォルダに拡張する必要があるかどうかを指定します。(false か省略されている場合、検索は最初の引数で指定されたディレクトリに制限されます。) *searchPattern* は、このプラグマの効果を特定のファイルに限定します。

```
'## AddLibraryPath "c:¥MyApps¥Libraries"
'## AddLibraryPath "c:¥Projects¥Net", True
```

The third argument is a regular expression that is applied against the base name of each file in the specified directory (or directory tree), that the file name without path and without extension (the extension is assumed to be ".dll"). For example, the following pragma:

3 番目の引数は、指定されたディレクトリ(または、ディレクトリツリー)の各ファイルの基底名に対して適用される正規表現であり、パスと拡張子のないファイル名(拡張子は".dll"とみなされます)です。次のプラグマサンプルをご参照下さい:

```
'## AddLibraryPath "c:¥Projects¥Net", False, "^CodeArchitects¥."
```

would include only the DLL files whose name starts with "CodeArchitects.".

上記のプラグマは、名前が「CodeArchitects.」で始まる DLL ファイルのみが対象になります。

You can apply the AddLibraryPath pragma only by storing it in a *.pragmas file. We suggest that you use the special file named VBMigrationPartner.pragmas, which you can conveniently share among all the projects of a complex application.

AddLibraryPath プラグマは、*.pragmas ファイルに含めることによってのみ適用することができます。複雑なアプリケーションのすべてのプロジェクト間で容易に共有できる VBMigrationPartner.pragmas という特別な名前のファイルを使用することを推奨します。

An important note:
if you are migrating a VB6 project group, this pragma should be included in the *.pragmas file that accompanies each project, regardless of whether that specific project uses the classes defined in the DLLs the pragma points to. If the *.vbp files are all located in the same folder, you can use one single VBMigrationPartner.pragmas file in that directory.

VB6 プロジェクトグループを移行している場合、そのプロジェクトがプラグマで指定した DLL に定義されているクラスを使用しているかどうかにかかわらず、全てのプロジェクトの*.pragmas ファイルにこのプラグマを含めなければなりません。*.vbp ファイルが全て同一フォルダに配置されているなら、そのディレクトリ内にただ一つの VBMigrationPartner.pragmas ファイルを使用することが出来ます。

AddSourceFile *filename [, addaslink]*

Adds a VB.NET source file to the project generated by VB Migration Partner. The *filename* argument is the absolute path of a *.vb source file; the second optional parameter is False (or omitted) if the file is copied to the output folder and included in the VB.NET project as a regular file, True if the file is added as a link an existing file. The pragma recognizes form and usercontrol source files and automatically imports the *.Designer.vb and *.resx files, if they exist.

このプラグマは、VB Migration Partner によって生成されたプロジェクトに VB.NET ソースファイルを追加します。*filename* 引数は*.vb ソースファイルの絶対パスです。2 番目の任意のパラメータは、False（または省略）の場合、ファイルが出力先フォルダにコピーされ、通常ファイルとして VB.NET プロジェクトに含まれます。True の場合は、そのファイルは既存ファイルのリンクとして追加されます。このプラグマは、フォームとユーザーコントロールのソースファイルを認識し、*.Designer.vb と*.resx ファイルが存在する場合、自動的にインポートします。

```
'## AddSourceFile "c:¥code architects¥AppFrameworkFormBase.vb"
'## AddSourceFile "c:¥code architects¥AppMainForm.vb"
'## AddSourceFile "c:¥code architects¥AppFrameworkCommon.vb", True
```

If the source file being imported requires a reference to an external DLL, you must provide an AddReference pragma; if the source file being imported assumes a project-level Imports, you must provide a suitable AddImports pragma. These pragma must be included in one of the original VB6 files:

インポートされるソースファイルが、外部の DLL の参照を必要とする場合は、AddReference プラグマを設定しなければなりません。インポートされるソースファイルが、プロジェクトレベルでインポートされる場合、適切な AddImports プラグマを設定しなければなりません。これらのプラグマはオリジナルの VB6 ファイルの 1 つに含まれていなければなりません。

```
'## AddReference "c:¥code architects¥appframework.dll"
'## AddImports CodeArchitects.AppFramework
```

AddReference *asmPathOrName*

Include a reference to a .NET assembly in the VB.NET project being created. It is useful when the code you inject by means of InsertStatement pragmas requires a .NET assembly that isn't automatically referenced by default, for example System.Data.OracleClient. The argument is mandatory, must be either the assembly's absolute file path or the assembly's display name, and must be enclosed in double quotes if it includes spaces:

このプラグマは、作成される VB.NET プロジェクトに、.NET アセンブリへの参照を含めます。これは、InsertStatement pragmas によって挿入するコードが、デフォルトで自動的に参照されない NET アセンブリを必要とする場合に役に立ちます。例えば、System.Data.OracleClient がそれにあたります。引数は必須であり、アセンブリの絶対パスかアセンブリの表示名のどちらかでなければいけません。また、半角スペースを含む場合は二重引用符で閉じられていなければなりません。

```
    '## AddReference
C:¥WINDOWS¥Microsoft.NET¥Framework¥v2.0.50727¥System.Data.OracleClient.dll
    '## AddReference "System.EnterpriseServices, Version=2.0.0.0,
Culture=neutral,
        PublicKeyToken=b03f5f7f11d50a3a"
```

You typically specify the assembly's display name for assemblies stored in the GAC.

通常、GAC に格納されているアセンブリに対する表示名を指定します。

ApplyRenameRules *xmlfile*

Specifies that VB Migration Partner should apply the rename rules contained in the specified XML file, or in the default RenameRules.xml file stored in the install folder:

このプラグマは、VB Migration Partner が指定された XML ファイル、または、インストールフォルダに格納されているデフォルトの RenameRules.xml ファイルに含まれたリネームルールを適用するように指定します。

```
    '## ApplyRenameRules c:¥app¥rules.xml
```

This pragma can be specified in both a *.pragmas file or in source code, however it affects **all** the projects in the solution being migrated.

このプラグマは*.pragmas ファイルとソースコードの両方で指定することが出来ますが、変換されるソリューションに含まれる**全ての**プロジェクトに影響を与えます。

For more information about renaming rules, see the Renaming program members section.

リネームルールに関する詳しい情報に関しては、プログラムのメンバの名前を変更するの項をご覧下さい。

AssemblyKeyFile *filename.snk, delaysign*

Signs the VB.NET assembly that results from the conversion, using the specified .snk file.
The *filename.snk*argument is the complete (absolute) path of the .snk file that contains the public/private key pair; *delaysign* is a Boolean that indicates whether the assembly must be delay-signed (default is False).

このプラグマは、指定された.snk ファイルを使用して、変換した結果できあがる VB.NET アセンブリに署名します。*filename.snk* 引数は、公開/秘密鍵の組み合わせを含む.snk ファイルの完全(絶対)パスです。*delaysign* 引数は、そのアセンブリが遅延署名されるかどうかを示す Boolean 型です。(デフォルト値は False です。)

```
'## AssemblyKeyFile "c:¥codearchitects.snk"
```

Using this pragma is equivalent to tick the "Sign the assembly" option in the Signing tab of the My Project designer after the migration.

このプラグマを使用することは、変換後に My Project のプロパティ画面の「署名」タブの「アセンブリの署名」オプションにチェック入れることと同じです。

BinaryCompatibility *force*

Determines whether the migrated VB.NET project should preserve binary compatibility with the COM component written in VB6. If the optional argument is False or omitted, then compatibility is enforced only if the original VB6 project specified binary compatibility; if the argument is True, then compatibility is enforced in any case.

このプラグマは、変換された VB.NET プロジェクトが VB6 で書かれた COM コンポーネントとのバイナリ互換性を維持するかどうかを決定します。任意の引数が False か省略されたとき、オリジナルの VB6 プロジェクトがバイナリ互換性を指定していた場合にのみ、互換性は維持されます。引数が True のときは、どのような場合でも、互換性は維持されます。

```
'## project:BinaryCompatibility True
```

When this pragma is active, VB Migration Partner generates a ComClass attribute for each public class in the generated VB.NET project; this attribute specifies the GUIDs of the original coclass, class interface, and event interface of the original VB6 class. In addition, a project-level Guid attribute in AssemblyInfo.vb is generated so that the .NET assembly has same GUID as the original type library. Finally, the Register for COM Interop setting is enabled in the MyProject property page. The neat effect of all these operations is that the .NET DLL is fully compatible with the original COM DLL and all existing VB6/COM clients can continue to work as before but use the new .NET DLL instead. (No recompilation is needed.)

このプラグマがアクティブであれば、VB Migration Partner は、生成された VB.NET プロジェクトの各パブリッククラスに対して ComClass 属性を生成します。この属性は、オリジナル VB6 クラスのイベントインターフェース、クラスインターフェース、オリジナルの共同クラス(coclass)の GUID を指定します。さらに、AssemblyInfo.vb にプロジェクトレベルの GUID 属性が生成されるので、.NET アセンブリはオリジナルのタイプライブラリと同じ GUID を保持します。最終的に、MyProject のプロパティページの COM Interop 設定への登録は有効になります。これらの全ての作用の効果の

素晴らしさは、代わりに新しい.NET DLL を利用するとしても、.NET DLL がオリジナル COM DLL と完全な互換性を持ち、すべての既存の VB6/COM クライアントが以前のように動作するということです。(再コンパイルは不要です)

By default, VB Migration Partner uses the COM DLL specified as the binary-compatible DLL for the original VB6 project. (This is the DLL selected in the Component tab of the Project Properties dialog box in the VB6 IDE.) If the original VB6 project doesn't specify binary compatibility (but the *force* argument is True) then VB Migration Partner uses the output DLL to extract all existing GUIDs.

デフォルトでは、VB Migration Partner はオリジナル VB6 のプロジェクトでバイナリ互換の DLL として指定された COM DLL を使用します。(これは、VB6IDE の Project プロパティダイアログボックスのコンポーネントタブで選択された DLL です。) オリジナルの VB6 プロジェクトがバイナリ互換を指定していない場合、(*force* 引数は True であっても)VB Migration Partner は、すべての既存の GUID を抽出するの出力用(output)DLL を使用します。

This pragma is ignored and no warning message is emitted if the VB6 project type isn't ActiveX DLL or if VB Migration Partner can't find a COM DLL to be used to extract coclass and interface GUIDs.

VB6 プロジェクトタイプが ActiveX DLL でない、あるいは、共同クラス(coclass)とインターフェースの GUID を抽出するのに使用される COM DLL を VB Migration Partner が認識できない場合、このプラグマは無視され、Warning メッセージは出力されません。

As a side-effect of this pragma, all public fields in public classes are rendered as properties: this is necessary because public fields aren't made visible to COM clients by the ComClass attribute. In addition, if the field was declared using As New, code is generated to preserve the auto-instancing semantics, regardless of whether the field is under the scope of an AutoNew pragma. The effect is therefore similar to the AutoProperty pragma, except that the BinaryCompatibility pragma affects only fields in public classes inside ActiveX DLL projects.

このプラグマの副次的な効果として、パブリッククラスのすべてのパブリックフィールドがプロパティと見なされます。これは、COMClass 属性によってパブリックフィールドが COM クライアントに対して可視化されないために必要です。さらに、そのフィールドが As New を使って宣言された場合、ソースコードは、そのフィールドが、AutoNew プラグマのスコープにあるかどうかにかかわらず、オートインスタンシングを維持するように生成されます。そのため、この効果は、BinaryCompatibility プラグマが ActiveX DLL プロジェクトの中のパブリッククラスのフィールドだけに影響することを除いては、AutoProperty プラグマと類似しています。

EnableAppFramework *mainform, splashform, visualstyles, shutdownmode, singleinstance*

Enables the VB.NET application framework feature. The parameter passed to this pragma correspond to the settings you can find in the Application tab of the My Project designer in Visual Studio. The *mainform* argument is the name of the startup form and is the only mandatory parameter for this pragma; *splashform* is the name of the splash form, if any; *visualstyles* is True if you want to enable Windows XP styles (the default is False);*shutdownmode* is 0 (the default) if the application exits when the main form closes or 1 if the application exits when all forms close; *singleinstance* is True if you want to prevent multiple instances of this application (the default is False).

このプラグマは、VB.NETアプリケーション・フレームワークの機能を有効にします。このプラグマに渡されたパラメータは、VisualStudio の My Project デザイナの Application タブ内において確認できる設定に対応しています。*mainform* 引数は、スタートアップフォームの名前であり、このプラグマの唯一の必須パラメータです。もしあれば、*splashform* 引数は、スプラッシュフォームの名前です。Windows XP スタイルを有効にしたい場合、*visualstyles* 引数を True に設定して下さい。(デフォルトは False です。)*shutdownmode* 引数に、0 が設定された場合(デフォルト)、メインフォームがクローズされるとアプリケーションが終了し、1 ならば、全てのフォームがクローズした時にアプリケーションが終了します。このアプリケーションのインスタンスを複数作らせたくないなら、*singleinstance* 引数を True に設定して下さい。(デフォルトは False です。)

For example, the following pragma sets frmMain as the main form, frmSplash as the splash screen form, enables XP visual styes and forces the application to close only when all loaded forms are closed:

以下のプラグマは、「スタートアップフォーム」に「frmMain」、「スプラッシュスクリーンフォーム」に「frmSplash」、「XP Visual スタイルを有効にする」にチェック、「シャットダウンモード」に「全てのフォームがクローズした時アプリケーションを終了」が設定されるという例です。

```
'## EnableAppFramework frmMain, frmSplash, true, 1
```

You should use this pragma only in standard EXE projects. When this pragma is specified, the Sub Main form is ignored (if the application has one).

このプラグマは、標準の EXE プロジェクトだけに使用して下さい。このプラグマが指定されるとき、(アプリケーションに Sub Main フォームが存在する場合は)Sub Main フォームは無視されます。

ImportTypeLib *typelibfilename, tlbimpcommand, assemblyname, copytoprojectfolder*

Specify the command line to be passed to the TlbImp.exe tool when importing a type library. *typelibfilename* is the name of the .dll or .tlb file to be imported (can be just the file name or the complete file path); *tlbimpcommand* is the complete command that must be passed to the TlbImp.exe tool.

このプラグマは、タイプライブラリをインポートする際に、TlbImp.exe ツールに渡されるコマンドラインを指定します。*typelibfilename* 引数は、インポートされる dll か.tlb ファイルの名前です。(ファイル名のみ、または完全なファイルパスも指定可能です。) *tlbimpcommand* 引数は、TlbImp.exe ツールに渡す完全なコマンドです。

The third and fourth arguments are optional and are taken into account only if the tlbimpcommand argument starts with a "@" character (see later).

3 番目と 4 番目の引数は、任意であり、tlbimpcommand 引数が"@"キャラクタから始まる場合にのみ考慮されます。(後述)

Notice that you need to specify the entire path in the first argument only if the current VB6 application uses two distinct type libraries with same file name:

現在の VB6 アプリケーションが、同じファイル名の 2 つの異なったタイプライブラリを使用する場合にだけ、第 1 引数に完全なパスを指定する必要があることに注意してください:

```
    '## ImportTypeLib mytypelib.dll, "c:¥myapp¥mytypelib.dll
/tlbreference:c:¥myapp¥support.dll
        /tlbreference:c:¥myapp¥interfaces.dll /keyfile:c:¥mycompany.snk"
```

This pragma is necessary when VB Migration Partner fails to correctly locate the type library to be imported or any of the *.dll or *.tlb files such a type library depends on, or when you need to specify any additional option for the TlbImp.exe tool, such as when you want to sign the imported assembly with a strong name. Notice that VB Migration Partner uses this pragma only to correctly resolve a reference to a type library: the type library is actually imported in the current project only if it was referenced by the original VB6 project.

このプラグマは、VB Migration Partner がインポートされるタイプライブラリやタイプライブラリが依存する*.dll や*.tlb ファイルの場所を正確に特定できない場合や、厳密な名前でインポートされたアセンブリに署名したいときのように、TlbImp.exe にオプションを指定しなければならない場合に必要となります。VB Migration Partner はタイプライブラリの参照を正常に解決するためだけにこのプラグマを使用することに注意してください。つまり、オリジナルの VB6 プロジェクトで参照されていた場合にのみ、現在のプロジェクトにタイプライブラリは実際にインポートされます。

You can apply this pragma only by storing it in a *.pragmas file. We suggest that you use the special file named VBMigrationPartner.pragmas, which you can conveniently share among all the projects of a complex application. (As explained above, a type library is effectively imported only if it is referenced by the original VB6 project and this pragma only teaches VB Migration Partner how the type library must be imported.)

*.pragmas ファイルに格納することによってのみ、このプラグマを適用することができます。複雑なアプリケーションのすべてのプロジェクト間で共有できる、VBMigrationPartner.pragmas という特別な名前のファイルを使用することをお勧めします。(上述のように、オリジナルの VB6 プロジェクトによって参照されている場合にだけ、有効にタイプライブラリがインポートされます。また、このプラグマは、どのようにタイプライブラリをインポートしなければならないかだけを VB Migration Partner に伝えます。)

As a special case, if the second argument starts with a “@” character it is interpreted as the path of a .NET assembly that is equivalent to the original type library. This syntax variation is useful in two cases: First, if you have already (manually) converted the type library and have obtained the corresponding .NET assembly; second, if the author of the original type library has provided a Primary Interop Assembly for the type library and VB Migration Partner fails to correctly locate such a PIA. Here’s an example of this technique

特殊なケースとして、2 番目の引数が「@」キャラクタから始まっている場合、オリジナルのタイプライブラリと同等な.NET アセンブリのパスとして解釈されます。この構文のバリエーションは 2 つのケースで役立ちます。つまり、1. 既に(手動で)タイプライブラリを変換し、対応する.NET アセンブリを取得している場合。2. タイプライブラリの作者がタイプライブラリに対してプライマリ相互運用機能アセンブリ (PIA) を提供しているが、VB Migration Partner がその PIA の場所を正しく特定できない場合です。以下がこのテクニックの例になります:

```
    '## ImportTypeLib msword.olb,
"@c:¥assemblies¥microsoft.office.interop.word.dll"
```

If the second argument begins with the “@” character, then the *assemblyname* optional argument should be specified and should be equal to the name of the assembly. This name is usually equal to the assembly’s root

namespace, but it can also be a different string. Specifying this value allows VB Migration Partner to determine whether the .NET DLL should be added as a reference for the project being migrated, without having to actually load all the assemblies in all folders referenced by the project itself.

2 番目の引数が「@」から始まる場合、任意の *assemblyname* 引数は指定されるべきであり、アセンブリの名前と等しくなるべきです。この名前は通常アセンブリのルートネームスペースと等しいのですが、異なった文字列にすることもできます。この値を設定することで VB Migration Partner は、プロジェクト自身によって参照されるフォルダすべてのアセンブリを実際には読み込んでいない場合に、.NET DLL が変換されたプロジェクトの参照として追加されるべきかどうかを決定することができます。

For example, assume that you have already migrated a VB6 DLL named “CAFramework” and created an assembly named “CodeArchitects.Framework.dll”. Here’s the pragma that does the trick:

たとえば、すでに「CAFramework」という名前がつけられた VB6 DLL を変換しており、また、「CodeArchitects.Framework.dll」という名前がつけられたアセンブリを作成したと仮定してください。目的を果たすプラグマは、以下になります：

```
    '## ImportTypeLib CAFramework.dll,
"@c:¥assemblies¥CodeArchitects.Framework.dll", "CAFramework"
```

If you omit the *assemblyname* argument VB Migration Partner attempts to locate the assembly by matching the assembly name with the filename. For example, in previous example we could omit the third argument if the .NET file were named CAFramework.dll.

assemblyName 引数を省略した場合、VB Migration Partner は、ファイル名とアセンブリ名を照合することによってアセンブリの場所を見つけようとします。例えば、前の例で.NET のファイルが CAFramework.dll と名付けられていた場合は、3 番目の引数を省略できます。

If the second argument begins with the “@” character, then the *copytoprojectfolder* optional argument specifies whether the converted VB.NET project should include a reference to the specified .NET DLL (if the argument is True) or if VB Migration Partner should first copy the .NET DLL into the SupportDLLs subfolder and have the VB.NET project reference this copy. In most cases you can omit this argument and rely on the default behavior (the DLL isn’t copied into SupportDLLs folder); you should specify True only if you later want to deploy the VB.NET project on another computer where the specified .NET DLL isn’t present.

2 番目の引数が「@」で始まる場合、任意の *copytoprojectfolder* 引数は変換された VB.NET プロジェクトが指定された.NET DLL の参照を含める（引数が True の場合）べきかどうか、または、VB Migration Partner が.NET DLL をコピーして SupportDLLs サブフォルダに入れ、VB.NET プロジェクトにコピーされた DLL を参照させるかどうかを指定します。ほとんどの場合、この引数を省略してデフォルトの動作に準拠させることができます。（その DLL は SupportDLLs フォルダにはコピーされません）。指定された.NET DLL が存在しない別のコンピュータ上に VB.NET プロジェクトを後で配置したい場合にだけ、True を指定すべきです。

ProjectKind *prjkind*

Determines the type of the current project. It is useful to specify whether an ActiveX EXE project should be converted to an EXE or a DLL Visual Basic project. The *prjkind* argument is mandatory and can be equal to *dll*or *exe*:

このプラグマは現在のプロジェクトのタイプを決定します。ActiveX EXE プロジェクトを EXE の Visual Basic プロジェクトに変換するか DLL の Visual Basic プロジェクトに変換するかを指定するのに役立ちます。*prjkind* 引数は必須で *dll* または *exe* を指定します:

```
'## Rem convert the current ActiveX project as a class library
'## ProjectKind dll
```

SetPragmaPrefix *newprefix*

Allows you to select a different prefix for pragmas inserted in VB6 code. The argument must not include the leading single quote used for remarks:

このプラグマは、VB6 コードに挿入されたプラグマのために、異なった接頭語を選択することを許可します。引数は注釈(コメント)に使用される先頭のシングルクォーテーションを含んではいけません:

```
'## SetPragmaPrefix %%
```

The argument is considered to be a regular expression, therefore it is necessary to escape the characters that have a special meaning inside a regex. For example, if you want to use a double dot as the pragma prefix, you need this pragma:

この引数は正規表現とみなされます。したがって、正規表現内で特別な意味を持つ文字をエスケープする必要があります。たとえば、プラグマ接頭語としてピリオドを二つ使いたい場合、以下のプラグマが必要となります:

```
'## SetPragmaPrefix ¥.¥.
```

Using a regex instead of a simple string allows you to specify multiple pragma prefixes. For example, you can have VB Migration Partner recognize the double dot prefix in addition to the default prefix:

簡単なストリングの代わりに正規表現を使用して複数のプラグマ接頭語を指定することが出来ます。例えば、VB Migration Partner に、デフォルトの接頭語に加えて二つのピリオドを接頭語として認識させることができます:

```
'## SetPragmaPrefix (##|¥.¥.)
```

The prefix doesn't need to be 2-char long.

この接頭語の長さを 2 文字にする必要はありません。

You can apply this pragma only by storing it in a *.pragmas file. We suggest that you use the special file named VBMigrationPartner.pragmas, which you can conveniently share among all the projects of a complex application.

Notice, however, that this pragma doesn't affect the pragmas that are stored in *.pragmas files and applies only to pragmas inserted in VB6 source code files.

このプラグマは*.pragmas ファイルに記述することでのみ適用できます。複雑なアプリケーションのすべてのプロジェクト間で容易に共有できる、VBMigrationPartner.pragmas という特別なファイルを使用することをお勧めします。しかし、このプラグマは他の*.pragmas ファイルに保存される他のプラグマには有効ではなく、VB6 ソースコードファイルに挿入されたプラグマに対してのみ適用されることに注意してください。

This pragma can be useful for two distinct and unrelated reasons:

このプラグマは、関係のない二つの別の理由から役立つ場合があります。

a. You use another source code management tool that relies on remarks that start with the '## sequence. (For example, only later in the development cycle we realized that this sequence is used by Innovasys Document! Tool).
「'##」から始める注釈（コメント）を使用する他のソースコード管理ツールを使用している場合。(例えば、Innovasys Document!Tool によってこの文字列が使用されていることを私たちは開発サイクルの後半になってから気付きました。)
b. You want to migrate the VB6 code using different sets of pragmas.
VB6 のコードを異なる組み合わせのプラグマを使用して変換したい場合。

As an example of the second approach, let's suppose that your VB6 code uses DAO to access an Access database. In the first migration attempt, you generate a VB.NET application that also uses DAO, but you can then improve the result by generating ADO.NET code, which surely requires a good amount of PostInclude, PostProcess, and InsertStatement pragmas. In this case it might make sense to use a different prefix for the pragmas that add just to support ADO.NET, so that you can quickly switch from the DAO to the ADO.NET version by just editing one SetPragmaPrefix pragma.

2 番目のアプローチの例は、VB6 のソースコードが Access データベースに DAO を使用して接続する場合です。最初の変換では DAO を使用する VB.NET アプリケーションを生成したとしても ADO.NET のソースコードを生成することで結果を改善することができます。それには、PostInclude、PostProcess と InsertStatement プラグマを適切に使用しなければなりません。この場合、ADO.NET をサポートするためのプラグマに異なる接頭語を使用することを理解できるかもしれません。その結果、SetPragmaPrefix pragma を編集するだけで、迅速に DAO から ADO.NET に切り替えることができます。

5.2 Pragmas that affect classes／クラスに影響を与えるプラグマ

ClassRenderMode *mode*

Specifies how a class must be rendered during the migration process. The *mode* argument can be *Class, Interface, ClassAndInterface,* and *Module*. This pragma can't be specified at the project-level:

このプラグマは、変換中にどのようにクラスを変換するかを指定します。*mode* 引数は、*Class* 引数にも、*Interface* 引数にも、*ClassAndInterface* 引数にも、*Module* 引数にもなります。 プロジェクトレベルでこのプラグマを指定することは出来ません:

```
'## ClassRenderMode Interface
```

By default, all VB6 classes are rendered as VB.NET classes; however, if the class name appears in one or more Implements statements, the class is also rendered as an interface. This pragma allows developers to generate the interface even if the class doesn't appear in any Implements statement (which may be necessary when converting an ActiveX DLL or ActiveX EXE project without also converting a client application) as well as generate only the interface and not the original class. This pragma can also be useful to convert a GlobalMultiuse or GlobalSingleUse class into a VB.NET module.

デフォルトでは、すべての VB6 のクラスは、VB.NET のクラスにされます。しかしながら、クラス名が 1 つ以上の Implements ステートメントに記述されている場合、クラスはインターフェースにもなります。このプラグマは、クラスがどの Implements ステートメントにもなくてもインターフェースを生成させることを可能にします。(これは、クライアントのアプリケーションを移行せずに ActiveX DLL や ActiveX EXE を移行するときに必要になるかもしません。)それだけではなく、元のクラスにはないインターフェースを生成することもできます。このプラグマは、GlobalMultiuse クラスや GlobalSingleUse クラスを VB.NET モジュールに変換するのに役立ちます。

ExcludeCurrentFile *boolean*

Specifies whether the current file must be ignored by VB Migration Partner. It is useful for files containing modules and classes that have a VB.NET counterpart – for example, the methods in the VBMigrationPartner_Support module. If the argument is True or omitted, the current file is ignored by VB Migration Partner:

このプラグマは、VB Migration Partner が現在のファイルを無視しなければならないかどうか指定します。それは、モジュールを含むファイルと VB.NET に対応したクラス-例えば VBMigrationPartner_Support モジュールのメソッド-にとって役立ちます。引数が True か省略された場合、現在のファイルは VB Migration Partner に無視されます:

```
'## ExcludeCurrentFile
```

This pragma can't be specified at the project-level.

プロジェクトレベルでこのプラグマを指定することはできません。

GenerateEventDispatchers *boolean*

Specifies how event definitions in user controls should be rendered. By default, VB Migration Partner generate an event dispatcher class for each event exposed by the user control, and then uses the dispatcher's Raise method instead of the standard RaiseEvent keyword. If the argument is False then the standard .NET event mechanism is used instead:

このプラグマは、ユーザコントロールのイベント定義をどのように変換するかを指定します。デフォルトでは、VB Migration Partner は、ユーザコントロールによって公開されている各イベントに対してイベントディスパッチャクラス

を生成し、標準 RaiseEvent キーワードの代わりにディスパッチャの Raise メソッドを使用します。引数が False の場合、標準的な.NET イベントが代わりに使われます。

```
'## Rem don't use event dispatchers in this user control
'## ExcludeCurrentFile
```

VB Migration Partner's default implementation of events inside user controls ensures that events are handled correctly even if the control belongs to a control array or is instantiated dynamically by means of a Controls.Add method. You can disable this feature and produce more compact and efficient code if you are sure that neither condition is met.

ユーザコントロール内部のイベントの VB Migration Partner におけるデフォルトの実装は、コントロールがコントロール配列に属している場合や、Controls.Add メソッドによって動的にインスタンス化されるとしても、イベントが正しくハンドルされることを保証します。どちらの条件にも合わないことが確かならば、この機能を無効にしてより小さくて効率的なソースコードを生成することができます。

ReuseResxFiles *resxpath*

Allows VB Migration Partner to reuse existing *.resx files, either created manually or generated by a previous migration (possibly with the Upgrade Wizard). The *resxpath* argument is the path of the folder that contains the existing *.resx files.

このプラグマにより、VB Migration Partner は、手作業で作成された、または、前回の変換（おそらくアップグレードウィザード）で生成された既存の*.resx ファイルを再利用できます。resxpath 引数は、既存の*.resx ファイルを含むディレクトリパスです。

```
'## project:ReuseResxFiles c:¥MigratedApps¥TestApp
```

The argument can be either an absolute or a relative path; in the latter case, it is considered to be relative to folder where the VB6 project being migrated resides. In addition, the ${ProjectPath} placeholder is replaced by the name of the folder where the VB6 project is stored. This placeholder and the ability to use relative path allows you to store this pragma in the global VBMigrationPartner.pragmas file, provided that you have used a consistent naming schema for your migrated apps. For example, if all your migrated projects are stored in the c:¥MigratedApps directory tree, you might use the following pragma

この引数は絶対パスまたは相対パスのどちらかになり得ます。後者の場合、変換された VB6 プロジェクトが存在するフォルダに対する相対パスとなります。さらに、プレースホルダーの$(ProjectPath)は VB6 のプロジェクトが格納されているフォルダの名前に置き換えられます。このプレースホルダーと相対パスのおかげで、移行したアプリケーションに対して、一貫した命名体系を使用したことがあるならば、このプラグマをグローバル VBMigrationPartner.pragmas ファイルに保存することができます。例えば、すべての変換されたプロジェクトが c:/MigratedApps ディレクトリツリーに格納される場合、以下のプラグマを利用できます。

```
'## project:ReuseResxFiles c:¥MigratedApps¥${ProjectPath}
```

This pragma is especially useful when converting forms that contain one or more ActiveX controls. In some (rare) cases, in fact, VB Migration Partner fails to correctly generate the OcxState property for these ActiveX controls. If you see that an ActiveX control isn't migrated correctly, you might try to convert the project using Microsoft Upgrade Wizard, and then use this pragma so that VB Migration Partner can reuse the generated *.resx files.

このプラグマは、1 つ以上の ActiveX コントロールを含むフォームを変換するときに特に役立ちます。いくつかの(稀な)ケースで、実際に、VB Migration Partner は、これらの ActiveX コントロールに対して正しく OcxState プロパティを生成することができないことがあります。ActiveX コントロールが正確に変換されていないことがわかった場合、MSUpgrade Wizard を使用してプロジェクトを変換し、生成された*.resx ファイルを再利用するようこのプラグマを使用してください。

Notice that you can specify this pragma at the project-, file, or component-level. When used at the file or component level, you can omit the argument to disable this feature for a specific form or control.

プロジェクトレベル、ファイルレベル、またはコンポーネントレベルでこのプラグマを指定できることに注目してください。ファイルレベルまたはコンポーネントレベルで使用する場合は、特定のフォームまたはコントロールに対してこの機能を無効にするための引数を省略することができます。

VariantConversionSupport *bool*

Indicates that the current class should be extended with two CType Operator methods that provide support for implicit conversion from/to the VB6Variant type. The argument should be True (or omitted) to enable this feature, or False to disable it. (The feature is disabled by default, therefore you typically use the False value only to override the setting at the project-level.)

このプラグマは、現在のクラスが、VB6Variant タイプへの(あるいは VB6Variant タイプからの)暗黙的な変換をサポートする 2 つの CType 演算子法(関数)で拡張されるべきであることを示します。この機能は、引数が True(または省略)の場合に有効になり、False の場合に無効になります。(この機能のデフォルトは使用不可です。したがって、通常は、プロジェクトレベルの設定を無効にするためにだけ False 値を使います。)

```
'## project:VariantConversionSupport
```

Starting with release 1.11, all the classes defined in the VBLibrary support implicit conversion from/to the VB6Variant type. In general it is a good idea to extend this feature to all the forms, classes, and usercontrols of the projects you migrate.

リリース 1.11 以降は、VBLibrary で定義されているすべてのクラスは、VB6Variant タイプへ(から)の暗黙的な変換をサポートします。一般に、変換を行うプロジェクトのすべてのフォーム、クラス、そしてユーザコントロールに対してこの機能を拡張するのが良い方法でしょう。

5.3 Pragmas that affect fields and variables／フィールドと変数に影響を与えるプラグマ

AddAttribute *text*

Associates a custom attribute to the current project, class, method or a specific variable. The text argument must be a valid VB.NET attribute. Surrounding < and > delimiters are optional:

このプラグマは、現在のプロジェクト、クラス、メソッドや特定の変数に対してカスタム属性を関連付けます。text 引数は有効な VB.NET の属性でなければなりません。囲むためのデリミタの「 < 」と「 > 」は任意です。

```
'## project:AddAttribute Description("this is an assembly")

Sub SetSize(w As Integer)
    ' attributes associated with a method can be inserted inside the method
    '## AddAttribute MyCustomAttribute(true)
    ' attributes associated with parameters require a prefix
    '## x.AddAttribute Description("the object's width")
    ...
End Sub
```

This pragma may not work as intended with fields that are converted into separate properties, e.g. public variables declared with As New that fall under the scope of an AutoNew pragma.

このプラグマは別々のプロパティに変換されるフィールドには意図したとおりに機能しないかもしれません。例えば、AutoNew プラグマのスコープにある As New で宣言されたパブリック変数がそうです。

ArrayBounds *mode*

Specifies how the declaration of arrays with a nonzero lower index must be converted to VB.NET. The valid values for the *mode* argument are *Unchanged* (arrays are migrated verbatim, the default behavior), *ForceZero* (the lower index is forced to be zero), *Shift* (both the lower and upper indexes are shifted so that the number of elements in the array doesn't change), *VB6Array* (convert the array into a VB6Array object), or *ForceVB6Array* (convert the array into the VB6Array even if the lower index is zero):

このプラグマは、下限のインデックスがゼロ以外の配列の宣言をどのように VB.NET.に変換するかを指定します。*mode* 引数に対する有効な値は、*Unchanged*（デフォルトの動作で、配列は、言葉通りに変換されます。）、*ForceZero*（下限のインデックスをゼロにします。）、*Shift*（配列の要素の数が変わらないように下限と上限の両方のインデックスをずらします。）、*VB6Array*（は配列を VB6Array オブジェクトに変換します。）、そして *ForceVB6Array*（は、下限のインデックスがゼロだったとしても、配列を VB6Array に変換します）です。

```
'## ArrayBounds VB6Array
```

Notice that this pragma, like all pragmas that can reference arrays, should be inserted immediately before the Dim statement that declares the array, rather than before the ReDim statement that actually creates the array. Pragmas inserted inside the method where the ReDim statement appears are ignored.

このプラグマは、配列を参照する全てのプラグマのように、実際に配列を作る ReDim ステートメントの前ではなく、配列を宣言する Dim ステートメントの直前に挿入されなければならないことに注意してください。ReDim ステートメントが現れるメソッドの内側で挿入されたプラグマは、無視されます。

Also notice that this pragma only affects the declaration (DIM) of an array, not the value of indexes used in code to reference individual array items. For that purpose you should use a ShiftIndexes pragma.

また、このプラグマが配列の宣言(DIM)にのみ効果があることに注意して下さい。個々の配列要素を参照するソースコード上の要素の値には効果がありません。これを解決するためには ShiftIndexes プラグマを使用するべきです。

ArrayRank *rank*

Specifies the rank of an array. It is useful when VB Migration Partner isn't able to determine the correct rank of a multi-dimensional array taken as an argument, returned by a property or function, or declared at the class level but initialized elsewhere. Consider the following VB6 example:

このプラグマは、配列の次元数を指定します。この引数は、VB Migration Partner が、クラスレベルで宣言されてはいるが他の場所で初期化された、または、ファンクションやプロパティによって戻された、引数となっている多次元配列の正しい次元数が判断出来ないときに役立ちます。以下の VB6 の例を考えてください:

```
'## arr.ArrayRank 2
'## GetArray.ArrayRank 2
Dim arr() As Integer

Function GetArray(initialize As Boolean) As Integer()
    If initialize Then ReDim arr(10, 10)
    GetArray = arr
End Function
```

If it weren't for the ArrayRank pragma, the field and the method would return a vector instead of a two-dimensional array:

もし、ArrayRank プラグマがなかったら、フィールドとメソッドは二次元配列の代わりに一次元配列を返します。

```
Dim arr(,) As Short

Function GetArray(initialize As Boolean) As Integer(,)
    If initialize Then ReDim arr(10, 10)
    Return arr.Clone()
End Function
```

Notice that this pragma must have a nonempty scope.

このプラグマは、空ではないスコープを持たねばならないことに注意してください。

Also notice that this pragma, like all pragmas that can reference arrays, should be inserted immediately before the Dim statement that declares the array, rather than before the ReDim statement that actually creates the array. Pragmas inserted inside the method where the ReDim statement appears are ignored.

また、このプラグマは、実際に配列を作成する ReDim ステートメントの前ではなく、配列を参照することができるすべてのプラグマと同様、配列を宣言する Dim ステートメントの直前に挿入すべきです。ReDim ステートメントがある場所に挿入されたプラグマは無視されます。

AssumeType *type*

Informs VB Migration Partner that a given Object, Variant, or Control variable must be translated as if it were of a specific VB6 type. It is useful to correctly solve default properties even in late-bound references. For example, consider the following VB6 code:

このプラグマは、与えられた Object、Variant または Control が、あたかも特定の VB6 タイプから変換されたように変換されなければならないことを VB Migration Partner に伝えます。これは遅延バインディングの参照であった場合においてもデフォルトプロパティの正しい解決に役立ちます。以下の VB6 の例を御覧下さい:

```
'## ctrl.AssumeType TextBox
Dim ctrl As Control

For Each ctrl In Me.Controls
    If TypeOf ctrl Is TextBox Then ctrl = ""
Next
```

If it weren't for the AssumeType pragma, VB Migration Partner wouldn't know how to convert the reference to the default property:

AssumeType プラグマがなかったら、VB Migration Partner はデフォルトプロパティに対する参照をどのように変換すればよいかわかりません:

```
Dim ctrl As Object

For Each ctrl In Me.Controls
    If TypeOf ctrl Is VB6TextBox Then ctrl.Text = ""
Next
```

Notice that this pragma must have a nonempty scope.

このプラグマは、空ではないスコープを持つ必要があることに注意してください。

AutoDispose *mode*

Specifies how fields pointing to disposable objects must be converted. The *mode* argument can be *No*, *Yes*, or *Force*. If equal to *Yes* or omitted, all Set x = Nothing statements are converted into calls to the SetNothing6 method; if

equal to *Force*, classes that contain disposable field is marked as IDisposable and all such fields are disposed of in the Dispose method; if equal to *No*, disposable objects aren't handled in any special way (this setting represents the default behavior and can be useful to override a pragma with a broader scope). The AutoDispose pragma can be applied to the project, file, class, or individual field or variable level.

このプラグマは、disposable オブジェクトを指し示すフィールドを、どのように変換しなければならないかを指定します。*mode* 引数は *No*、*Yes*、*Force* を指定することが可能です。引数が *Yes* か省略された場合、すべての「Set x = Nothing」ステートメントは、SetNothing6 メソッドの呼び出しに変換されます。また、引数が *Force* であれば、disposable フィールドを含むクラスは IDisposable として注釈が付けられ、そのようなフィールドはすべて Dispose メソッドに配置されます。引数が *No* であれば、disposable オブジェクトはいかなる特別な方法でも取り扱われません。（この設定はデフォルトの動作を意味し、より広いスコープのプラグマをオーバーライドするのに役立ちます。）AutoDispose プラグマは、プロジェクトレベル、ファイルレベル、クラスレベルまたは個々のフィールドレベル、変数レベルに適用することができます。

```
'## Rem force disposal of all disposable objects in current class
'## Rem except those of the Test method
'## AutoDispose Force
'## Test.AutoDispose No
```

AutoNew *boolean*

Specifies how auto-instancing object – that is, fields and variables declared with the As New clause – must be converted. By default, such declarations are converted verbatim to VB.NET, even though the VB.NET semantics differs from VB6. If the *mode* argument is True or omitted, then VB Migration Partner generates code that ensures that the VB.NET is identical to VB6.

このプラグマは、自動インスタンス化オブジェクトーいわゆる、As New 節で宣言されたフォールドや変数ーをどのように変換するかを指定します。デフォルトではそのような宣言は、たとえ VB6 の意味に違いがあったとしても、文字通りに VB.NET に変換されます。*mode* 引数が True もしくは省略されている場合は、VB Migration Partner は VB.NET が VB6 と完全に同一になるコードを生成します。

```
'## Rem ensure that Connection and Recordset objects have auto-instancing
semantics
'## Connection.AutoNew
'## Recordset.AutoNew
Public Recordset As New ADODB.Recordset
Public Connection As New ADODB.Recordset
```

The actual code being generated is different for local variables and fields: if the object is declared as a class-level field, VB Migration Partner converts it into a property whose Get block contains code that implements the auto-instancing behavior; if the object is declared as a local variable, all occurrences are wrapper by AutoNew6 method calls, that ensure that the object is instantiated if necessary.

実際に生成されたコードはローカル変数とフィールドでは異なります。オブジェクトがクラスレベルフィールドとして宣言された場合、VB Migration Partner は GetBlock が自動インスタンス化動作を実行するコードを含むプロパティに

変換します。オブジェクトがローカル変数として宣言された場合、全てのオカレンスは AutoNew6 メソッド呼び出しによるラッパーになり、オブジェクトが必要であればインスタンス化され保証されます。

ChangeType *vb6type, nettype*

Specifies that all members of a given VB6 type in the pragma's scope must be converted to a given .NET type. It is useful to convert Variant variables to VB6Variant objects and Control variables to System.Windows.Forms.Control variables (without this pragma, such variables would be migrated as Object variables):

このプラグマは、プラグマのスコープに存在する、特定の VB6 タイプのすべてのメンバーを、特定の.NET タイプに変換するように、指定できます。これは、Variant 変数を VB6Variant オブジェクトに、Control 変数を System.Windows.Forms.Control 変数に変換するときに役に立ちます。(このプラグマがない場合は、それらの変数は、Object 変数として移行されます):

```
'## ChangeType Variant, VB6Variant
'## ChangeType Control, Control
```

AutoProperty *boolean*

Specifies whether public fields should be rendered as properties. If the argument is True (or omitted) then all public fields are replaced by a read-write property with same name. For example, the following VB6 code:

このプラグマは、パブリックフィールドがプロパティとしてレンダリングされる必要があるかどうかを指定します。引数が True(または省略された)場合全てのパブリックフィールドは、同じ名前のリード、ライトプロパティに置き換えられます。例えば以下の VB6 コードを御覧下さい:

```
'## AutoProperty

Public Name As String
Public Widget As New Widget
```

is rendered as follows:

以下のようにレンダリングされます:

```
Public Property Name() as String
    Get
        Return Name_InnerField
    End Get
    Set(ByVal value As String)
        Name_InnerField = value
    End Set
End Property

Private Name_InnerField As String = ""
```

```
Public Property Widget() As Widget
    Get
        If Widget_InnerField Is Nothing Then Widget_InnerField = New Widget()
        Return Widget_InnerField
    End Get
    Set(ByVal value As Widget)
        Widget_InnerField = value
    End Set
End Property

Private Widget_InnerField As Widget
```

Notice that the AutoProperty pragma automatically enforces the auto-instancing (As New) semantics if possible, regardless of whether the variable is under the scope of an AutoNew pragma.

AutoPropety プラグマは自動的に、変数が AutoNew プラグマのスコープの配下にあるかどうかに関わらず、可能であれば自動インスタンス化(As New)を行います。

ContainsVariantArray *bool*

Specifies how arrays contained inside a VB6Variant member should be processed. If the argument is True (or omitted), the array contained inside all the VB6Variant variables under the scope of the pragma is considered to be a Variant array (which has been translated as a VB6Variant array). If False, the contained array is considered as an array of regular objects.

このプラグマは、VB6Variant メンバーに含まれる配列がどのように処理されなければならないかを指定します。引数が True(または省略)の場合、プラグマのスコープの全ての VB6Variant 変数に含まれる配列は、Variant 配列(VB6Variant 配列として変換されます)にみなされます。False の場合、含まれている配列は通常のオブジェクト配列とみなされます。

```
'## project:ContainsVariantArray
```

By default, when VB Migration Partner parses a Variant member that is followed by an index and a dot, it assumes that the member contains an array of regular objects. To see why this pragma can be useful, consider the following VB6 code:

デフォルトで、VB Migration Partner がインデックスとドットがあとに続く Variant メンバーを分析する際、メンバーがレギュラーのオブジェクト配列を含んでいると仮定します。このプラグマがなぜ役に立つかを確認するために、以下の VB6 コードを御覧ください:

```
'## ChangeType Variant, VB6Variant
Sub Test(ByVal arrObj() As Object, ByVal arrVar() As Variant)
    Dim v1 As Variant, v2 As Variant
    V1 = arrObj
```

```
        v2 = arrVar
        v1(0).DoSomething
        v2(0).DoSomething
    End Sub
```

This is how the code is converted to VB.NET if no other pragma is used:

他のプラグマが使用されていない場合、コードは以下のように.NET に変換されます:

```
    Sub Test(ByVal arrObj() As Object, ByVal arrVar() As VB6Variant)
        Dim v1 As VB6Variant, v2 As VB6Variant
        V1 = arrObj
        v2 = arrVar
        v1(0).DoSomething    ' << correct
        v2(0).DoSomething    ' << wrong!
    End Sub
```

The problem in previous code is that the v2 variable contains an array of VB6Variant objects, therefore the reference to v2(0) returns a VB6Variant element which does not expose the DoSomething method. You solve the problem by adding the following pragma inside the Test method:

前述のコードの問題は、v2 変数が VB6Variant オブジェクトの配列を含んでいるということです、したがって、v2(0)の参照が DoSomething メソッドを公開しない VB6Variant 要素を返します。Test メソッド内に、次のプラグマを追加することで、この問題を解決することが出来ます:

```
    '## v2.ContainsVariantArray
```

which causes the following code to be generated

以下のコードが生成されます。

```
    v2(0).Value.DoSomething    ' << correct
```

IMPORTANT:

This pragma has been obsoleted in release 1.11.01. In this and all subsequent releases, the indexing operation on a simple VB6Variant variable always returns a VB6Variant object, therefore VB Migration Partner always appends the ".Value" suffix when the expression is followed by a dot. The ContainsVariantArray pragma is therefore useless and is ignored during the migration process (but no warning is issued).

このプラグマは、リリース 1.11.01 で廃棄されました。これ以降のリリースにおいて、単純な VB6Variant 変数のインデックス操作は常に VB6Variant オブジェクトを返します。したがって、式の後にドットが続くとき、VB Migration Partner は常に「.Value」接尾語を追加します。 ContainsVariantArray プラグマは役に立たず、変換プロセスの間、無視されます(警告は発行されません)。

DeclareImplicitVariables *boolean*

Specifies whether VB Migration Partner should generate one Dim statement for each local variable that isn't declared explicitly. If the argument is True or omitted, all local variables in the converted VB.NET methods under the pragma's scope are declared explicitly.

このプラグマは、VB Migration Partner が、明示的に宣言されない各ローカル変数のために、1 つの Dim ステートメントを生成するかどうか指定します。引数が True か省略された場合、プラグマのスコープ下にある、変換された VB.NET メソッド内の、全てのローカル変数が明示的に宣言されます。

```
'## Rem generate code for implicitly-declared variables in current file
'## InsertStatement Option Explicit On
'## DeclareImplicitVariables
```

FixParamArray *maxargs*

Specifies whether VB Migration Partner should generate a method overload for methods that take a ParamArray argument and that modify an element of the array argument inside the method. The method overload uses Optional ByRef parameters instead of one single ParamArray parameter, and the *maxargs* value specifies the maximum number of arguments that you expect. For example, assume that you have the following VB6 code:

このプラグマは、VB Migration Partner が、ParamArray 引数を使い、メソッド内部の配列引数の要素を変更するメソッドに対して、メソッドオーバーロードを生成するべきかどうかを指定します。メソッドオーバーロードは 1 つの ParamArray 引数の代わりに、Optional ByRef 引数を使います、そして、*maxargs* の値は、想定される引数の最大数を指定します。たとえば、以下の VB6 コードを御覧下さい。:

```
'## FixParamArray 3
Sub Increment(ParamArray arr() As Variant)
    Dim i As Integer
    For i = 0 To UBound(arr)
        arr(i) = arr(i) + 1
    Next
End Sub
```

Notice that the elements of the *arr* ParamArray array are modified inside the method and these changes are propagated back to callers under VB6 but not under VB.NET. However, the FixParamArray pragma causes the following code to be generated:

VB6 では、ParamArray 配列の要素 *arr* はメソッドの中で変更され、これらの変更が呼び出し元に反映されますが、VB.NET ではそうならないことに注目してください。しかし、FixParamArray プラグマは以下のコードを生成させます:

```
Public Sub Increment(Optional ByRef arr0 As Object = Nothing, _
    Optional ByRef arr1 As Object = Nothing, _
    Optional ByRef arr2 As Object = Nothing)
```

```
        Dim arr() As Object = GetParamArray6(arr0, arr1, arr2)
        Try
            Increment(arr)
        Finally
            SetParamArray6(arr, arr0, arr1, arr2)
        End Try
    End Sub

    Public Sub Increment(ByVal ParamArray arr() As Object)
        Dim i As Short
        For i = 0 To UBound6(arr)
            arr(i) += 1
        Next
    End Sub
```

The neat effect is that any call to the Increment method that takes 3 arguments or fewer will use the first overload, which ensures that changes to any argument are correctly propagated back to callers even in VB.NET.

この素晴らしい効果は、3 個以下の引数を取る Increment メソッドのどの呼出も、最初のオーバーロードを使用することです。そして、この最初のオーバーロードにより、VB.NET においてさえ、引数の変更が呼出元に正しく戻されます。

If the *maxargs* argument is 0 or omitted, no overloaded method is generated. (This behavior is useful to override another FixParamArray pragma with broader scope.) Values higher than 20 are rounded down to 20.

maxargs 引数が 0 または省略の場合はオーバーロードメソッドを生成しません。(この動作はより広いスコープの FixParamArray プラグマを上書きするのに便利です。)20 より多い値は 20 で切り捨てられます。

InferType *mode, includeParams*

Specifies whether VB Migration Partner should try to infer the type of the local variables, class fields, functions, and properties that are under the scope of this pragma. The mode argument can be one of the following: **No** if type inference is disabled; **Yes** if type inference is enabled only for members that lacks an explicit As Variant clause and for local variables that are implicitly declared; **Force** if type inference is enabled for all Variant members. The second argument is True if type inference is extended to method parameters.

このプラグマは、VB Migration Partner がこのプラグマのスコープにある、ローカル変数、クラスフィールド、関数、及びプロパティのタイプをどのように推測しなければならないかを指定します。mode 引数は、次のいずれかを指定できます:**No** タイプの推測は適用されない。**Yes** 明示的な As Variant 節が欠落したメンバーか、暗黙的に宣言されるローカル変数のみにタイプの推測が適用される。**Force** 全ての Variant メンバーにタイプの推測が適用される。二つ目の引数が True であれば、タイプの推測はメソッドパラメータも対象とします。

VB Migration Partner is capable, in most cases, to correctly deduce a less generic data type for Variant local variables, parameters, class fields, functions and properties. This applies both to members that were explicitly

declared as Variant and to members that lacked an explicit As clause. To see how this feature works, consider the following VB6 code:

ほとんどの場合 VB Migration Partner は、バリアント型のローカル変数、パラメータ、クラスフィールド、関数、プロパティの汎用的なデータ型を、正しく推測することが出来ます。これは、Variant と明らかに宣言されたメンバーと、明示的な As 節を欠いていたメンバーに適用されます。この機能がどのように働くかは、以下の VB6 コードを見てください。

```
'## DeclareImplicitVariables True
'## InferType Force, True

Private m_Width As Single

Property Get Width ()
    Width = m_Width
End Property

Property Let Width (value)
    m_Width = value
End Property

Sub Test(x As Integer, frm)
    Dim v1 As Variant, v2
    v1 = x * 10
    v2 = Width + x
    Set frm = New frmMain
    res = frm.Caption
End Sub
```

Here's the migrated VB.NET code:

VB.NET に移行すると以下のようになります:

```
Private m_Width As Single

Public Property Width() As Single
    Get
        Return m_Width
    End Get
    Set(ByVal value As Single)
        m_Width = value
    End Set
End Property
```

```
    Public Sub Test(ByRef x As Short, ByRef frm As Form1)
        ' UPGRADE_INFO (#0561): The 'frm' symbol was defined without an explicit
"As" clause.
        ' UPGRADE_INFO (#05B1): The 'res' variable wasn't declared explicitly.
        Dim res As String = Nothing
        Dim v1 As Integer
        ' UPGRADE_INFO (#0561): The 'v2' symbol was defined without an explicit "As"
clause.
        Dim v2 As Single
        v1 = x * 10
        v2 = Width + x
        frm = New Form1()
        res = frm.Caption
    End Sub
```

Extending the pragma scope to method parameters isn't always a good idea, because the current version of VB Migration Partner infers the type of parameters by looking at assignments *inside* the method and doesn't account for implicit assignments that result from method calls. For this reason, the type inferred during the migration process might not be correct in some cases.

VB Migration Partner の現在のバージョンは、メソッド*内での*割り当てを調べることでタイプの推測を行い、メソッドの呼び出しからの暗黙の割り当てを考慮していないため、プラグマのスコープをメソッドパラメータに拡張するのは、常に最適な方法というわけではありません。この理由から、移行プロセス中に推測されるタイプは、場合によっては正しくないかもしれません。

Important note:
Type inference isn't an exact science and you should always double-check that the data type inferred by VB Migration Partner is correct. Even better, we recommend that you use the InferType pragma only during in early migration attempts and then you later assign a specific type to members by editing the original VB6 code or by means of the SetType pragma.

タイプの推測は万能ではありませんので、VB Migration Partner によって推測されるデータ型が正しいことを常に再確認する必要があります。更には InferType プラグマは変換の初期段階においてのみ使用することをお勧めします。その後、元の VB6 コードを編集するか、SetType プラグマによってメンバーに特定のタイプを割り当ててください。

LateBoundMethods *methodregex, indexregex, varregex*

Specifies which late-bound members should be considered as methods with one or more ByRef arguments. The first *methodregex* argument is a regular expression that specifies which member names are to be considered as the names of methods with ByRef arguments (use ".+" to indicate "all members"). The second *indexregex*argument is a regex that identifies all (0-based) numeric indexes of ByRef parameters of indicated methods (if omitted, it defaults to "¥d+", which means "all parameters"). The third parameter is optional and is considered as a regular

expression that specifies which Variant and Object fields and variables the pragma applies to (if omitted, it defaults to ".+" and therefore all the fields and variables in the pragma's scope are affected).

このプラグマは、どの遅延バインディングメンバーが、一つ以上の ByRef 引数を持つメソッドとして考慮されるべきかを指定します。最初の methodregex 引数は、どのメンバーの名前が ByRef 引数を持つメソッドの名前としてみなすべきかを指定する正規表現です。(全てのメンバーを示す、".+"を使用して下さい。) 第 2 の indexregex 引数は、示されたメソッドの ByRef パラメータのすべての(0 ベースの)数値インデックスを特定する正規表現です。(省略された場合、デフォルトで、全てのパラメータを意味する"¥d+"になります。)第 3 の引数は任意であり、プラグマが、どの Variant、Object フィールド、および変数に適用されるかを指定する正規表現としてみなされます。(省略された場合、デフォルトで".+"となり、プラグマのスコープのすべてのフィールドと変数が影響を受けます)

```
' all the late-bound members in current project are to be considered as
' methods that take ByRef arguments
'## project:LateBoundMethods ".+"
```

This pragma becomes important in applications that use late-binding quite extensively and accounts for a subtle but important difference between VB6 and VB.NET. Under VB6, passing a writable property to a ByRef parameter of a method does ***not*** affect the property (because behind the scenes VB6 is actually passing the return value from the Property Get block). Vice versa, passing a writable property to a ByRef parameter ***does***modify the property value on exiting the method under VB.NET.

このプラグマはかなり広範囲に遅延バインディングを使い、VB6 と VB.NET の微妙であるが、重要な違いの主な原因となるアプリケーションでは重要になります。VB6 では、メソッドの ByRef パラメータに書き込み可能なプロパティを渡すのは、プロパティに影響を与えません(実際には、VB6 は裏で、"Property Get"ブロックの戻り値を渡しているからです。) 逆に、VB.NET では、書き込み可能なプロパティを ByRef パラメータへ渡すと、メソッドを出る時にプロパティの値を修正します。

Consider the following VB6 code:

以下の VB6 コードを御覧ください。

```
Sub Test(ByRef lbl As Label, ByVal txt As TextBox, ByVal wi As Widget, ByVal obj As Object)
    Set obj = wi
    wi.ModifyValues lbl.Caption, txt.Text
    obj.ModifyValues lbl.Caption, txt.Text
End Sub
```

If the ModifyValue in the Widget class takes one ByRef argument, this is how VB Migration Partner translates the code to VB.NET:

Widget クラスでの ModifyValue が 1 つの ByRef 引数をとる場合、このようにして、VB Migration Partner はコードを VB.NET に変換します:

```
Sub Test(ByRef lbl As VB6Label, ByVal wi As Widget, ByVal obj As Object)
```

```
        Set obj = wi
        wi.ModifyValue(ByVal6(lbl.Caption), ByVal6(txt.Text))
        obj.ModifyValue(lbl.Caption, txt.Text)
    End Sub
```

The ModifyValue method can modify the *lbl.Caption* and *txt.Text* writable properties under VB. NET and that this modification would break functional equivalence with the original VB6 code. VB Migration Partner knows how to work around this difference and correctly wraps the two property references inside a ByVal6 method when calling the *wi.ModifyValue* method.

ModifyValue メソッドは VB.NET で lbl.Caption と txt.Text の書き込み可能なプロパティを変更できます、そして、この変更は元の VB6 コードで機能的な等価性を壊すでしょう。 wi.ModifyValue メソッドを呼ぶとき、VB Migration Partner はどのようにこの違いを回避するかを把握していて、ByVal6 メソッド内の正確に 2 つのプロパティの参照箇所を包みます。

VB Migration Partner can carry out this correctly because the *ModifyValue* is referenced using early-binding (i.e. the *wi* member has a definite type) and it is possible to detect that the *lbl.Caption* and *txt.Text* values are being passed to a ByRef parameter. Conversely, the *obj.ModifyValue* call uses late binding and VB Migration Partner can't decide whether the called method takes one or more ByRef parameters.

ModifyValue は、初期バインディングを使用して参照されているため（つまり *wi* メンバーが明確なタイプを持っている）VB Migration Partner はこれを正しく行うことができ、*lbl.Caption* と *txt.Text* 値が ByRef パラメータに渡されることを検出することが可能です。逆に、*obj.ModifyValue* 呼び出しは、遅延バインディングを使用し、VB Migration Partner は呼び出されたメソッドが、1 つまたは複数の ByRef パラメータを取るかどうかを判断することができません。

In this scenario you can use the LateBoundProperties pragma to let VB Migration Partner know which late-bound members are to be considered as methods that take one or more ByRef parameters. Here's how the pragma might be used in this specific example:

このシナリオでは、どの遅延バインディングメンバーが一つ以上の ByRef パラメータをとるメソッドとしてみなされるということを、VB Migration Partner に通知するために、LateBoundProperties プラグマを使うことができます。この特定の例で、プラグマをどのように使用するかを以下に示します：

```
    Sub Test(ByRef lbl As Label, ByVal txt As TextBox, ByVal wi As Widget, ByVal
obj As Object)
        '## obj.LateBoundMethods "ModifyValues"
        Set obj = wi
        wi.ModifyValues lbl.Caption, txt.Text
        obj.ModifyValues lbl.Caption, txt.Text
    End Sub
```

which produces the following code:

そして以下のコードを生成します：

```
Sub Test(ByRef lbl As VB6Label, ByVal wi As Widget, ByVal obj As Object)
    Set obj = wi
    wi.ModifyValue(ByVal6(lbl.Caption), ByVal6(txt.Text))
    obj.ModifyValue(ByVal6(lbl.Caption), ByVal6(txt.Text))
End Sub
```

Consider that the pragma's first argument is actually a regular expression, which allows you to specify multiple methods in just one pragma. (This is necessary because you can associate only one LateBoundMethods pragma to a given variable.) Therefore you might write something like

プラグマの最初の引数は、実際には 1 つのプラグマで複数のメソッドを指定することができる、正規表現だと考えてください。(所定の変数に対し、ただ 1 つの LateBoundMethods プラグマを関連づけることができるので、必要になります。)よって、次のような記述になります。

```
'## obj.LateBoundMethods "^(ModifyValues|ProcessInfo)$"
```

or just assume that all exposed members are to be considered as methods that can modify their arguments:

あるいは、すべての公開されているメンバーが、それらの引数を変更できるメソッドとして考えられると仮定します:

```
'## obj.LateBoundMethods ".+"
```

The second optional argument of the LateBoundMethods pragma is a regular expression that is applied to the 0-based numeric index of the argument being analyzed. If omitted, this regex defaults to "¥d+", therefore all parameters are considered to be defined with ByRef. For example, if you knew that only the second argument of the Widget.ModifyValues method is defined with ByRef, you can avoid useless ByVal6 insertions using this pragma:

LateBoundMethods プラグマの 2 番目の任意の引数は、分析される引数の 0 から始まる数値インデックスに適用される正規表現です。省略された場合は、この正規表現のデフォルトは"¥d+"となり、すべてのパラメータが ByRef で定義されていると考えられます。例えば、Widget.ModifyValues メソッドの 2 番目の引数だけが ByRef として定義されているのがわかっている場合、このプラグマを使用することで不要な ByVal6 の挿入を避けることができます:

```
'## obj.LateBoundMethods "ModifyValues", "^1$"
```

If just the 2nd and 3rd parameters are defined with ByRef you need this pragma:

2 番目と 3 番目のパラメータが ByRef で定義されている場合は、このプラグマを必要とします:

```
'## obj.LateBoundMethods "ModifyValues", "^(1|2)$"
```

and so on.
The third argument of the LateBoundMethods pragma allows you to apply the pragma to multiple variables and fields under the pragma's scope, without having to specify a different pragma for each one of them. For example, consider the following VB6 code:

LateBoundMethods プラグマの 3 番目の引数は、個別に異なるプラグマを指定することなく、プラグマのスコープ下の複数の変数およびフィールドにプラグマを適用することができます。たとえば、次の VB6 コードを御覧下さい:

```
Sub Test(ByVal obj As Object, ByVal obj2 As Object, ByVal obj3 As Object)
    Set obj = New Widget
    Set obj2 = New Widget
    Set obj3 = New Person
    obj.ModifyValues lbl.Caption, txt.Text
    obj2.ModifyValues lbl.Caption, txt.Text
    obj3.ModifyValues lbl.Caption, txt.Text
End Sub
```

If both the Widget and Person classes expose a ModifyValues method that takes ByRef arguments, you might use the following pragma to affect all of them:

両方の Widget クラスと Person クラスが、ByRef 引数を取る ModifyValues メソッドを公開する場合は、それらのすべてに影響を与える、次のプラグマを使用することが出来ます:

```
Sub Test(ByVal obj As Object, ByVal obj2 As  Object, ByVal obj3 As Object)
    '## LateBoundMethods "ModifyValues"
    Set obj = New Widget
    Set obj2 = New Widget
    Set obj3 = New Person
    obj.ModifyValues lbl.Caption, txt.Text
    obj2.ModifyValues lbl.Caption, txt.Text
    obj3.ModifyValues lbl.Caption, txt.Text
End Sub
```

However, if you don't want to apply the pragma to *obj3*, you would be forced to specify a distinct pragma for*obj* and *obj2*. The pragma's third argument allows you to write more concise code:

obj3 にプラグマを適用したくない場合は、*obj* と *obj2* に異なったプラグマを指定する必要があります。プラグマの 3 番目の引数で、より簡潔なコードを書くことができます:

```
'## LateBoundMethods "ModifyValues", "¥d+", "^(obj|obj2)$"
```

Notice that VB Migration Partner applies the two regular expressions using the Regex.IsMatch method, therefore the following pragma wouldn't achieve the intended effect:

VB Migration Partner は、Regex.IsMatch メソッドを使用することで 2 つの正規表現を適用しています。したがって、以下のプラグマが意図された結果を得られないことに注意してください:

```
'## LateBoundMethods "ModifyValues", "¥d+", "(obj|obj2)"
```

because it would also match the *obj3* member name.

それはまた、*obj3* メンバー名を一致させるためです。

If the VB6 code consistently uses a naming convention for its late-bound members, the pragma's second argument can be quite useful. For example, if all the Variant and Object variables that are meant to store a reference to an ADODB.Connection have a leading "conn" in their name, you might use the following pragma:

VB6 のコードが、遅延バインディングのメンバーに一貫した命名規則を使用している場合は、そのプラグマの 2 番目の引数は非常に役に立ちます。例えば、ADODB.Connection への参照を格納する、すべての Variant と Object 変数の名前の先頭が、"conn"となっているなら、以下のプラグマを利用することが出来ます:

```
'## project:LateBoundMethods "Execute", "^1$", "^conn"
```

to indicate that the second argument (i.e. the *RecordsAffected* argument) of the Execute method can be modified on return from the call.

上記は、実行メソッドの 2 番目の引数(例:*RecordAffected* 引数)が I 呼び出しから戻る際に変更されうるということを意味します。

Finally, you can also change VB Migration Partner's default behavior by considering all late-bound names as references to methods with ByRef arguments:

最後に、ByRef 引数を持つメソッドへの参照として全ての遅延バインディング名を考慮することによって、VB Migration Partner のデフォルトの動作を変更することが出来ます。

```
'## project:LateBoundMethods ".+", ".+"
```

or more simply with

より簡潔に

```
'## project:LateBoundMethods ".+"
```

Notice that a LateBoundMethods pragma with a narrow scope shadows all the LateBoundMethods pragmas at a broader scope. For example, the above project-level pragma doesn't apply to methods where another LateBoundMethods pragma is defined.

スコープの狭い LateBoundMethods プラグマが、より広いスコープの LateBoundMethods プラグマに優先することに注意してください。 例えば、上記のプロジェクトレベルプラグマは別の LateBoundMethods プラグマが定義されるメソッドに適用されません。

LateBoundProperties *propregex, varregex*

Specifies which late-bound members should be considered as writable properties. The first *propregex* argument is a regular expression that specifies which member names are to be considered as the names of writable properties (use ".+" to indicate "all members"). The second parameter is optional and is considered as a regular expression that specifies which Variant and Object fields and variables the pragma applies to (if omitted, it defaults to ".+"

and therefore all the fields and variables in the pragma's scope are affected). This pragma can be applied to the project, file, method, and variable level:

このプラグマは、どの遅延バインディングのメンバーが書き込み可能なプロパティであるとみなすかを指定します。最初の *propregex* 引数は、どのメンバー名が書込み可能なプロパティ名とみなされるかを指定する正規表現です。(「すべてのメンバー」を示す場合、「.+」を使います。)第 2 のパラメータは任意で、プラグマがどの Variant と Object のフィールドと変数をあてはめるかを指定する正規表現と考えてください。(省略された場合、デフォルトで「.+」となり、プラグマのスコープのすべてのフィールドと変数が影響を受けます。)このプラグマは Project、File、メソッド、変数レベルに適用できます。

```
    ' all the late-bound members in current project are to be considered as writable properties
    '## project:LateBoundProperties ".+"
```

This pragma becomes important in applications that use late-binding quite extensively and accounts for a subtle but important difference between VB6 and VB.NET. Under VB6, passing a writable property to a ByRef parameter of a method does ***not*** affect the property (because behind the scenes VB6 is actually passing the return value from the Property Get block). Vice versa, passing a writable property to a ByRef parameter ***does***modify the property value on exiting the method under VB.NET.

このプラグマはかなり広範囲に遅延バインディングを使い、VB6 と VB.NET の微妙であるが、重要な違いの主な原因となるアプリケーションでは重要になります。VB6 では、メソッドの ByRef パラメータに書き込み可能なプロパティを渡すのは、プロパティに影響を与え***ません***。(実際には、VB6 は裏で、「Property Get」ブロックの戻り値を渡しているからです。) 逆に、VB.NET では、書き込み可能なプロパティを ByRef パラメータへ渡すと、メソッドを出る時にプロパティの値を修正***します***。

Consider the following VB6 code:

以下の VB6 コードを御覧ください。

```
    Sub Test(ByVal lbl As Label, ByVal obj As Object)
        DoSomething lbl.Caption, obj.Caption
    End Sub

    Sub DoSomething(ByRef str As String)
        str = UCase(str): str2 = UCase(str2)
    End Sub
```

If no pragma is added, this is how VB Migration Partner translates the code to VB.NET:

プラグマが全く加えられない場合、VB Migration Partner は以下のような VB.NET コードに変換します:

```
    Sub Test(ByVal lbl As VB6Label, ByVal obj  As Object)
        DoSomething(ByVal6(lbl.Caption), obj.Caption)
    End Sub
```

```
Sub DoSomething(ByRef str As String, ByRef str2 As String)
    str = UCase(str): str2 = UCase(str2)
End Sub
```

As you see, VB Migration Partner knows how to work around this difference and correctly wraps the *lbl.Caption*property reference inside a ByVal6 method when calling the method. This is possible because the property is referenced using early-binding (i.e. the *lbl* member has a definite type) VB Migration Partner can detect that a writable property is being passed. Conversely, the obj.Caption member uses late binding and VB Migration Partner can't decide whether Caption is a reference to a writable property and omits the necessary ByVal6 code.

ご覧のとおり、VB Migration Partner はこの違いに対処するかを把握し、メソッドを呼び出す際に、正しく ByVal6 メソッド内に *lbl.Caption* のプロパティ参照を包含します。これはプロパティが初期バインディングを使用して参照されている(つまり *lbl* のメンバーは、明確な型を持っている)為に可能であり、VB Migration Partner は書き込み可能なプロパティが渡されていることを検出することができます。逆に、obj.Caption メンバーが遅延バインディングを使用し、VB Migration Partner は Caption が書込み可能なプロパティの参照かの判断が出来ず、必要な ByVal6 コードが省略されます。

In this scenario you can use the LateBoundProperties pragma to let VB Migration Partner know which late-bound members are writable properties. In this specific case you might use this pragma

このシナリオでは、どの遅延バインディングのメンバーが、書き込み可能なプロパティであるかを VB Migration Partner に知らせるのに LateBoundProperties プラグマを使用できます。この特定のケースでは、このプラグマを使用する場合があります。

```
Sub Test(ByVal lbl As Label, ByVal obj As Object)
    '## obj.LateBoundProperties "Caption"
    DoSomething lbl.Caption, obj.Caption
End Sub
```

However, consider that the pragma's first argument is actually a regular expression, which allows you to specify multiple properties in just one pragma. (This is necessary because you can associate only one LateBoundProperties pragma to a given variable.) Therefore you might write something like

プラグマの最初の引数は、実際には 1 つのプラグマで複数のプロパティを指定することができる、正規表現だと考えてください。(所定の変数に対し、1 つの LateBoundProperties プラグマのみ結びつけることができるので、これが必要になります。)よって、最初の引数には以下のような記述になります。

```
'## obj.LateBoundProperties "^(Caption|Left|Top|Width|Height)$"
```

or just assume that all exposed members are to be considered as writable properties:

あるいは、すべての公開されているメンバーは、書き込み可能なプロパティであるとみなされると仮定してください:

```
'## obj.LateBoundProperties ".+"
```

The second optional argument of the LateBoundProperties pragma allows you to apply the pragma to multiple members under the pragma's scope, without having to specify a different pragma for each one of them. For example, consider the following VB6 code:

LateBoundProperties プラグマの 2 番目の任意の引数は、プラグマのスコープ内の複数のメンバーに対し、個別にプラグマを指定することなくプラグマを適用することが出来ます。以下の VB6 コードを御覧ください。

```
Sub Test(ByVal obj As Object, ByVal obj2 As Object, ByVal obj3 As Object)
    DoSomething obj.Caption, obj2.Caption, obj3.Caption
End Sub
```

If you can assume that all three parameters represent a control you might use the following pragma to affect all of them:

3 つの全てのパラメータが、コントロールを表していると想定できる場合、それらの全てに影響を与えるために、次のプラグマを使用することが出来ます。

```
Sub Test(ByVal obj As Object, ByVal obj2 As Object, ByVal obj3 As Object)
    '## LateBoundProperties "Caption"
    DoSomething obj.Caption, obj2.Caption, obj3.Caption
End Sub
```

However, if you don't want to apply the pragma to *obj3*, you would be forced to specify a distinct pragma for*obj* and *obj2*. The pragma's second argument allows you to write more concise code:

ただし、*obj3* にプラグマを適用したくない場合には、*obj* と *obj2* へ個別にプラグマを指定する必要があります。プラグマの 2 番目の引数には、より簡潔なコードを書きます:

```
'## LateBoundProperties "Caption", "^(obj|obj2)$"
```

Notice that VB Migration Partner applies the two regular expressions using the Regex.IsMatch method, therefore the following pragma wouldn't achieve the intended effect:

VB Migration Partner は、Regex.IsMatch メソッドを使用する 2 つの正規表現を適用しています。したがって、以下のプラグマは意図された結果を得られないということに注意してください:

```
'## LateBoundProperties "Caption", "(obj|obj2)"
```

because it would also match the *obj3* member name.

なぜなら、*obj3* メンバー名を一致させるからです。

If the VB6 code consistently uses a naming convention for its late-bound members, the pragma's second argument can be quite useful. For example, if all the Variant and Object variables that are meant to store a reference to an ADODB.Connection have a leading "conn" in their name, you might use the following pragma:

VB6 のコードが、遅延バインディングのメンバーに一貫した命名規則を使用している場合は、プラグマの 2 番目の引数は非常に役に立ちます。例えば、ADODB.Connection への参照を格納する、すべての Variant と Object 変数の名前の先頭が、“conn”となっているなら、以下のプラグマを利用することが出来ます:

```
'## project:LateBoundProperties "(ConnectionString|CursorLocation)", "^conn"
```

Finally, you can also change VB Migration Partner's default behavior by having all references to all late-bound members wrapped in ByVal6 method, as follows:

最後に、以下のように、参照される全ての遅延バインディングメンバーを ByVal6 メソッドで包むように、VB Migration Partner のデフォルトの動作を変更することが出来ます:

```
'## project:LateBoundProperties ".+", ".+"
```

or more simply with

より簡潔にすると以下になります

```
'## project:LateBoundProperties ".+"
```

Notice that a LateBoundProperties pragma with a narrow scope shadows all the LateBoundProperties pragmas at a broader scope. For example, the above project-level pragma doesn't apply to methods where another LateBoundProperties pragma is defined.

スコープの狭い LateBoundProperties プラグマが、より広いスコープの LateBoundProperties プラグマより優先されることに注意してください。 例えば、上記のプロジェクトレベルプラグマは別の LateBoundProperties プラグマが定義されるメソッドに適用されません。

LateBoundVariantMembers *propregex, varregex, canReturnEmpty*

Specifies which late-bound members should be considered as properties or methods that return a Variant value. The first *propregex* argument is a regular expression that specifies which member late-bound names can return a Variant value (use ".+" to indicate "all members"). The second parameter is optional and is considered as a regular expression that specifies which Variant and Object fields and variables the pragma applies to (if omitted, it defaults to ".+" and therefore all the fields and variables in the pragma's scope are affected). The third Boolean parameter is also optional and should be set to True if the Variant being returned can be the special Empty value.

このプラグマは、どの遅延バインディングのメンバーが、Variant 値を返すプロパティまたはメソッドと見なすべきかを指定します。最初の *propregex* 引数は、どの名前の遅延バインディングのメンバーが、Variant 値を返すことができるかについて指定する、正規表現です。(「すべてのメンバー」を指定する場合は、「.+」を使います。)第 2 のパラメータは任意で、プラグマがどの Variant と Object のフィールドと変数をあてはめるかを指定する正規表現と考えてください。(省略された場合、デフォルトで「.+」となり、プラグマのスコープのすべてのフィールドと変数が影響を受けます。)第 3 のブールのパラメータも任意ですが、もどされる Variant に特別な Empty 値が設定される可能性がある場合、True を設定する必要があります。

```
        ' all the late-bound members in current project are to be considered as
properties
        ' or methods that can return a Variant value
        '## project:LateBoundProperties ".+"
```

This pragma can be necessary to prevent a few runtime exceptions when working with VB6Variant values and therefore is always used together with a ChangeType pragma like this one:

このプラグマは、VB6Variant 値を使用して動作させる際のいくつかのランタイムエラーを防止するために設定が必要な場合があります。よって、常に以下のような ChangeType プラグマと一緒に使用されます。

```
        '## project:ChangeType Variant, VB6Variant
```

If the **LateBoundVariantMembers** pragma is used, all methods and properties (within the scope of the pragma) invoked in late-bound mode – that is, through an Object or VB6Variant variable – are assumed to return a VB6Variant value. In practice, this pragma causes the invocation to be wrapper inside a **CVar6** method in the following cases:

LateBoundVariantMembers プラグマが使用されている場合、オブジェクトか VB6Variant 変数を通過する遅延バインディングで起動された全てのメソッドとプロパティ(プラグマのスコープ内)が VB6Variant 値を戻すとみなします。実際には、このプラグマは以下のケースのような **CVar6** メソッド内部のラッパーになる起動を引き起こします。

- The member appears to the left or right of a comparison operator (e.g. = or <> operators)
  メンバーが比較演算子の右か左に現れる場合（例　= または　<> 演算子）
- The member appears to the left or right of a math operator, and the other operand isn't a VB6Variant value
  メンバーが数学演算子の右か左に現れる場合、かつ他のオペランドが VB6Variant 値ではない場合
- The return value from the member is being assigned to an array or an object variable
  メンバーからの戻り値が配列またはオブジェクト変数に割り当てられている場合
- The return value from the member is being assigned to a scalar value (only if pragma's 3rd argument is True)
  メンバーからの戻り値がスカラー値に割り当てられている場合（プラグマの 3 番目の引数が True の場合のみ）

MergeInitialization *mode*

Specifies whether a specific local variable (or all local variables in a method) must be merged with the first statement that assigns it a value. By default, VB Migration Partner merges a variable declaration with its first assignments only if the value being assigned is constant and if the variable isn't used in any statement after the declaration and before the assignment. You can use this pragma to inform VB Migration Partner that it is safe to do the merge even if the value being assigned isn't constant. For example, consider this VB6 code:

このプラグマは、特定のローカル変数(あるいはメソッド内のすべてのローカル変数)に値を割り当てる最初のステートメントにマージしなければならないかどうかを指定します。VB Migration Partner のデフォルトの動作は、割り当

てられている変数が定数で、かつ変数宣言後、割り当て前にどのステートメントでも使用されていない場合、最初の割り当ての変数宣言をマージします。割り当てられている値が定数でない場合でも、以下のプラグマを使用することで、VB Migration Partner にマージをすることが安全であると通知することが出来ます。以下の VB6 コードを御覧ください。

```
Sub Test(value As Integer)
    '## MergeInitialization Force
    Dim x As Integer, y As Integer
    x = value * 2
    y = x * value * 3
    ' ...
End Sub
```

The values being assigned aren't constant, therefore by default VB Migration Partner wouldn't attempt to merge the declaration and the assignment statements. Thanks to the MergeInitialization pragma, the code generator produces the following code:

割り当てられている値は、定数ではありません。よって、デフォルトでは、VB Migration Partner は宣言と代入文のマージを試みません。 MergeInitialization プラグマのおかげで、コードジェネレータは、以下のコードを生成します:

```
Sub Test(value As Short)
    Dim x As Short = value * 2
    Dim y As Short = x * value * 3
    ' ...
End Sub
```

Other possible values for this pragma are **No** (always disable the merging) or **Yes** (apply the merging if it is safe to do so). The Yes value represents the default behavior, but this setting can be necessary to override the effect of another MergeInitialization pragma with a broader scope:

このプラグマの他の設定できる値として、**No**(常に、マージしない)または **Yes**(安全であるならば、マージする)があります。Yes はデフォルトの動作を意味します。しかしこの設定はより広いスコープで使われる MergeInitialization プラグマの効果を無効にするために必要になります。

```
' Disable initialization merging for all variables except n1
'## MergeInitialization No
'## n1.MergeInitialization Yes
```

Note:

in a method that contains one or more Gosub statements that are converted to separate methods because of a ConvertGosubs pragma, this pragma is ignored and the variable initialization merging feature is disabled.

ConvertGosubs プラグマによって、別々のメソッドに変換された1つ以上の Gosub ステートメントを含むメソッドでは、このプラグマは無視されます、そして、変数の初期化マージ機能は無効にされます。

SetName *membername*

Specifies whether the current project, class, method, property or variable must have a different name in the converted VB.NET application. It can be useful to avoid name clashes with reserved VB.NET keywords or names of .NET Framework classes and methods. Consider the following VB6 code:

このプラグマは、現在のプロジェクトか、クラスか、メソッドか、プロパティか変数が、変換後の VB.NET アプリケーションで異なる名前が必要であるかを指定します。このプラグマは、予約された VB.NET キーワード、または.NET Framework のクラスの名前とメソッドの名前との衝突を避けるために役立つことがあります。次の VB6 のコードを御覧ください。

```
' ... (inside the Console VB6 class)
'## SetName ConsoleNET
'## Id.SetName Identifier
Public Id As String

Sub AddHandler()
    '## SetName AddEventHandler
    '
End Sub

Sub Test()
    Dim c As New Console
    c.Id = "abc"
    c.AddHandler
End Sub
```

Here's how the code inside the Test method is converted to VB.NET:

Test メソッド内のコードは、以下のように VB.NET に変換されます:

```
Sub Test()
    Dim c As New ConsoleNET
    c.Identifier = "abc"
    c.AddEventHandler()
End Sub
```

ShiftIndexes *applytovariables, delta1 [, delta2, delta3]*

Specifies whether the indexes of the element of an array should be adjusted by a given value.
The*applytovariables* argument is False if only constant indexes should be adjusted, True if all indexes should be adjusted; *delta1* is the value that must be subtracted from the first index; *delta2* and *delta3* are the values that must be subtracted from the second and third index, respectively.

このプラグマは、配列の要素のインデックスが、所定の値によって調整すべきかどうか指定します。*applytovariables* 引数には、定数インデックスのみを調整する場合は False を、すべてのインデックスが調整されるべきであれば True を設定します。*delta1* は、最初のインデックスから減算する必要がある値です；*delta2* と *delta3* は、それぞれ第 2 および第 3 のインデックスから減算する必要がある値です。

This pragma is typically used together with the ArrayBounds pragma. Consider the following VB6 code:

このプラグマは、通常、ArrayBounds プラグマと一緒に使用されます。次の VB6 のコードを御覧ください。

```
Sub Test()
    '## ArrayBounds Shift
    '## primes.ShiftIndexes False, 1
    Dim primes(1 To 10) As Integer
    primes(1) = 1: primes(2) = 2: primes(3) = 3: primes(4) = 5: primes(5) = 7

    '## powers.ShiftIndexes False, 1
    Dim powers(1 To 10) As Double
    powers(1) = 1
    Dim i As Integer
    For i = LBounds(powers) + 1 To UBound(powers)
        powers(i) = powers(i - 1) * 2
    Next
End Sub
```

The ShiftIndexes pragmas ensure that the array elements whose index is constants are scaled by 1 when this code is converted to VB.NET:

ShiftIndexes プラグマは、このコードが VB.NET に変換されるとき、1でスケールされた定数であるインデックスをもつ配列要素であることを確実にします。

```
Sub Test()
    Dim primes(9) As Short
    primes(0) = 1: primes(1) = 2: primes(2) = 3: primes(3) = 5: primes(4) = 7

    Dim powers(9) As Double
    powers(0) = 1
    Dim i As Short
    For i = LBounds6(powers) + 1 To UBound6(powers)
        powers(i) = powers(i - 1) * 2
    Next
End Sub
```

In most cases, the first argument of the ShiftIndexes pragma is False, which prevents this pragma from affecting array elements whose index is a variable or an expressions, as it usually happens inside a loop. However, this isn't

a fixed rule and the value for the *applytovariables* argument actually depends on how the loop is defined. For example, consider a variations of the above code:

多くの場合、ShiftIndexes プラグマの最初の引数は False であり、ループの中で通常起こってしまう、インデックスが変数か式である配列の要素に影響を与えることから、このプラグマを防ぎます。しかしながら、これは固定規則ではありません、*applytovariables* 引数の値は、実際にループがどう定義されるかに依存します。例えば、上のコードのバリエーションを考えてください。

```
'## powers.ShiftIndexes True, 1
Dim powers(1 To 10) As Double
powers(1) = 1
Dim i As Integer
For i = 2 To 10
    powers(i) = powers(i - 1) * 2
Next
```

In this case the lower and upper bounds of the loop are constants and aren't expressed in terms of LBound and UBound functions, therefore you need to pass True in the first argument of the ShiftIndexes pragma to adjust the indexes inside the loop. This is the generated VB.NET code:

この場合、ループの上限値と下限値は、定数であり、LBound 関数および UBound 関数で表現されていないため、ループ内でインデックスを調整する ShiftIndexes プラグマの最初の引数に True を渡す必要があります。以下は生成された VB.NET のコードです。

```
Dim powers(9) As Double
powers(0) = 1
Dim i As Short
For i = 2 To 10
    powers(i - 1) = powers(i - 1 - 1) * 2
Next
```

When dealing with a multi-dimensional array you need to pass more than two elements to the ShiftIndexes pragma, as in the following code:

多次元配列に対処する場合、以下のコードのように、ShiftIndexes プラグマに 2 つ以上の要素を渡す必要があります。

```
'## mat.ArrayBounds Shift
'## mat.ShiftIndexes False, 1, 2
Dim mat(1 To 10, 2 To 20) As Double
mat(1, 2) = 1: mat(2, 3) = 2
```

which is converted to VB.NET as follows:

これは次のように VB.NET に変換されます:

```
Dim mat(9, 18) As Double
mat(0, 0) = 1: mat(1, 1) = 2
```

Finally, notice that the delta values don't have to be constants, as in this example:

最後に、以下の例のように、デルタ値を定数にする必要がないことに注意してください。

```
Sub Test(arr() As Double, firstEl As Integer, lastEl As Integer)
    '## arr.ArrayBounds Shift
    '## arr.ShiftIndexes True, firstEl
    ReDim arr(firstEl To lastEl) As Double
    arr(firstEl) = 1
End Sub
```

which is converted as follows:

これは次のように変換されます。

```
Sub Test(ByRef arr() As Double, ByRef firstEl As Short, ByRef lastEl As Short)
    ReDim arr(lastEl - (firstEl)) As Double
    arr(firstEl - firstEl) = 1
End Sub
```

Notice that this pragma, like all pragmas that can reference arrays, should be inserted immediately before the Dim statement that declares the array, rather than before the ReDim statement that actually creates the array. Pragmas inserted inside the method where the ReDim statement appears are ignored.

このプラグマは、配列を参照できる他のすべてのプラグマのように、実際に配列をつくる ReDim ステートメントの直前ではなく、配列を宣言する Dim ステートメントの直前に挿入されなければならないことに注意してください。プラグマは、挿入されたメソッド内部で ReDim ステートメントが現れた場合、無視されます。

SetStringSize *nnn*

Specifies that a fixed-length string must be converted as a VB6FixedString_*NNN* type instead of the VB6FixedString type. For example, the following VB6 code:

このプラグマは、固定長文字列が VB6FixedString 型の代わりに VB6FixedString_*NNN* 型として変換させるように指示します。以下の VB6 コードを御覧ください。

```
'## name.SetStringSize 128
Dim id As String * 20, name * As String * 128
```

is converted to VB.NET as follows:

以下のように VB.NET に変換します。

```
Dim id As New VB6FixedString(20)
Dim id As New VB6FixedString_128
```

where the VB6FixedString_128 class is defined in the VisualBasic6.Support.vb file under the My Project folder.

VB6FixedString_128 クラスは My Project フォルダ配下の VisualBasic6.Support.vb ファイルに定義されます。

SetType *nettype*

Specifies that the specified variable, field, property or method must be rendered with a different .NET type. It is useful to specify a type other than Object for Variant and Control variables. For example, the following VB6 code:

指定された変数、フィールド、プロパティ、メソッドが異なる.NET タイプにレンダリングされるように指示します。バリアントとコントロール変数のためのオブジェクト以外のタイプを指定するのに便利です。

```
'## m_Name.SetType String
'## Name.SetType String
Private m_Name As Variant
Property Get Name() As Variant
    Name = m_Name
End Property
```

is translated to VB.NET as follows:

以下のように VB.NET に変換します。

```
Private m_Name As String
Public ReadOnly Property Name() As String
    Get
        Return m_Name
    End Get
End Property
```

Notice that this pragma must have a nonempty scope.

このプラグマは必ずスコープを指定する必要があることに注意してください。

ThreadSafe *Boolean*

Specifies whether variables defined in modules must be decorated with the ThreadStatic attribute. This attribute is only useful when migrating VB6 DLLs that are meant to be used by free-threaded .NET clients, for example ASP.NET pages or COM+ server components. It serves to reduce the gap between the Single-Thread Apartment (STA) model used by VB6 and the free-thread model used by .NET:

このプラグマは、モジュールで定義された変数が ThreadStatic 属性を付与されるべきかどうかを指定します。この属性は、例えば、ASP.NETPages や COM＋サーバーコンポーネントなど、フリースレッド化された.NET クライアントによって利用される VB6DLL を移行する際にのみとても便利です。VB6 で利用されるシングルスレッドアパートメント(STA)モデルと.NET で利用されるフリースレッドモデルの間のギャップを少なくする効果があります。

```
'## ThreadSafe
Public UserName As String
```

If previous code is inside a BAS module, then VB Migration Partner generates what follows:

もし前述のコードが BAS モジュール内であれば、VB Migration Partner は次のように生成します。

```
<ThreadStatic()> _
Public UserName As String
```

The ThreadStatic attribute forces the VB.NET compiler to allocate the UserName variable in the Thread-Local Storage (TLS) area, so that each thread sees and works with a different instance of the variable. This approach makes free-threading more akin to the STA model and avoid many subtle runtime errors due to concurrency and race conditions between threads.

属性はスレッドローカルストレージ(TLS)エリアのユーザーネーム変数を割り当てるために VB.NET コンパイラに強制させます。よってそれぞれのスレッドは変数の異なるインスタンスをもった動作を行います。このアプローチは STA モデルへのより同種のフリースレッド化を作成します。そしてスレッド間の並列処理と競合条件に起因するたくさんの微妙なランタイムエラーを回避します。

UseByVal *mode*

Specifies how to convert by-reference parameters that might be rendered using by-value semantics.
The *mode*argument can be *Yes* (the ByVal keyword is used for the parameter unless an explicit ByRef keyword is present), *Force* (the ByVal keyword is used for the parameter, regardless of whether an explicit ByRef keyword is present), *No* (the parameter is translated as is):

このプラグマは、値による意味論を使用することでレンダリングされるかもしれない参照されるパラメータをどのように変換するかを指定します。*mode* 引数は *Yes*(ByVal キーワードが明確な ByRef キーワードで存在している場合を除いたパラメータで使われている場合)、*Force*(ByVal キーワードが明確な ByRef キーワードがあるかどうかに関わらず、パラメータで使われている場合)、*No*(パラメータがそのまま変換される場合)が指定できます。

```
'## project:UseByVal Yes
```

If the parameter is omitted, the *Yes* value is assumed.

パラメータがない場合は *Yes* としてみなされます。

UseSystemString *boolean*

Specifies whether fixed-length strings inside Type…End Type blocks must be converted to VB.NET. If the argument is True or omitted, fixed-length strings inside the pragma's scope are converted as properties that take or return System.String values.

このプラグマは、Type…End Type ブロック内部の固定長文字列を VB.NET に変換されるように指示します。引数が True または省略された場合、プラグマのスコープ内の固定長文字列は SystemString 値を取得または戻すプロパティとして変換されます。

```
'## testudt.UseSystemString
```

5.4 Pragmas that affect how code is converted／コード変換の仕方に影響を与えるプラグマ

ConvertGosubs *boolean, optimize*

Specifies whether VB Migration Partner should attempt to convert old-styled Gosubs into calls to separate methods. The first boolean argument is True (or omitted) to enable this conversion; the second Boolean argument is True if you want to optimize the generated code so that parameters of the separate method are declared with ByVal if possible. This pragma can have project-, file-, or method-level scope. For example, consider the following VB6 code:

このプラグマは、VB Migration Partner が古い形式の Gosubs をメソッドを分割する呼び出しに変換するかどうかを指定します。最初の Boolean 引数が True（または無し）を設定するとこの変換が有効になります。生成されるコードを最適化したい場合、第 2 の Boolean 引数を True にすると、可能であれば、分割メソッドのパラメータが ByVal で宣言されます。このプラグマはスコープをプロジェクトレベル、ファイルレベル、またはメソッドレベルにすることが出来ます。以下の VB6 コードを御覧ください。

```
Function GetValue(x As Integer) As Integer
    '## ConvertGosubs True
    On Error Resume Next

    Dim s As String, name As String
    s = "ABC"
    name = "Code Architects"
    GoSub BuildResult
    Exit Function
BuildResult:
    GetValue = x + Len(s) + Len(name)
    Return
End Function
```

VB Migration Partner detects that the code that begins at the BuildResult label (a.k.a. the target label) can be safely refactored to a separate method, that receives four arguments. The result of the conversion is therefore:

VB Migration Partner は、4 つの引数を受け取ることができ、分割メソッドへ安全にリファクタリング可能な BuildResult ラベル（別名ターゲットラベル）で始まるコードを見つけ出します。変換結果を御覧ください。

```
    Public Function GetValue(ByRef x As Short) As Short
        Dim s As String
        Dim name As String
        On Error Resume Next

        s = "ABC"
        name = "Code Architects"
        Gosub_GetValue_BuildResult(x, GetValue, s, name)
        Exit Function
    End Function

    Private Sub Gosub_GetValue_BuildResult(ByRef x As Short, ByRef GetValue As
Short, _
        ByRef s As String, ByRef name As String)
        On Error Resume Next
        GetValue = x + Len6(s) + Len6(name)
    End Sub
```

Notice that the external method contains an On Error Resume Next statement, because the original GetValue method also contains this statement.

外部メソッドは On Error Resume Next ステートメントを含むことに注意してください。その理由は元の GetValue メソッドもまたこのステートメントを含むためです。

All arguments to the new *Gosub_GetValue_BuildResult* method are passed by reference, so that the caller can receive any value that has been modified inside the method (as is the case with the *GetValue* parameter in previous example.) You can have VB Migration Partner optimize this passing mechanism and use ByVal if possible by passing True as the second argument to the ConvertGosubs pragma:

新しい *Gosub_GetValue_BuildResult* メソッドに対する全ての引数は参照によって渡されます。そのため、呼び出し側は（前述のサンプルの *GetValue* パラメータを伴うケースがそうであるように）メソッドの内部で修正されたどんな値でもを受け取ることができます。ConvertGosubs プラグマの 2 番目の引数として True を引き渡すことによって、VB Migration Partner に受け渡しメカニズムを最適化させ、可能ならば ByVal を使用させることができます。

```
    '## ConvertGosubs True, True
```

If this optimization is used in the above code, the separate method is rendered as follows:

この最適化が上述のコードに使用されるなら、分割メソッドは以下のように作成されます。

```
    Private Sub Gosub_GetValue_BuildResult(ByVal x As Short, ByRef GetValue As
Short, _
        ByVal s As String, ByVal name As String)
        On Error Resume Next
        GetValue = x + Len6(s) + Len6(name)
    End Sub
```

If you don't like the name that VB Migration Partner assigns to the automatically-generated method, you can easily change it by means of a **PostProcess** pragma:

もし VB Migration Partner が自動的に生成されたメソッド名に割り当てた名称が気に入らない場合、PostProcess プラグマを使って簡単に変更することができます。

```
    '## PostProcess "Gosub_GetValue_BuildResult", "AssignValueResult"
```

It is important to keep in mind that the conversion of a Gosub block of code into a separate method isn't always possible. More precisely, VB Migration Partner can perform this conversions only if all the following conditions are met:

ソースコードの Gosub ブロックを分割メソッドにする変換はいつも可能というわけではないことを覚えておくことは重要です。より正確には、以下のすべての条件が満たされる場合にだけ、VB Migration Partner はこの変換を実行できます。

a. The method doesn't contain any Resume statement. (On Error Resume Next is OK, though).
   メソッドが Resume ステートメントを含んでいないこと（ただし、On Error Resume Next は OK)。
b. The target label isn't located inside a If, Select, For, Do, or Loop block.
   ターゲットラベルが If、Select、For、Do また Loop ブロック内部にないこと。
c. The target label isn't referenced by a Goto, On Goto/Gosub, or On Error statement.
   ターゲットラベルが Goto、On Goto/Gosub または On Error ステートメントによって参照されていないこと。
d. The target label must be preceded by a Goto, Return, End, Or Exit Sub/Function/Property statement.
   ターゲットラベルが Goto、Return、End、または Exit Sub/Function/Property ステートメントより前になければならない。
e. The code block must terminate with an inconditional Return.
   コードブロックが無条件リターンで終了されていなければならない。
   (Blocks that terminate with End Sub/Function/Property or Exit Sub/Function/Property statements aren't converted.)
   （End Sub/Function/Property または Exit Sub/Function/Property ステートメントで終了するブロックは変換されません。）
f. The block of code between the target label and the closing Return/End statement doesn't contain another label.

ターゲットラベルと終了のReturn/Endステートメント間のコードブロックが他のラベルを含んでいないこと。

If a method contains one or more labels that can't be converted to a separate method, VB Migration Partner still manages to convert the remaining labels. In general, a label can be converted to a separate method if it neither references nor is referenced by a label that can't be converted (for example, a label that is defined inside an If block or a label that doesn't end with an unconditional Return statement.)

もしメソッドが分割メソッドに変換できないラベルをひとつ以上含んでいる場合、VB Migration Partner は残りのラベルを変換しようとします。一般的に、参照しておらず、また変換できないラベルからの参照もなければ、そのラベルを分割メソッドに変換できます。(例えば、If ブロックの内部で定義されたラベルや無条件 Return ステートメントで終了しないラベルがそうです。)

Note:
in a method that contains one or more Gosub statements that are converted to separate methods, the variable initialization merging feature is disabled.

分割メソッドに変換される Gosub ステートメントをひとつ以上含むメソッドでは、変数初期化マージ機能は無効となります。

DefaultMemberSupport *boolean*

Specifies whether VB Migration Partner must wrap references to default members inside calls to the special GetDefaultMember6 and SetDefaultMember6 methods defined in the language support library:

このプラグマは、VB Migration Partner が、言語サポートライブラリで定義された特別な GetDefaultMember6 メソッドと SetDefaultMember6 メソッドの呼び出しの中のデフォルトメンバに対する参照をラッピングしなければならないかどうかを指定します。

```
'## Rem enable default member support for this class, except for the Test member
'## DefaultMemberSupport
'## Test.DefaultMemberSupport False
```

FixRemarks *boolean*

Specifies how VB Migration Partner must convert VB6 remarks that start with three consecutive apostrophes. By default they are prefixed by an additional apostrophe+space, so that the VB.NET compiler doesn't mistakenly interpret them as XML remarks. You can use False for the argument to disable such automatic fix, which can be useful if you decide to insert VB.NET XML remarks right in the VB6 code:

このプラグマは、VB Migration Partner が 3 つの連続したアポストロフィで始まる VB6 コメントをどのように変換しなければならないかを指定します。デフォルトではそれらの先頭にはアポストロフィと空白があります。そのため、VB.NET コンパイラは XML コメントと間違えることはありません。そのような自動的な修正を無効化にするために引数に False を指定することができます。それは、VB6 コードに VB.NET XML コメントを適切に挿入しようと思った場合に役立ちます。

```
'## project:FixRemarks False
```

LogicalOps *boolean*

Specifies whether And and Or VB6 keywords must be translated to AndAlso and OrElse VB.NET keywords. For example, the following code:

このプラグマは、And と Or という VB6 キーワードを AndAlso と OrElse という VB.NET キーワードに変換しなければならないかどうかを指定します。下記のコードを御参照ください。

```
Sub Test(ByVal x As Integer)
    '## LogicalOps True
    If x < 0 And x > 100 Then Err.Raise 9
    ' ...
End Sub
```

The pragma can have project, class, and method scope, and can issued multiple times inside the same method, to selectively enable and disable this feature. For example, this VB6 code:

このプラグマはプロジェクト、クラス、メソッドをスコープとすることができ、この機能を有効にしたり無効にしたりするために、同じメソッド内に複数回発行することができます。以下の VB6 コードを御参照ください。

```
'## LogicalOps True
If x > 0 Or (x = 0 And y < 0) Then …
'## LogicalOps False
If x1 > 0 Or (x1 = 0 And y1 < 0) Then …
```

translates to:

これを変換すると以下のようになります。

```
If x > 0 OrElse (x = 0 AndAlso y < 0) Then …
If x1 > 0 Or (x1 = 0 And y1 < 0) Then …
```

MergeIfs *boolean, includeParens*

Specifies whether two or more nested Ifs are automatically merged into a single statement by means of an AndAlso operator. This pragma can have project, class, and method scope. The first argument can be True (default if omitted) or False, where the latter value can be used to override another pragma with broader scope.
The *includeParens* optional argument is True (default if omitted) if you want to enclose individual expressions within parenthesis, or False to drop these parenthesis.

このプラグマは、二つ以上の入れ子になった If 文が AndAlso 演算子で自動的にひとつのステートメントにマージされるかどうかを指定します。このプラグマはプロジェクトレベル、クラスレベル、メソッドレベルのスコープにすることができます。最初の引数は True（省略した場合の初期値）、または False にすることができ、False はより広いスコープのほかのプラグマを無効にするために利用できます。任意の *IncludeParens* 引数は、括弧で個々の語句を囲みたい場合は True（省略した場合の初期値）、これらの括弧を除去するには False になります。

For example, consider the following code

例として、次のコードを御覧ください。

```
    Sub Test(ByVal obj As Object, ByVal obj2 As Object)
        '## MergeIfs True
        If Not obj Is Nothing Then
            If obj.Text = "" Then Debug.Print "Empty Text"
        End If

        If Not obj Is Nothing Then
            If Not obj2 Is Nothing Then
                If obj.Text = obj2.Text Then Debug.Print "Same text"
            End If
        End If
    End Sub
```

This is how VB Migration Partner translates it to VB.NET

これが VB Migration Partner が VB.NET に変換する方法です。

```
    Sub Test(ByVal obj As Object, ByVal obj2 As Object)
        ' UPGRADE_INFO (#0581): Two nested If...End If blocks have been merged.
        If (Not obj Is Nothing) AndAlso (obj.Text = "") Then
            Debug.WriteLine("Empty Text")
        End If

        ' UPGRADE_INFO (#0581): Three nested If...End If blocks have been merged.
        If (Not obj Is Nothing) AndAlso (Not obj2 Is Nothing) AndAlso (obj.Text =
obj2.Text) Then
            Debug.WriteLine("Same text")
        End If
    End Sub
```

You can further simplify the expression by dropping the parenthesis around each subexpressions, by using the following pragma:

さらに、次のプラグマを使用することにより、各語句の括弧を削除して式を簡素化することができます。

```
    '## MergeIfs True, False
```

Notice that this optimization is disabled if there are one or more special pragmas between the two IF statements, including InsertStatement, ReplaceStatement, PreInclude and PostInclude.

この最適化は、二つの If ステートメントの間に特別なプラグマ、InsertStatement、ReplaceStatement,、PreInclude や PostInclude が含まれている場合に無効になりますのでご注意ください。

NullSupport *boolean*

Specifies whether VB Migration Partner must generate code that provides better support for null propagation in expressions. For example, consider the following VB6 code:

このプラグマは、VB Migration Partner が語句の Null 伝播をより高度にサポートするコードを生成するかどうかを指定します。例えば、次の VB6 コードを御覧ください。

```
'## NullSupport
Sub Test(name As Variant, city As Variant)
    name = UCase(name) + Trim(city)
End Sub
```

and its VB.NET translation

変換された VB.NET コードは以下になります。

```
Sub Test(ByRef name As Object, ByRef city As Object)
    name = UCase6(name) + Trim6(city)
End Sub
```

Methods affected by this pragma are: Chr, CurDir, Environ, Error, Hex, LCase, LTrim, Mid, Oct, Right, RTrim, Space, and UCase. The corresponding method in the language support library has same name plus a "6" suffix.

このプラグマによる影響を受けるメソッドは、Chr、CurDir、Environ、Error、Hex、LCase、LTrim、Mid、Oct、Right、RTrim、Space、UCase です。言語サポートライブラリの対応するメソッドは等しく"6"の添字がついた名前を持っています。

PreInclude *includefile*

Includes a text file before the parsing process begins. The argument is the path to the text file that contains the text to be included. (It can be an absolute path or a path relative to the folder that contains the .vbp project file.). The argument never needs to be included in double quotes:

このプラグマは、構文解析プロセスが始まる前に、テキストファイルを含めます。引数は含めるべきテキストを含んでいるテキストファイルに対するパスです。(絶対パスまたは vbp プロジェクトファイルが含まれているフォルダに対する相対パスを使用することができます。)引数はダブルコーテーションで囲まれた記述である必要はありません。

```
'## PreInclude c:¥includes¥copyright.txt
```

If the *includefile* argument points to a non-existing file, no text is included and no migration warning is generated. The PreInclude pragma supports recursion, in the sense that the file being included can contain additional

PreInclude pragmas, up to any nesting level. VB Migration Partner doesn't allow circular references among include files.

Includefile 引数が存在しないファイルを指す場合、テキストには何も含まれず、変換時の Warining も生成されません。PreInclude プラグマは、そのファイルにさらに PreInclude プラグマを含むことができるという意味で、入れ子内部まで再帰的に使用することができます。VB Migration Partner は含まれるファイル間での循環参照を許容しません。

Keep in mind that you typically can't use this pragma to include a standard CLS or BAS file, because such VB6 files contains hidden text and attributes that would be included in the wrong position and might deliver unexpected results or even crash VB Migration Partner. You use this pragma at your own risk.

通常、標準的な CLS や BAS ファイルを含めるのにこのプラグマを使えないことに注意してください。なぜなら、そのような VB6 ファイルは間違った場所に含まれている非表示のテキストや属性を含んでおり、予期しない結果になったり、VB Migration Partner をクラッシュさせるかもしれないからです。このプラグマを利用する場合、自己の責任において使用してください。

A good use for this pragma is as a simple way for multiple project to share the same set of project-level pragmas. For example, let's assume that you have to migrate ten VB6 projects that require the same set of PostProcess pragmas. Instead of copying the pragma text in each and every project, you can gather them in a text file and reference the text from inside any source file in each project:

このプラグマの上手な利用法は、プロジェクトレベルのプラグマの同じ組み合わせを共有する複数のプロジェクトに対して簡単な方法をとして利用することです。例えば、PostProcess プラグマの同じ組み合わせを必要とする 10 個の VB6 プロジェクトを変換しなければならないと仮定しましょう。各プロジェクトすべてにプラグマのテキストをコピーする代わりに、ひとつのテキストファイルにそれらを集めることができ、それぞれのプロジェクトのソースファイル内部からテキストを参照することができます。

```
'## PreInclude c:¥code¥pragmas.txt
```

Notice, however, that in this specific case, the pragma.txt file can't contain the special pragmas that must be parsed before the project itself and that are necessarily to be stored in the VBMigrationPartner.pragmas file. However, quite conveniently PreInclude pragmas are honored even inside *.pragmas files, therefore each folder might contain a VBMigrationPartner.pragmas file that contains only one line containing the PreInclude pragma pointing to the text file that contains the actual pragmas that are shared among multiple projects.

しかし、この特別なケースにおいて、pragma.txt ファイルは、VBMigrationPartner.pragmas ファイルに格納されている必要がありかつプロジェクト自身の前に解析されなければならない特別なプラグマを含めることはできません。しかし、非常に便利な PreInclude プラグマは*.pragmas ファイルの内部でさえ優先されます。そのため、各フォルダは、複数のプロジェクト間で共有される実際のプラグマを含むテキストファイルを指し示す PreInclude プラグマを含むたった 1 行を含む VBMigrationPartner.pragmas ファイルを含めることができます。

PreProcess *searchpattern, replacepattern, ignorecase, applytohiddencode, regionpattern*

Applies a replace regular expression to the VB6 code before it is parsed. The first argument is the search regex pattern string. The second argument is the replace regex pattern and can reference match groups defined in the

search pattern; it abides to the same syntax rules as the Regex.Replace method but it also recognizes character sequences such as ¥n (newline), ¥r (carriage return) and ¥t (tab). Backslash characters in the replace regex must be doubled (as in ¥¥). The third argument is a Boolean value that specifies whether searches are case insensitive (if omitted, it defaults to True). The fourth argument is True if this pragma applies also to the .vbp file and to the hidden portion of an .frm or .ctl file, False (or omitted) if it applies only to the VB6 code that you see inside the code editor. The fifth argument is an optional regex pattern that can be used to precisely indicate the code regions to which the search pattern must be applied.

このプラグマは、解析される前に VB6 のソースコードに対する正規表現の置換を適用します。最初の引数は、検索用の正規表現パターン文字列です。2 番目の引数は置換用の正規表現パターンで検索パターンに定義された一致グループを参照することができます。つまり、それは Regex.Replace メソッドと同じ構文規則に従いますが、¥n（改行）、¥r（キャリッジリターン）、¥t（タブ）のような文字列も認識します。置換用の正規表現における¥はふたつ（¥¥）でなければなりません。3 番目の引数は、検索が大文字と小文字を区別しないかどうかを指定するブール値（省略時は True）です。4 番目の引数は、このプラグマが.vbp ファイルと、.frm や.ctl ファイルの非表示領域に対して適用される場合は True であり、コードエディタ内で見える VB6 コードだけに適用する場合は False(または、省略)です。5 番目の引数は、検索パターンが適用されなければならないコード領域を正確に示すのに使われる任意の正規表現パターンです。

For example, the following pragma pre-processes the VB6 code and converts all OLE_COLOR variables into plain Long variables:

例えば、以下のプラグマは VB6 コードを事前処理して、すべての OLE_COLOR 変数を簡単な Long 変数に変換します。

```
'## PreProcess "As OLE_COLOR¥b", "As Long"
```

The following pragma renames a button named “AddHandler” into “btnAddHandler”, to avoid the problem caused by AddHandler being a reserved VB.NET keyword. In this case it is necessary to apply the pragma to the hidden portion of the file and to account for event names, where the control’s name is followed by an underscore

以下のプラグマは、VB.NET の予約語である AddHandler によって起こる問題を回避するために、“AddHandler”という名称のボタンを“btnAddHandler”という名称に変更します。このケースではアンダースコアが後に続くコントロール名の場所の、イベント名の記述とファイルの非表示領域にプラグマを適用する必要があります。

```
'## PreProcess "(¥bAddHandler¥b|(? < =¥bSub¥s+)AddHandler(?=_))",
"btnAddHandler", True, True
```

The following example uses the optional *regionpattern* argument to restrict the action of the PreProcess pragma to Sub methods only:

以下の例では Sub メソッドにのみ PreProcess プラグマの動作を制限する任意の *regionpattern* 引数を使用します。

```
'## PreProcess "Dim ctrl As Control", "Dim ctrl As Object", False, _
        False, "¥bSub¥b.+?¥bEnd Sub¥b"
```

Notice that the last argument uses the .+? pattern to ensure that the search region doesn't extend to more than one method at a time.

最後の引数は、一度に一つ以上のメソッドに検索範囲が拡張しないことを保証するために、「.+？」パターンを使うことを覚えてください。

You can apply this pragma at the project-level only by storing it in a *.pragmas file – for example, in the Widgets.vbp.pragmas file for all the files in the Widgets.vbp project. An important note: this pragma can't have a**project:** prefix: if the pragma appears in a source file it is implicitly scoped at the file level; if it appears in a *.pragmas file, it is implicitly scoped at the project level.

*pragmas ファイルに格納することによって、プロジェクトレベルでこのプラグマを適用できます。(例：Widgets.vbp プロジェクト内の全てのファイルに適用する、Widgets.vbp.pragmas ファイル) 重要なこととして、このプラグマは **Project:**を接頭辞にすることはできません。プラグマが Source ファイル内に表示されている場合、それは暗黙的にファイルレベルでのスコープであり、*.pragmas ファイルに表示されている場合、それは暗黙的にプロジェクトレベルでのスコープです。

PostInclude *includefile*

Includes a text file after the conversion process has been completed. The argument is the path to the text file that contains the text to be included. (It can be an absolute path or a path relative to the folder that contains the .vbp project file.). The argument never needs to be included in double quotes:

このプラグマは、変換プロセスが完了した後に、テキストファイルを含んでいます。引数は含まれるべきテキストを含むテキストファイルへのパスです。(それは絶対パスまたは vbp プロジェクトファイルが含まれているフォルダへの相対パスを設定することができます。) 引数は、二重引用符にて記述する必要はありません。

```
'## PreInclude c:¥includes¥copyright.txt
```

If the *includefile* argument points to a non-existing file, no text is included and no migration warning is generated. The PostInclude pragma supports recursion, in the sense that the file being included can contain additional PostInclude pragmas, up to any nesting level. VB Migration Partner doesn't allow circular references among include files.

includefile 引数が非存在ファイルを指定する場合、どのようなテキストも含まれません、そして、どのような移行警告も生成されません。PostInclude プラグマは再帰をサポートします、その意味は、含まれているファイルが、ネストレベルの PostInclude プラグマも含めることができるという意味です。VB Migration Partner は含まれるファイルの中の循環参照を許容しません。

A good use for this pragma is to add one or more methods or classes to the resulting VB.NET code.

このプラグマの良い利用方法としては、1 つ以上のメソッドやクラスを生成される VB.NET コードに追加する場合に有効です。

PostProcess *searchpattern, replacepattern, ignorecase, applytohiddenfiles, regionpattern*

Applies a replace regular expression to the VB.NET code generated by the migration process. The first argument is the search regex pattern string. The second argument is the replace regex pattern and can reference match groups defined in the search pattern; it abides to the same syntax rules as the Regex.Replace method but it also recognizes character sequences such as ¥n (newline), ¥r (carriage return) and ¥t (tab). Backslash characters in the replace regex must be doubled (as in ¥¥). The third argument is a Boolean value that specifies whether searches are case insensitive (if omitted, it defaults to True). The fourth argument specifies whether the PostProcess pragma applies only to “hidden” files such as AssemblyInfo.vb and *.vbproj files (if omitted, it defaults to False and the pragma applies only to standard *.vb files, including *.Designer.vb files). The fifth argument is an optional regex pattern that can be used to precisely indicate the code regions to which the search pattern must be applied.

このプラグマは、移行プロセスによって生成された VB.NET コードに正規表現を置き換えることを適用します。最初の引数は検索する正規表現パターン文字列です。2 番目の引数は置換する正規表現パターンで、検索パターンに定義された該当するグループを参照することが出来ます。それは、Regex.Replace メソッドと同じ構文規則に準拠します。しかしまた Replace メソッドは¥n（改行）、¥r（キャリッジリターン）、¥t（タブ）としてキャラクターシークウェンスを認識します。置換正規表現中のバックスラッシュキャラクターは 2 重記述でなければなりません（¥¥）。3 番目の引数は、検索が大文字と小文字を区別するかどうか指定するブール値です（省略の場合、デフォルトは True）。4 番目の引数は、PostProcess プラグマが AssemblyInfo.vb や*.vbproj ファイルなどの「隠された」ファイルだけに適用するかどうか指定します。（省略の場合、デフォルトは False。プラグマは*.Designer.vb ファイルを含む標準の*.vb ファイルだけに適用されます）5 番目の引数は、正確に、検索パターンを適用しなければならないコード領域を示すのに使用できる任意の正規表現パターンです。

For example, the following pragma changes the base form for the converted application:

以下のプラグマは変換されたアプリケーションについて標準の Form に変更する例です。

```
'## project:PostProcess "Inherits CodeArchitects.VB6Library.VB6Form", _
    "Inherits AppFramework.FormBase", False, True
```

The search pattern can include match groups and you can reference these search groups in the replacement pattern. For example, the following pragma moves the declaration of the controlling variable into a For Each loop, if the variable is defined immediately before the loop:

検索パターンはマッチするグループを含むことができます。そして、置換パターンにこれらの検索グループを参照することができます。例えば、以下のプラグマは変数がループの直前に定義されている場合、コントロール変数の宣言を For Each ループ内部に移動します。

```
'## PostProcess "Dim (?< name > ¥w+) As (?< type> ¥w+)( = .+)?¥s+For Each ¥k< name > In", _
    "For Each ${name} As ${type} In"
```

The following example is similar to previous one, except it uses the optional *regionpattern* argument to restrict the action of the PostProcess pragma only to Function methods named GetArray and GetString:

GetArray と GetString メソッドという名前の関数のメソッドに対してのみ PostProcess プラグマの動作を制限するオプション *regionpattern* 引数を使用する点を除き、以下の例は前の例と同様です。

```
    '## PostProcess  "Dim (?< name> ¥w+) As (?< type > ¥w+)( = .+)?¥s+For Each ¥k
< name > In", _
                     "For Each ${name} As ${type} In", False, False, _
                     "¥bFunction (GetArray|GetString)¥b.+?¥bEnd Function¥b"
```

Notice that the last argument uses the .+? pattern to ensure that the search region doesn't extend to more than one function at a time.

最後の引数で使用されている。+？パターンは、検索領域を一度に複数の関数に拡張しないことに注意してください。

PreservePropertyAssignmentKind *mode*

Specifies whether VB Migration Partner must take additional cure when converting custom properties. If *Yes* (or omitted) the pragma applies only to properties whose definition contains both a Property Let and a Property Set; if *Force* it applies to all properties that have a Property Let block, even if the Property Set block is missing. In all cases, the pragma applies only to properties whose Let block specifies a non-scalar data type.

このプラグマは、カスタムプロパティを変換するとき、VB Migration Partner が追加対応を取らなければならないかどうか指定します。もし引数を Yes（また省略）の場合、プラグマは Property Let と Property Set の両方を含む定義のプロパティにのみ適用します。Force の場合、Property Set ブロックがない場合であっても Property Let ブロックを持っているすべてのプロパティに適用されます。全てのケースにおいて、プラグマは Let ブロックが非スカラデータ型を指定するプロパティのみに適用します。

```
'## DataResult.PreservePropertyAssignmentKind Yes
```

To understand when this pragma can be necessary, let's consider the following VB6 code:

このプラグマがいつ必要になるかを理解するために、次の VB6 を御覧下さい。

```
Private m_MyData As Variant

Property Get MyData() As Variant
    If Not IsObject(m_MyData) Then
        MyData = m_MyData
    Else
        Set MyData = m_MyData
    End If
End Property

Property Let MyData(ByVal newValue As Variant)
    m_MyData = newValue
```

```
            ProcessDataValue newValue    ' (ProcessDataValue is defined elsewhere)
        End Property

        Property Set MyData(ByVal newValue As Variant)
            Set m_MyData = newValue
        End Property

        Sub Main()
            Dim rs As New  ADODB.Recordset, par As New ADODB.Parameter
            ' ... init the recordset here
            MyData = par
            Set MyData = par
            ' ...
            MyData = rs
            Set MyData = rs
        End Sub
```

This is how VB Migration Partner usually converts the above code:

VB Migration Partner が通常どのように上記のコードを変換するかを示します。

```
        Private m_MyData As Object

        Property MyData() As Object
            Get
                ' *** VB Migration Partner has simplified this code
                Return m_MyData
            End Get
            Set(ByVal newValue As Object)
                If Not IsObject6(newValue) Then
                    m_MyData = newValue
                    ProcessDataValue(newValue)      ' (ProcessDataValue is
defined elsewhere)
                Else
                    m_MyData = newValue
                End If
            End Set
        End Property

        Sub Main()
            Dim rs As New ADODB.Recordset
            ' ... init the recordset here
            MyData = par.Value   ' ** resolution of default property
```

```
        MyData = par
        ' ...
        MyData = rs.Fields   ' ** resolution of default property
        MyData = rs
    End Sub
```

Notice that VB Migration Partner expands *par* into par.Value and *rs* into *rs.Fields* when the assignment to the*MyData* property lacks the Set keyword. While the two cases seem similar, they aren't. The key different is that the ADODB.Parameter's default property (Value) is a scalar value (a string, a number, a date, etc.) whereas the ADODB.Recordset's default property (Fields) is an object.

MyData プロパティへの割り当てが Set キーワードを欠いている際、VB Migration Partner が *rs.Fields* 内の *rs* と par.Value 内の *par* を拡張することに注意してください。二つのケースは似ておりますが、異なります。異なるキーポイントは、ADODB.Parameter のデフォルトのプロパティ(値)がスカラー値(文字列、数値、日付など)であるということですが、ADODB.Recordset のデフォルトプロパティ(フィールド)はオブジェクトです。

When *par.Value* is assigned to the MyData property, the following statement

以下のステートメントのように *par.Value* が MyData プロパティに割り当てられます。

```
If Not IsObject6(newValue) Then
```

recognizes that it's a scalar and executes the code block that was originally in the Property Let procedure (and therefore the *ProcessDataValue* method is correctly invoked). However, when *rs.Fields* is assigned to the*MyData* property, the test in the If statement finds that it's not a scalar and mistakenly executes the code block that was originally in the Property Set procedure. As a result, the *ProcessDataValue* method isn't invoked as it should have been.

それがスカラーであると認識し、元々の Property Let プロシージャにあったコードブロックを実行します。(よって、*ProcessDataValue* メソッドは正しく呼び出されます。)しかし、*rs.Fields* が *MyData* のプロパティに割り当てられる際、If ステートメントにおけるテストは、それがスカラーでないことを認識し、元々の Property Set プロシージャにあったコードブロックを誤って実行します。結果として、それがそうあるべきであったとき、*ProcessDataValue* メソッドは呼び出されません。

(You can also omit the argument, because Yes is the default). You can solve this problem by applying the PreservePropertyAssignmentKind pragma to the MyData property, as follows:

(デフォルトは Yes なので引数を省略することができます。)MyData プロパティに PreservePropertyAssignmentKind プラグマを適用させることによって、この問題を解決することが出来ます。以下を御覧下さい。

```
'## MyData.PreservePropertyAssignmentKind Yes
```

In this case VB Migration Partner generates the following VB.NET code:

このケースにおいて VB Migration Partner は以下の VB.NET のコードを生成します。

```
        Private m_MyData As Object
        Property MyData(ByVal Optional _assignmentKind As PropertyAssignmentKind
_
                        = PropertyAssignmentKind.Let) As Object
            Get
                ' *** VB Migration Partner has simplified this code
                Return m_MyData
            End Get
            Set(ByVal newValue As Object)
                If _assignmentMode = PropertyAssignmentKind.Let Then
                    m_MyData = newValue
                    ProcessDataValue(newValue)     ' (ProcessDataValue is defined
elsewhere)
                Else
                    m_MyData = newValue
                End If
            End Set
        End Property

        Sub Main()
            Dim rs As New ADODB.Recordset
            ' ... init the recordset here
            MyData = par         ' ** no resolution of default property
            MyData(PropertyAssignmentKind.Set) = par
            ' ...
            MyData = rs          ' ** no resolution of default property
            MyData(PropertyAssignmentKind.Set) = rs
        End Sub
```

In this new version the task of determining whether the original assignment was performed by means of a Let or a Set is carried out by means of the *_assignmentMode* parameter. Being optional, the parameter is omitted when assigning using a “let” and is used only in “set” assignments, which helps generating less cluttered code.

この新しいバージョンでは、元の割り当てが Let か Set によって実行されたかどうか決定するタスクが *_assignmentMode* パラメタによって行われます。オプションなので、パラメタは、「let」を使用し割り当てる際に省略されます。そして「set」割り当てのみ使用された際にも省略されます。どちらにおいてもよりより整理されたコードを生成するのに役立ちます。

Notice that when this pragma is used, the default property isn’t expanded any longer and the *newValue* parameter receives the actual object being assigned, not its default property. This is exactly what the happens in VB6.

このプラグマが使用されている場合、デフォルトプロパティはこれ以上拡張されません。そして、*newValue* パラメタがデフォルトプロパティではなく、割り当てられる実際のオブジェクトを受けとることに注意してください。これは VB6 でも同様な事象です。

By default, the pragma applies only to properties that have both a Property Let and a Property Set pragma. However, if you use *Force* as an argument, the pragma applies also to properties that have only the Property Let block (provided that the parameter for such block is Variant or a non-scalar type). In this case the PropertyAssignmentKind argument is generated but never used, and the pragma only has the effect of preventing VB Migration Partner from generating.

デフォルトで、プラグマは Property Let と Property Set プラグマの両方を持っているプロパティだけに適用されます。しかし、引数として *Force* を使用する場合、(もしそのようなブロックのパラメタが Variant 型か非スカラ型であるならば)プラグマは Property Let ブロックしか持っていないプロパティに適用されます。この場合、PropertyAssignmentKind 引数は生成されますが、決して使用されません。そして、プラグマには VB Migration Partner の生成を防ぐという効果があるだけです。

SetOption *optionname, boolean*

Specifies a code generation option, depending on the argument passed in the first argument. Supported options are:

このプラグマは、コード生成オプションを指定します。最初の引数に渡される引数に依存します。サポートされるオプションは以下になります。

VariantContainsArray:
VB Migration Partner assumes that VB6Variant variables under the scope of this pragma may contain an array; as a result, when passing a VB6Variant variable to a ByVal VB6Variant parameter, the variable is wrapped in a CVar6 method, to ensure that the inner array is correctly cloned.

VB Migration Partner は、このプラグマのスコープである VB6Variant 変数が配列を含むかもしれないとみなします。結果として、VB6Variant 変数を ByVal VB6Variant パラメタに引き渡す際に、変数が、内部配列が正しくクローン化されるのを保証するために CVar6 メソッドで包装されます。

FixedStringsAsStructures:
VB Migration Partner generates one member named "VB6FixedString_NNN" (where NNN is a number) for each fixed-length string that is used inside an array. By default such members are generated as classes, but if this parameter is True or omitted then the member is rendered as a strictire. (This option has been introduced in version 1.23 to preserve compatibility with older versions, which rendered those members as classes.) This option is always applied to the entire project, regardless of the specified scope.

VB Migration Partner は配列の内部で利用されているそれぞれの固定長文字列に対し"VB6FixedString_NNN"(NNN の部分は数値)で命名されたひとつのメンバを生成します。デフォルトでそのようなメンバはクラスとして生成されます。しかし、このパラメータが True もしくは省略の場合、メンバは構造体としてレンダリングされます。(このオプションは、クラスとしてそれらのメンバーをレンダリングした旧来の Version との互換性を保持するために

Version1.23 で紹介されました。)このオプションは指定されたスコープにかかわらず常に全体のプロジェクトに適用されます。

RemarkMethodBody:
VB Migration Partner remarks out all the statements inside all properties and methods under the scope of this pragma, so that you can correctly resolve property/method references without having to fix individual statements that cause a compilation error. This pragma is especially useful to incrementally migrate large VB6 projects, one file at a time. You can specify this pragma at the project, file, and method level. (If applied at the variable level, this pragma is ignored.)

コンパイルエラー引き起こす個々のステートメントを修正する必要がないプロパティ/メソッド参照を正確に解決できるように、VB Migration Partner はこのプラグマのスコープ下のすべてのプロパティとメソッド内部のすべてのステートメントをコメント記述します。このプラグマは、一度に 1 個のファイルが大きい VB6 プロジェクトを徐々に移行するのに大いに役立ちます。プロジェクト、ファイル、メソッドレベルにこのプラグマを指定することができます。(変数レベルで適用する場合は、このプラグマは無視されます)

For example, the following pragmas force VB Migration Partner to generate fixed-length strings as structures and to remark all statements in all methods in the current project. Notice that you can omit the second argument if it is True:

例として、以下のプラグマは構造体としての固定長文字列を生成する、また現行プロジェクトの全てのメソッドの全てのステートメントをコメントアウトすることを VB Migration Partner に指示します。それが True の場合、2 番目の引数を省略できることに注意してください。

```
'## project:SetOption FixedStringsAsStructures, True
'## project:SetOption RemarkMethodBody
```

ExcludeMethodBody:
VB Migration Partner excludes all statements inside methods and property procedures when parting the VB6 source code, but not method and property signatures (so that it can correctly resolve all references). This pragma is especially useful to incrementally migrate *very large* VB6 projects, one file at a time, to speed up the migration process of large files, and to avoid out-of-memory errors during the migration process. You can specify this pragma at the project, file, and method level. (If applied at the variable level, this pragma is ignored.)

メソッドとプロパティシグネチャではない VB6 ソースコードを分割する際に、メソッドとプロパティプロシージャ内の全てのステートメントを除外します。(結果として全ての参照を正確に解決することができます。)このプラグマは、一度に 1 個のファイルが*とても大きい* VB6 プロジェクトを徐々に移行するのに大いに役立ちます。また、大きいサイズのファイルの移行プロセスのスピードアップや移行プロセスにおいての OutOfMemory エラーを回避するのに役立ちます。プロジェクト、ファイル、メソッドレベルにこのプラグマを指定することができます。(変数レベルで適用する場合は、このプラグマは無視されます)

For example, the following pragmas force VB Migration Partner to exclude statements in all methods in the current project. Notice that you can omit the second argument if it is True:

例として、以下のプラグマは現行プロジェクトの全てのメソッドの全てのステートメントを除外することを VB Migration Partner に指示します。それが True の場合、2 番目の引数を省略できることに注意してください。

```
'## project:SetOption ExcludeMethodBody
```

When applied at the project level, the SetOption ExcludeMethodBody pragma must be included in a *.pragmas file.

プロジェクトレベルに適用する場合は、*.pragmas ファイルの中に SetOption ExcludeMethodBody プラグマを含めなければなりません。

UseTryCatch *boolean, maxExecLines*

Specifies whether VB Migration Partner should attempt to convert error-handling code inside a method by means of structured error handling based on the Try…Catch block. The first optional argument can be True (or omitted) if you want to use structured exception handling, or False to suppress this feature (can be useful to override a UseTryCatch pragma with a broader scope. The second optional argument *maxExecLines* is the max number of executable statements in a method that contains an On Error Resume Next statement (default value is 2).

このプラグマは、VB Migration Partner が、Try…Catch ブロックに基づく構造化されたエラー処理によってメソッド内部でエラー処理コードを変換するのを試みるべきかどうか指定します。構造化された例外処理を使いたい場合、最初の任意の引数は True(または省略)に設定することができます。また、False にするとこの機能を無効にします。(より広範なスコープの UseTryCatch プラグマを無効にするのに便利です。)2 番目の任意の *maxExecLines* 引数は On Error Resume Next ステートメント(デフォルト値は 2)を含むメソッド内の実行ステートメントの最大行数です。

Here's an example of this pragma:

以下はこのプラグマの例です。

```
Sub Test()
    '## UseTryCatch
    On Error Goto ErrorHandler
    ' method body
 ErrorHandler:
    ' error handler
 End Sub
```

This is how the code is translated to VB.NET:

VB.NET にどのように変換されたかを以下に示します。

```
Sub Test()
    Try
        ' IGNORED: On Error Goto ErrorHandler
        ' method body
```

```
        Catch _ex As Exception
            ' IGNORED: ErrorHandler:
            ' error handler
        End Try
    End Sub
```

The Try…Catch block can be inserted only if the following conditions are all true:

Try…Catch ブロックは以下の条件が全て True の場合にのみ挿入することができます。

a. The method contains only a single On Error GoTo < label > method.
   メソッドが On Error GoTo < label > メソッドをひとつだけ含んでいる場合。
b. The On Error GoTo < label > statement doesn't appear inside a conditional block such as If, For, Select Case, and isn't preceded by a GoTo statement.
   On Error GoTo < label > ステートメントが条件ブロック（If、For、Select Case、GoTo ステートメントでコードが先に進まないなど）内部に現れない場合。
c. The method doesn't contain GoSub, Resume, or Resume Next statements. (Resume < label > statements are acceptable, though)
   メソッドが GoSub、Resume、Resume Next ステートメントを含まない場合。（Resume < label > ステートメントは許容されます。）
d. There is no Goto statement in the method body that points to a label that is defined in the error handler. (Notice that it is ok if a GoTo in the error handler points to a label defined in the method body.)
   エラー処理内で定義された Label を指すメソッド本体に GoTo ステートメントがない場合。（メソッド本体内に定義された Label へ指すエラー処理内の GoTo は問題ないことに注意して下さい。）
e. If the method contains one or more On Error Goto 0 statements, such statements must immediately precede a statement that causes exiting from the current method, e.g. Exit Sub or End Function.
   メソッドがひとつ以上の On Error GoTo 0 ステートメントを含む場合、そのようなステートメントは即座に現在のメソッドから抜き出ることになるステートメントに先行しなければなりません。（例 Exit Sub、End Function）
f. If the current method is a Property Let procedure, there must not be a Property Set procedure for the same property. Likewise, If the current method is a Property Set procedure, there must not be a Property Let procedure for the same property.
   現在のメソッドが Property Let　プロシージャである場合、同じプロパティへの Property Set プロシージャであってはなりません。同様に、現在のメソッドが Property Set　プロシージャである場合、同じプロパティへの Property Let プロシージャであってはなりません。
g. The execution flow is such that it can enter the error handler only when an error occurs, which means that the label the precedes the error handler must not be the target of a GoTo statement in method body and the statement that precedes the error handler must be an Exit Sub/Function keyword or a Goto statement, so that execution cannot "flow" into the error handler from the method body.
   処理の流れは次のように行われます。エラーが発生したときのみエラーハンドラーに入ります。すなわ

ち、エラーハンドラーの前にあるラベルはメソッドボディの中の Goto ステートメントのターゲットになってはならず、また、エラーハンドラーの前にあるステートメントは Exit Sub/Function キーワードか Goto ステートメントでなければなりません。こうすることで処理はメソッドボディからエラーハンドラーに流れなくなります。

For example, the following VB6 code:

以下の VB6 コードを御覧下さい。

```
Sub Test()
    '## UseTryCatch
    On Error Goto ErrorHandler
    ' method body
    If x > 0 Then GoTo ErrorHandler2

ErrorHandler:
    ' error handler
ErrorHandler2:
    ' do something here
End Sub
```

can't be converted to VB.NET using a Try-Catch block because a GoTo statement in the method body points to a label that is defined in the error handler.

Try-Catch ブロックを使用するように VB.NET に変換されません。メソッド本体の GoTo ステートメントがエラー処理で定義された Label を指すからです。

By default, VB Migration Partner inserts a Try-Catch also if the method contains the On Error Resume Next statement plus another executable statement. For example, the following VB6 code:

デフォルトで、メソッドが On Error Resume Next ステートメントと別の実行ステートメントを含んでいるのであれば、VB Migration Partner は Try-Catch を挿入します。以下の VB6 コードを御覧下さい。

```
Function Reciprocal(ByVal x As Double) As Double
    '## UseTryCatch True
    On Error Resume Next
    Reciprocal = 1 / x
    ' returns zero if an error occurs (e.g. X is zero)
End Function
```

contains only two executable statements (including On Error Resume Next) and is converted to:

たった 2 行の実行ステートメント(On Error Resume Next を含む)を含んでおり、また、次のように変換されます。

```
Function Reciprocal(ByVal x As Double) As Double
```

```
        Try
            Return 1 / x
            ' returns zero if an error occurs (e.g. X is zero)
        Catch
            ' Do nothing if an error occurs
        End Try
    End Function
```

Methods with three or more executable statements can be converted using Try Catch if you specify a value higher than 2 in the second argument for the UseTryCatch pragma. For example, all methods with up to 5 executable statements under the scope of the following pragma:

UseTryCatch プラグマの 2 番目の引数に 2 よりも高い値を指定した場合は、3つ以上の実行ステートメントを持つメソッドは Try-Catch を使用する変換が行われます。例として、以下のプラグマのスコープ下に最大 5 行の実行ステートメントを持つ全てのメソッドがあります。

```
    '## UseTryCatch True, 5
```

are converted using a Try-Catch block.

Try-Catch ブロックを使用し、変換されます。

Important Note:
the UseTryCatch pragma can be used together with the AutoDispose Force pragma, in which case VB Migration Partner generates a complete Try-Catch-Finally block. However, for the VB.NET code to be perfectly equivalent to the original VB6 code it is essential that the On Error Goto statement be located at the very top of the method body.

UseTryCatch プラグマは AutoDispose Force プラグマと一緒に使用することができます。その場合、VB Migration Partner は完全な Try-Catch-Finally ブロックを生成します。しかし、VB.NET コードが元の VB6 コードに完全に対応しているので、On Error GoTo ステートメントがメソッド本体の最も上位の位置に配置されていることは必須となります。

WrapDeclareWithCallbacks *boolean*

Specifies whether VB Migration Partner must generate additional code to ensure that delegate objects passed to Declare statements aren't orphaned during garbage collections:

VB Migration Partner が、Declare ステートメントに渡されたデリゲイトオブジェクトが、ガーベージコレクションの間に置き忘れられないように追加コードを生成するように指定します。

```
    '## project:WrapDeclareWithCallbacks True
```

5.5 Pragmas that affect forms and controls／フォームとコントロールに影響を与えるプラグマ

BringToFront

Changes the z-order of a control and brings it in front of all other controls. This pragma can be only scoped at the control level. When using this pragma for two or more controls on the same form, the last control becomes the topmost control:

このプラグマは、コントロールの z-order を変更し、それを最前面に配置します。このプラグマはコントロールレベルのみスコープとすることができます。同一フォーム上の二つ以上のコントロールに対してこのプラグマを利用するときは、最後のコントロールが最前面のコントロールになります。

```
'## Rem bring the ListView1 in front of all other controls, except Text1
'## ListView1.BringToFront
'## Text1.BringToFront
```

ChangeProperty *propertyname, newvalue [, controltype] [, controlname]*

Modifies the value of a given property for all occurrences of a control of given name or type.
The *propertyname*argument is the name of the property to be affected, *newvalue* is the new property value. The *controltype*argument is a regex that identifies the type of the controls to be affected and must match the complete name of the control's type (e.g. "VB.TextBox"); if omitted, controls are affected regardless of their type. The*controlname* argument is a regex that identifies the name of the controls to be affected by the pragma; if omitted, all controls are affected regardless of their name. The *propertyname, controltype,* and *controlname*arguments are case-insensitive.

propertyname 引数は変更するプロパティの名称、*newvalue* は新しいプロパティ値です。*Controltype* 引数は、変更するコントロールの型を特定する正規表現です。またこの引数はコントロールの型の名称（例「VB.TextBox」）に完全に一致していなければなりません。省略した場合、コントロールはその型に関わらず影響を受けます。
Controlname 引数はプラグマによって影響を受けるコントロールの名称を特定する正規表現です。省略した場合、全てのコントロールはその名称に関わらず影響を受けます。*Propertyname*、*Controltype*、*Controlname* 引数は大文字小文字を区別しません。

The pragma can have project or form project, therefore a single pragma can affect controls that are hosted in different forms; because the first two arguments are regexes, a single pragma can affect multiple controls of same or different type.

このプラグマはプロジェクトまたは Form プロジェクトを持つことができます。そのため、ひとつのプラグマで異なる form に配置されたコントロールに影響を与えることができます。すなわち、最初の 2 つの引数は正規表現なので、ひとつのプラグマで同じ型と異なる型の複数のコントロールに影響を与えることができます。

```
'## Rem all command buttons in current form  whose name begins with "cmd" must be 400 twips high
'## ChangeProperty Height, 400, "VB¥.CommandButton", "cmd.+"

'## Rem reset the Text property of all TextBox controls in all forms
```

```
'## project:ChangeProperty Text, "", "VB¥.TextBox"
```

The name of controls that belong to a control array include the index between parenthesis:

コントロール配列に属するコントロールの名称は括弧内のインデックスを含みます。

```
'## Rem reset the Text property for all the elements of the "txtFields" control array
'## ChangeProperty Text, "", "VB¥.TextBox", "txtFields¥(¥d+¥)"
```

You can also perform math operations on the current property value, by prefixing the *newvalue* argument with the +, −, *, /, ¥ symbols:

+, −, *, /, ¥ 記号を伴う *newvalue* 引数によって、現在のプロパティ値に演算操作を実行することもできます。

```
'## Rem expand the width of all command buttons in current form by 200 twips
'## ChangeProperty Width, +200, "VB¥.CommandButton"

'## Rem ensure that all textboxes on current form are multilined and double their height
'## ChangeProperty Height, *2, "VB¥.TextBox"
```

The *newvalue* argument can be preceded by the "=" symbol, which allows you to set a negative value for a numeric property or to assign a string property that begins with a math symbol:

newvalue 引数は「=」記号の後に記述することができます。これによって、数値プロパティに負の値を設定したり、演算記号で始まる文字列プロパティを割り当てることができます。

```
'## Rem set the ListIndex property to -1 for all ListBox controls
'## ChangeProperty ListIndex, =-1, "VB¥.ListBox"

'## Rem set the Caption property to "+1" for controls named "lblIncr" or "cmdIncr"
'## project:ChangeProperty Caption, "=+1", , "(lblIncr|cmdIncr)"
```

FormFont *fontname [,fontsize]*

Set the default font for a specific form or (if the pragma has project scope) all the forms in the application. The*fontname* argument is the name of the form (embedded in double quotes if contains spaces), the optional*fontsize* argument is the size of the font:

このプラグマは、特定のフォームや(プラグマがプロジェクトをスコープにしている場合には)アプリケーションの全てのフォームにデフォルトのフォントをセットします。*fontname* 引数は(半角スペースを含む場合は二重引用符で囲まれた)フォームの名称であり、任意の *Fontsize* 引数はフォントのサイズです。

```
'## Rem set the default font for forms that don't define a custom font
```

```
'## project:FormFont Arial, 10
```

This pragma allows you to specify a different default font for forms; it affects only fonts that use the default font and don't specify a specific font. If the *fontsize* argument is omitted, the original font size is retained.

このプラグマによって各フォームに対して異なる初期フォントを指定することができます。すなわち、デフォルトフォントを使っており、特定のフォントを指定しないフォントにのみに効果があります。*fontsize* 引数を省略した場合はオリジナルのフォントサイズが適用されます。

KeepObsoleteProperties *boolean*

Specifies whether VB Migration Partner should discard assign design-time assignments to properties that aren't supported under VB.NET. Such properties are marked as obsolete in the control language library and, by default, they aren't included in the converted project. If the argument is True or omitted, assignments to these properties are preserved in the converted VB.NET project:

このプラグマは、VB.NET 下でサポートされていないプロパティに対して行われたデザイン時の設定を VB Migration Partner に破棄させるべきかどうかを指定します。そのようなプロパティはコントロール言語ライブラリでは無視されるものと見なされており、デフォルトでは、それらは変換されたプロジェクトには含まれません。引数が True または省略された場合は、これらのプロパティの設定は変換された VB.NET プロジェクトに引き継がれます。

```
'##  KeepObsoleteProperties True
```

You can scope this pragma at the project, file, and individual control level. This pragma can be useful to browse all the properties that are usually discarded during the migration process.

このプラグマはプロジェクトレベル、ファイルレベル、個々のコントロールレベルをスコープとすることができます。このプラグマはマイグレーション工程で通常は破棄される全てのプロパティを閲覧するのに役立ちます。

ReplaceFont *oldfontname, newfontname*

Replace all occurrences of a font with another font, for all controls in current form or in all forms in current project (depending on the pragma's scope):

"このプラグマは、(プラグマのスコープに従って)現在のフォームまたは現在のプロジェクトのすべてのコントロールに対して発見されたフォントを他のフォントに置き換えます。"

```
'## Rem replace all occurrences of MS Sans Serif with Helvetica font
'## project:ReplaceFont "MS Sans Serif", "Helvetica"
```

VB.NET doesn't support system fonts, a group which includes the following fonts: Courier, Fixedsys, Modern, MS Sans Serif, MS Serif, Roman, Script, Small Fonts, System, and Terminal. Because of this limitation, VB Migration Partner converts Courier and Fixedsys fonts to Courier New and all other fonts to Arial, while preserving the font size. You can use this pragma to enforce a different substitution schema.

VB.NET は次のフォントを含むグループとシステムフォントとをサポートしません。すなわち、Courier、Fixedsys、Modern、MS Sans Serif、MS Serif、Roman、Script、Small Fonts、System、Terminal です。この制約によって、VB Migration Partner は Courier と Fixedsys フォントは Courier New に、その他のフォントは Arial に変換します。しかしながら、フォントサイズは引き継がれます。異なる置換計画を実施するために、このプラグマを使用することができます。

ReplaceFontSize *oldfontname, oldfontsize, newfontname, newfontsize*

Replaces all occurrences of a font/size combination with a different font/size combination, for all controls in current form or in all forms in current project (depending on the pragma's scope):

このプラグマは、(プラグマのスコープに従って)現在のフォームまたは現在のプロジェクトのすべてのフォームのすべてのコントロールに対して、発見されたすべての font/size の組み合わせを異なる font/size の組み合わせに置換します。

```
'## Rem replace all occurrences of MS Sans Serif 12 with Helvetica 10 font
'## project:ReplaceFont "MS Sans Serif", 12, "Helvetica", 10
```

See the ReplaceFont pragma for more information on the automatic font replacement mechanism that VB Migration adopts.

VB Migration Partner が採用する自動フォント置換機構に関する詳細は ReplaceFont プラグマを参照してください。

SendToBack

Changes the z-order of a control and sends it behind all other controls. This pragma can be only scoped at the control level. When using this pragma for two or more controls on the same form, the first control becomes the bottommost control:

このプラグマは、コントロールの z-order を変更し、他のすべてのコントロールの後ろにそれを配置します。このプラグマはコントロールレベルのみスコープとすることができます。同じフォームの2つ以上のコントロールにこのプラグマを使用する際、最初のコントロールは最背面のコントロールになります。

```
'## Rem send the ListView1 behind all other controls, except Text1
'## Text1.BringToFront
'## ListView1.BringToFront
```

WriteProperty *propertyname, value*

Assigns a design-time property to the current form or to a specific control:

このプラグマは、デザイン時のプロパティを現在のフォームや特定のコントロールに割り当てます。

```
'## Rem change the caption of current form
'## WriteProperty Text, ".NET Framework options"
'## Rem set the  DTPicker1.Format property equal to 2
```

```
'## DTPicker1.WriteProperty Format, 2
```

This pragma is useful in many cases, for example to manually assign a control property that VB Migration Partner can't migrate correctly (as is the case of the DTPicker's Format property) or to purposely introduce minor differences between the VB6 and VB.NET applications.

このプラグマは多くのケースにおいて役立ちます。例えば、(DTPicker の Format プロパティのように)VB Migration Partner が正しく変換できないコントロールプロパティを手作業で割り当てる場合や、VB6 と VB.NET アプリケーション間のわずかな仕様の違いを意図的に比較する場合です。

5.6 Pragmas that affect user controls／ユーザコントロールに影響を与えるプラグマ

TranslateProperties *propertylist*

Specifies a list of properties that have a different name in the .frm file. The *propertylist* argument is a comma-delimited list of *oldname=newname* pairs, as in this example:

このプラグマは、.frm ファイル内の異なる名称を持つプロパティのリストを指定します。*propertylist* 引数は*古い名称=新しい名称*のカンマ区切りのリストで、次のようになります。

```
'## TranslateProperties "FC=ForeColor,BC=BackColor"
```

When storing properties in a .frm file, VB6 doesn't necessarily use the property name; more precisely, the actual name used in the .frm file is equal to the name specified when saving the value in the WriteProperties event handler. If the name specified in the event handler is different from the property name, you should use this pragma to tell VB Migration Partner which name is used in .frm files.

.frm ファイルにプロパティを保存する場合、VB6 は必ずしもプロパティ名称を必要としません。より正確には、.frm ファイルで使用されている実際の名称は WriteProperties イベントハンドラの値を保存する際に指定される名称と同じです。イベントハンドラで指定された名称がプロパティ名称と異なる場合、.frm ファイルで使用される名称を VB Migration Partner に伝えるためにこのプラグマを利用するべきです。

This pragma can't have a scope prefix and is implicitly scoped at the file level. It is ignored if it appears in a VB6 file other than a .ctl (user control) class.

このプラグマはスコープを持つことができず、暗黙的にファイルレベルをスコープとします。(ユーザコントロールの).ctl クラス以外の VB6 ファイルに存在する場合は無視されます。

TranslateEvents *eventlist*

Specifies a list of events that must be renamed when migrating a UserControl class. The *eventlist* argument is a comma-delimited list of *oldname=newname* pairs, as in this example:

このプラグマは、ユーザコントロールのクラスを変換する際に、名前を変更しなければならないイベントのリストを指定します。*eventlist* 引数は*古い名称=新しい名称*のカンマ区切りのリストで、次のようになります。

```
'## TranslateEvents "KeyPress=KeyPress6,KeyDown=KeyDown6,KeyUp=KeyUp6"
```

When this pragma is used, VB Migration Partner marks the converted UserControl class with a TranslateEvents attribute. In turn, this attribute is taken into account when VB Migration Partner converts any VB6 form that contains one or more instances of the user control. Renaming an event is sometimes necessary because the original name used under VB6 conflicts with events exposed by the .NET UserControl class.

このプラグマが使用される場合、VB Migration Partner は TranslateEvents 属性をもつ変換済みユーザコントロールクラスを識別します。言い換えると、ユーザコントロールのひとつ以上のインスタンスを含んでいる VB6Form を VB Migration Partner が変換する際にこの属性が考慮されます。VB6 で使用される元のイベントの名称が.NET ユーザコントロールのクラスが公開しているイベントと衝突することがある場合、イベントの名称を変更しなければなりません。

This pragma can't have a scope prefix and is implicitly scoped at the file level. It is ignored if it appears in a VB6 file other than a .ctl (user control) class.

このプラグマはスコープを持つことができず、暗黙的にファイルレベルをスコープとします。(ユーザコントロールの).ctl クラス以外の VB6 ファイルに存在する場合は無視されます。

IgnoreMembers *memberlist*

Specifies a list of properties and methods that should be ignored when translating a VB6 user control. The*memberlist* argument is a pipe-delimited list of member names:

このプラグマは、VB6 ユーザコントロールを変換する際に、無視されるべきプロパティとメソッドのリストを指定します。*memberlist* 引数は「|」で区切られたメンバ名称のリストです。

```
'## IgnoreMembers "Appearance|Bindings|CompanyName"
```

This pragma can't have a scope prefix and is implicitly scoped at the file level. It is ignored if it appears in a VB6 file other than a .ctl (user control) class.

このプラグマはスコープを持つことができず、暗黙的にファイルレベルをスコープとします。(ユーザコントロールの).ctl クラス以外の VB6 ファイルに存在する場合は無視されます。

5.7 Pragmas that insert or modify code／コードを挿入または変更するプラグマ

Note:

This group of pragmas allow you to control how VB Migration Partner parses VB6 statements or outputs VB.NET code. None of these pragmas can have a scope, because they have an immediate effect on the code that follows.

このプラグマグループは VB Migration Partner がどのように VB6 ステートメントを解析するか、どのように VB.NET コードを出力するのかを制御するためのものです。次に示すようにコードに対して直接効果を持つのでこれらのプラグマはスコープを持つことができません。

InsertStatement *code*

Inserts a statement in the converted application. The *code* argument is an (unquoted) string that represents a valid VB.NET statement (or statements):

このプラグマは、変換されたアプリケーションにステートメントを挿入します。*code* 引数は有効な VB.NET ステートメント(あるいは複数の VB.NET ステートメント)を表す(引用符で閉じられた)文字列です。

```
'## InsertStatement MsgBox6("End of  processing"): Exit Sub
```

Note *text*

Inserts an UPGRADE_NOTE remark in the converted VB.NET:

このプラグマは、変換された VB.NET に UPGRADE_NOTE コメントを挿入します。

```
'## Note Double-check following statement
x =  Abs(y) Mod y
```

OutputMode *mode [,count]*

Sets VB Migration Partner's output mode. The *mode* argument can be one of the following: *On* (enables code generation), *Off* (disable code generation), *Remarks* (generate remarked out code)), *Uncomment* (removes a leading apostrophe). The count argument is the number of VB.NET statements affected by this pragma; if zero or omitted, the effect extends to the next OutputMode pragma:

このプラグマは、VB Migration Partner の出力モードを設定します。Mode 引数は次の中のひとつを設定することができます。すなわち、(コード変換を有効にする) *On*、(コード変換を無効にする) *Off*、(コメント化されたコードを生成する) *Remarks*、(先頭のアポストロフィを削除する) *Uncomment* です。Count 引数はこのプラグマの効果が及んだ VB.NET ステートメントの行数です。0 もしくは省略された場合、その効果は次の OutputMode プラグマまで拡張されます。

```
'## Rem don't output the next line (two statements, counting the remark)
'## OutputMode Off, 2
PrintReport  "Products", 1, 10    ' print pages 1-10 of the Products Report

'## Rem don't output the next method
'## OutputMode Off
Private Sub TestReport()
    ' ... (any number of lines here) ' ...
End Sub
'## OutputMode On
```

The Uncomment setting is especially useful for including a portion of VB.NET code that doesn't exist in the original VB6 project:

Uncomment 設定は元の VB6 プロジェクトに存在しない VB.NET コードの一部を含めるのに特に役立ちます。

```
'## OutputMode Uncomment
'' This is a piece of VB.NET code
'Debug.Print("x={0}", x)
'  ...
'  ...
'## OutputMode On
```

ParseMode *mode [,count]*

Sets VB Migration Partner's parsing mode. The *mode* argument can be one of the following: *On* (enables parsing), *Off* (disable parsing), *Remarks* (consider next statements as remarks). The count argument is the number of VB6 statements affected by this pragma; if zero or omitted, the effect extends to the next ParseMode pragma:

このプラグマは、VB Migration Partner の解析モードを設定します。*mode* 引数は次のひとつを設定できます。すなわち、(解析を有効にする) *On*、(解析を無効にする) *Off*、(次のステートメントをコメントと見なす) *Remarks*。*count* 引数はこのプラグマの効果が及ぶ VB.NET ステートメントの行数です。0 もしくは省略された場合、その効果は次の ParseMode プラグマまで拡張されます。

```
'## Rem don't parse the next line (two statements, counting the remark)
'## ParseMode Off, 2
PrintReport  "Products", 1, 10    ' print pages 1-10 of the Products Report
'## Rem don't parse the next method
'## ParseMode Off

Private Sub TestReport()
    ' ... (any number of lines here) ' ...
End Sub
'## ParseMode On
```

Notice the subtle difference between OutputMode and ParseMode pragmas. If parsing is enabled when a variable or method is parsed, VB Migration Partner creates a symbol for that method and correctly solves all references to it; if parsing is disabled, no symbol can be created, which prevents VB Migration Partner from correctly resolving references to that symbol.

OutputMode と ParseMode プラグマには微妙な違いがあることに注意して下さい。変数かメソッドが解析される際に解析が有効な場合、VB Migration Partner はそのメソッドの記号を作成し、それに対する参照を正しく解決します。解析が無効の場合、記号は作成されません。そのため VB Migration Partner はその記号に対する参照を正確に解決することはできません。

If you don't want to include a VB6 member in the VB.NET program, in most cases it is preferable to have VB Migration Partner parse the symbol and then use an OutputMode pragma to omit it during the code generation phase.

VB.NET プログラムに VB6 メンバを含めたくない場合、ほとんどすべてのケースにおいて、コード生成工程では、VB Migration Partner に記号を解析させた後、それを省略するのに OutputMode プラグマを使用するのが望ましいです。

ParseReplace *vb6code*

Replace the following line of VB6 code with a different line. The replacement is performed before the parsing takes place, therefore you can use this pragma to change the way a method is declared, the scope of a variable, and so forth:

このプラグマは、次の VB6 コードの行を異なる行に置き換えます。解析が実行される前に置換が実行されます。そのため、メソッドが宣言される方法、変数のスコープなどを変更するのにこのプラグマを使用することができます。

```
'## ParseReplace Sub Test(Optional Byval arg As Integer = -1)
Sub Test()
```

Notice that this pragma replaces an entire line of code, therefore it can replace multiple statements if the next line contains multiple statements. It can even replace a portion of a statement if the next line has a trailing underscore.

このプラグマはコードの全ての行を置き換えることに注意して下さい。そのため、次の行が複数のステートメントを含む場合は、複数のステートメントを置換することができます。次の行がアンダースコアを含んでいる場合、ステートメントの一部を置き換えることさえできます。

Rem *text*

Inserts a remark in the VB6 source code that won't be translated to VB.NET. This pragma is useful to explain what other pragmas do:

このプラグマは、VB.NET に変換されない VB6 ソースコードにコメントを挿入します。このプラグマは他のプラグマがすることを説明するために役立ちます。

```
'## Rem we use the Shift option to preserve number of array elements
'## arr.ArrayBounds Shift
Dim arr(1 To 10) As Integer
```

RemoveUnreachableCode *mode*

Determines how unreachable code blocks are processed. The *mode* argument can be one of the following three values: *Off* (unreachable code is left in the generated VB.NET code, the default behavior), *On* or omitted (unreachable code is removed from the generated VB.NET code), *Remarks* (unreachable code is remarked out.) Consider the following VB6 code:

このプラグマは、参照されないコードブロックをどのように処理するかを指定します。*mode* 引数は以下の3つの値のひとつにすることができます。すなわち、(デフォルトであり、生成された VB.NET コードに参照されないコードを残す) *Off*、(参照されないコードを生成された VB.NET コードから除外する) *On* または省略、(参照されないコードをコメントにする) *Remarks* です。以下の VB6 コードを御覧下さい。

```
'## RemoveUnreachableCode Remarks
Sub Test(x As Long, y As Long)
    x = 123
    Exit Sub
    x = 456
    y = 789
End Sub
```

This is the VB.NET code that VB Migration Partner generates:

以下は VB Migration Partner で生成された VB.NET コードです。

```
Sub Test(ByRef x As Integer, ByRef y As Integer)
    x = 123
    Exit Sub
    '## UPGRADE INFO (#06F1): Unreachable code removed
    ' EXCLUDED: x = 456
    ' EXCLUDED: y = 789
End Sub
```

If the VB6 code had included the followed pragma:

VB6 コードが以下のプラグマを含んでいる場合、

```
'## RemoveUnreachableCode On
```

then the result would have been as follows:

結果は以下のようになります。

```
Sub Test(ByRef x As Integer, ByRef y As Integer)
    x = 123
    Exit Sub
    '## UPGRADE INFO (#06F1): Unreachable code removed
End Sub
```

Notice that current version of VB Migration Partner might, under some conditions, fail to recognize a piece of code as unreachable. This happens, for example, when a label is referenced by a piece of code that is unreachable. Consider the following code snippet

現在のバージョンの VB Migration Partner はある条件の下では、参照されないコードの一部を識別することに失敗するかもしれないことに注意して下さい。この事例として、あるラベルが、参照されないコードの一部によって参照されている場合があります。以下のコードの切り抜きを御覧下さい。

```
Sub Test(ByRef x As Integer, ByRef y As Integer)
    x = 123
    Exit Sub
Restart:
    ' ...
    If x = 10 Then Goto Restart
End Sub
```

The block of code between Exit Sub and End Sub is unreachable and should be removed. However, the *Restart* label is referenced by the GoTo statement in the unreachable portion of the method, and this detail cheats VB Migration Partner into believing that the label is indeed reachable. We are aware of this limitation and hope to work around it in a future release of the tool.

Exit Sub と End Sub の間のコードブロックは参照されていないので、削除されるべきです。しかし、*Restart* ラベルは参照されないメソッドの一部の GoTo ステートメントによって参照されています。また、この部分によって VB Migration Partner はラベルが参照されていると誤解してしまいます。弊社ではこの変換の制約を認識しており、将来のリリースでは解決できればと考えております。

RemoveUnusedMembers *removeMode, safeMode*

Determines how unused members blocks are processed. The first argument can be one of the following three values: *Off* (unused members are left in the generated VB.NET code, the default behavior), *On* or omitted (unused members are removed from the generated VB.NET code), *Remarks* (unused members are remarked out.) The second argument is True if the pragma must affect only members that can't be reached via late binding. Consider the following VB6 code:

このプラグマは、利用されないメンバブロックの処理の方法を指定します。第1引数は、以下の 3 つの値のうちの 1 つになりえます。つまり、(デフォルトであり、利用されないメンバが作成された VB.NET コードに含まれない) *Off*、(利用されないメンバが作成された VB.NET コードから削除される) *On* または省略、(利用されないメンバをコメントする) *Remarks* です。第 2 引数は、プラグマが、遅延バインディングによってアクセスできないメンバだけに効果を与える場合に、True となります。以下の VB6 コードをご参照下さい:

```
Const UnusedValue As Integer = 1
```

If the constant is never referenced, VB Migration Partner generates the following message:

定数が参照されない場合は、VB Migration Partner は、以下のメッセージを出します。

```
' UPGRADE_INFO (#0501): The 'UnusedValue' member isn't used anywhere in current application
Const UnusedValue As Integer = 1
```

If you use a RemoveUnusedMembers pragma, the message is different. For example:

RemoveUnusedMembers プラグマが使用される場合、メッセージが異なります。例えば、

```
'## RemoveUnusedMembers Remarks
Const UnusedValue As Integer = 1
```

generates this VB.NET code:

以下の VB.NET コードが生成されます:

```
' UPGRADE_INFO (#0701): The 'UnusedValue' member has been remobed because
' isn't used anywhere in current application
' EXCLUDED: Const UnusedValue As Integer = 1
```

If the On option is used, the warning message is emitted but the member itself is removed. You need a DisableMessage pragma to drop the warning.

On オプションが使用されると、警告メッセージが発行されますが、メンバは削除されます。警告メッセージを解除するには DisableMessage プラグマが必要となります。

When applied to methods, the RemoveUnusedMembers pragma used with the Remarks disables a few other refactoring features, for example the ConvertGosubs pragma. In other words, if the following pragmas are active:

メソッドに適用する場合、Remarks を伴う RemoveUnusedMembers プラグマは、例えば ConvertGosubs プラグマのような他のリファクタリング機能を無効にします。つまり、以下のプラグマがアクティブな場合、

```
'## RemoveUnusedMembers Remarks
'## ConvertGosubs
'## UseTryCatch
```

then VB Migration Partner generates (remarked out) nonoptimized VB.NET code, where Gosub keywords are not rendered as separate methods and On Error statements are not converted into Try-Catch blocks.

VB Migration Partner は、最適化されていない VB.NET コードを(コメントにして)生成し、そこでは Gosub のキーワードは独立したメソッドとして変換されず、On Error ステートメントは Try-Catch ブロックに変換されません。

Important note:
the RemoveUnusedMembers pragma deletes or remarks all members that haven't found to be referenced by any other member in the current project or other projects in the current solution. However, this mechanism can lead to unwanted removal if the member is part of a public class that is accessed by another project (not in the solution) or if the member is accessed via late-binding. You can use
the MarkAsReferenced orMarkPublicAsReferenced pragmas to account for the first case.

RemoveUnusedMembers プラグマは、現ソリューションでの、現行のプロジェクトや他のプロジェクトにおけるどのメンバにも参照されていない全てのメンバを削除するか、あるいはコメントにします。しかし、メンバが、(別のソリュー

ションの)他のプロジェクトからアクセスされるパブリッククラスのメンバであった場合や、メンバーが遅延バインディングによってアクセスされた場合は、このメカニズムは、不要な削除を実行するかもしれません。MarkASReferenced や MarkPublicAsReferenced プラグマを前者のケースを解決するために使用することができます。

However, there is no simple way to detect whether a member is accessed via late-binding. For this reason the RemoveUnusedMembers pragma supports a second *safeMode* parameter. If this parameter is True then the pragma affects only members that can't be reached via late-binding, e.g. constants, Declare statements, and members with a scope other than public.

しかし、メンバが遅延バインディングによってアクセスされたかを確認する簡単な方法はありません。その為、RemoveUnsedMembers プラグマは、2 番目の *sefeMode* パラメータをサポートします。このパラメータが True であると、例えば、定数や、Declare ステートメント、パブリック以外のスコープにあるメンバのように、遅延バインディングによってアクセスできないメンバにのみプラグマは効果があります。

ReplaceStatement *code*

Replaces the next statement with a piece of code. The *code* argument is an (unquoted) string that represents a valid VB.NET statement (or statements):

このプラグマは、次のステートメントをコードの一部に置換します。この *code* 引数は有効な VB.NET ステートメント(あるいは複数の VB.NET ステートメント)を表す(引用符で閉じられた)文字列です。

```
'## ReplaceStatement Dim obj As Widget = Widget.Create(12)
Dim  obj As New Widget
```

Keep in mind that this pragma replaces only the VB6 *statement* (not the line) that follows immediately the pragma. If the following line includes two statements separated by a semicolon, only the first statement is replaced by this pragma.

このプラグマが、直後にこのプラグマが続く(行ではない)VB6 の*ステートメント*だけを置換することに留意して下さい。もし、以下のラインがセミコロンで分離されている2つのステートメントを含む場合は、最初のステートメントのみがこのプラグマによって置換されます。

5.8 Pragmas that affect upgrade messages／アップグレードメッセージに影響を与えるプラグマ

Note:

This group of pragmas allow you to select which upgrade messages are generated during the migration process. VB Migration Partner can generate issue, warning, info, and to-do messages and, by default, all of them are included as remarks in the converted VB.NET application. You can selectively disable and enable messeges by their ID, by their severity (Issue, Warning, Info, ToDo), and type (Syntax, Language, CodeAnalysis, Library)

このグループのプラグマによって、どのアップグレードメッセージがマイグレーション中に生成されるかが選択されます。VB Migration Partner はデフォルトで、課題、警告、TODO メッセージを生成しますが、その全てが変換された VB.NET アプリケーションに、コメントとして含まれます。ID、重要性(課題、警告、TODO)、タイプ(文法、言語、コード分析、ライブラリ)等の選択によって、メッセージを無効にしたり有効にしたりすることができます。

DisableMessage *messageid*

Disables generation for a specific message. The *messageid* argument is the hex numeric value of the message:

このプラグマは、特定のメッセージの生成を無効にします。*messageidID* 引数は、16 進数のメッセージ ID です。

```
'## Rem disable the message  “Controls of type XYZ aren’t supported…”
'## DisableMessage 02C8
```

DisableMessages *category*

Disables message generation for a specific language category. The argument can be: *Info, ToDo, Warning, Issue, Syntax, Language, CodeAnalysis, Library, All*:

このプラグマは、特定の言語カテゴリーのメッセージ生成を無効にします。この引数は、*Info*、*ToDo*、*Warning*、*Issue*、*Syntax*、*Language*、*CodoAnalysis*、*Library*、*All* のどれかになりえます。

```
'## Rem disable all messages except issues and warnings
'## DisableMessages All
'## EnableMessages Issue
'## EnableMessages Warning
```

EnableMessage *messageid*

Enables generation for a specific message. The *messageid* argument is the hex numeric value of the message:

このプラグマは、特定のメッセージ生成を有効にします。*messageid* 引数は 16 進数のメッセージ ID です。

```
'## Rem disable all messages except  “Controls of type XYZ aren’t supported…”
'## DisableMessages All
'## EnableMessage 02C8
```

EnableMessages *category*

Enables message generation for a specific language category. The argument can be: *Info, ToDo, Warning, Issue, Syntax, Language, CodeAnalysis, Library, All*:

このプラグマは、特定の言語カテゴリーについてのメッセージ作成を有効にします。この引数は、*Info*、*ToDo*、*Warning*、*Issue*、*Syntax*、*Language*、*CodoAnalysis*、*Library*、*All* のどれかになりえます。

```
'## Rem disable messages except those related to code analysis
'## DisableMessages All
```

```
'## EnableMessages CodeAnalysis
```

MarkAsReferenced

Specifies that a specific symbol should be considered as used even if VB Migration Partner can't detected any reference to it. It is useful when the member is accessed through late binding or the CallByName method:

このプラグマは、例え VB Migration Partner が、それについての参照を検出しなくても、特定の記号が使用されたとみなされるよう特定します。これは、メンバが遅延バインディングや CallByName メソッドによりアクセスされる場合に役立ちます。

```
'## Rem consider the PrintReport method as referenced
'## PrintReport.MarkAsReferenced
```

Notice that this pragma must have a scope prefix.

このプラグマはスコープを表す接頭辞が必要であることにご注意下さい。

MarkPublicAsReferenced

Specifies that all public members of current user control or class should be considered as used even if VB Migration Partner can't detected any reference to it. It is useful inside ActiveX DLL or EXE projects that aren't migrated together with their clients:

このプラグマは、例え VB Migration Partner が、それについての参照を検出しなくても、現在のユーザーコントロールやクラスの全てのパブリックメンバが使用されたとみなされるように指定します。これは顧客が移行しない ActiveX DLL プロジェクトや EXE プロジェクトで役立ちます。

```
'## Rem mark all public members in all public classes as referenced
'## project:MarkPublicAsReferenced
```

5.9 Miscellaneous pragmas／その他のプラグマ

PreCommand *cmdtext*

Executes an external application or an operating system command before starting the conversion of current project; *cmdtext* is the name of the external application or O.S. command, including its path if the application can't be found on system path. If the command is an internal O.S. command (e.g. Copy) or the name of a batch file, the command should be prefixed by **cmd /C** and be run through the command-line interpreter:

このプラグマは、現行のプロジェクトを変換し始める前に外部のアプリケーションや OS のコマンドを実行します。*cmdtext* は、外部のアプリケーションや OS コマンドの名前で、アプリケーションがシステムパスにない場合は、そのパスを含みます。コマンドが内部の OS コマンド(例えば Copy)やバッチファイルの名前だった場合は、そのコマンドは、**cmd** か **C** を接頭辞として付けなければならず、コマンドラインインタープリタによって作動します。

```
'## PreCommand notepad.exe c:¥docs¥instructions.txt
```

```
'## PreCommand cmd /C copy c:¥vbproject¥*.* c:¥backup
```

cmdtext can include the following special placeholders (not case sensitive)

cmdtext は、以下の特別なプレースフォルダ(大文字と小文字の区別は無視されます)を含みます。

- **${vb6projectfile}**:
  the complete path+name of the VB6 .vbp project file
  VB6.vbp プロジェクトファイルの完全な path+name
- **${vb6projectpath}**:
  the path of the folder containing the VB6 .vbp project file
  VB6.vbp プロジェクトファイルを含むフォルダパス

It is a good idea to enclose these placeholders between double quote, in case the project name or path include spaces. For example, you can use the PreCommand pragma to restore all VB6 files from a backup folder before starting the migration:

プロジェクト名やパスが半角スペースを含む場合、これらのプレースフォルダを二重引用符で囲むことは効果的です。例えば、移行開始前に、全ての VB6 ファイルをバックアップフォルダから復元する為に、PreCommand プラグマを使用できます。

```
'## PreCommand cmd /c copy "${vb6projectpath}¥backup¥*.*" "${vb6projectpath}"
```

The PreCommand pragma can only have an (implicit or explicit) project scope. You can apply the PreCommand pragma only by storing it in a *.pragmas file. We suggest that you use the special file named VBMigrationPartner.pragmas, which you can conveniently share among all the projects of a complex application, and use project placeholders to differentiate between individual projects.

PreCommand プラグマは(明示的または暗黙的な)プロジェクトスコープのみ持つことができます。また、このプラグマは、*.pragma ファイルに格納しないと利用できません。弊社は VB Migration Partner.pragma という名前の専用ファイルと – このファイルは複雑なアプリケーションを持つ全てのプロジェクト間で共有することができます – 個々のプロジェクトを区別するプロジェクトプレースフォルダを使用することを推奨します。

PostCommand *cmdtext*

Executes an external application or an operating system command after saving the migrated .NET version of current project; *cmdtext* is the name of the external application or O.S. command, including its path if the application can't be found on system path. If the command is an internal O.S. command (e.g. Copy) or the name of a batch file, the command should be prefixed by **cmd /C** and be run through the command-line interpreter:

このプラグマは、移行された現行プロジェクトの.NET バージョンを保存した後に、外部のアプリケーションや、OS システムを実行します。*cmdtext* は、外部アプリケーションや OS コマンドの名前で、もしアプリケーションがシステムパスで見つけられない場合は、そのパスを含みます。もし、コマンドが、内部 OS(例えば Copy)であったり、バッチファイルの名前であったりした場合、そのコマンドは、**cmd** か **C** を接頭辞として付けなければならず、コマンドラインインタープリターによって動きます。

```
'## PostCommand notepad.exe "c:¥docs¥after migration instructions.txt"
'## PostCommand cmd /C copy c:¥vbproject¥*.* c:¥backup¥vbproject
```

cmdtext can include the following special placeholders (not case sensitive)

cmdtext は以下の専用のプレースフォルダ、(大文字と小文字の区別を無視されます)を含みます。

- **${vb6projectfile}**:
  the complete path+name of the VB6 .vbp project file
  VB6.vbp プロジェクトファイルの完全な path+name
- **${vb6projectpath}**:
  the path of the folder containing the VB6 .vbp project file
  VB6.vbp プロジェクトファイルを含むフォルダパス
- **${ProjectName}**:
  the .NET name of the project that has been converted
  変換されたプロジェクトの.NET 名
- **${ProjectFile}**:
  the path of the .NET project that has been converted
  変換された.NET プロジェクトのパス
- **${ProjectPath}**:
  the path of the folder containing the converted .NET project
  変換された.NET プロジェクトを含んでいるフォルダパス

It is a good idea to enclose these placeholders between double quote, in case the project name or path include spaces. For example, you can use the PostCommand pragma to copy all VB.NET source files to a backup folder immediately after saving them

プロジェクト名やパス名に半角スペースが含まれる場合に、これらのプレースフォルダを二重引用符で囲むことはとても効果的です。例えば、PostCommand プラグマで全ての VB.NET のソースファイルをコピーし、保存した直後にバックアップフォルダを作成出来ます。

```
'## PostCommand cmd /c copy "${projectpath}¥*.*" "c:¥backup¥${vb6projectname}"
```

The PostCommand pragma can only have an (implicit or explicit) project scope.

PostCommand プラグマは(明示的または暗黙的な)プロジェクトスコープのみ持つことができます。

The PostCommand pragma is very helpful to preserve one or more .NET “definitive” source files that have been accurately edited and refined and that you don’t want to be overwritten in subsequent migrations. VB Migration Partner always deletes all the files in the target .NET directory, therefore you can’t really prevent a file from being overwritten; however, you can store all these “definitive” files in a given folder and use a PostCommand pragma to ensure that these files are copied over the files produced by the migration that has just been completed:

PostCommand プラグマは、正確に修正または改善した、以降の変換では上書きしたくない、1 つ以上の「最終的な」.NET ソースファイルを保存するのにとても役立ちます。VB Migration Partner は、常にターゲットの.NET ディレク

トリの全てのファイルを削除するので、実際にはファイルの上書きを防ぐことはできません。しかし、全ての「最終的な」ファイルを特定のフォルダに保存するために、PostCommand プラグマを使用して変換によって生成されたファイルをこれらのファイルに上書きコピーすることが出来ます。

```
    '## PostCommand cmd /c copy "c:¥definitiveprj¥${projectname}¥*.*"
"${projectpath}" /Y
```

This technique is especially useful to preserve *.designer.vb files containing the code-behind portions of converted .NET forms.

このテクニックは、変換された.NET フォームのコードビハインド部分を含む*designer.vb ファイルを保存する時に特に便利です。

SetTag *tagname, tagvalue*

Creates a tag with a given name and value. This pragma can be only useful when a VB Migration Partner extender recognizes and uses it. The following example assumes that you have installed an extender that recognizes the pragma tag named “DatabaseName”

このプラグマは、特定の名前と値を持つタグを作ります。このプラグマは VB Migration Partner のエクステンダーがそれを認識し、使用した時のみ役立ちます。以下のサンプルは、DatabaseName という名前のプラグマタグを認識するエクステンダーをインストールすることを想定しています。

```
    '## SetTag DatabaseName,  “SqlServer”
```

Appendix A. ADOLibrary／付録 A ADOLibrary

One of the most complex problems in migrating a VB6/COM application to .NET is the conversion of legacy database access techniques to ADO.NET.

VB6 と COM のアプリーケーションを.NET に移行する場合のもっとも複雑な問題のひとつは ADO.NET. に対するレガシーなデータベース接続技法の変換です。

ADO.NET differs from all the database access libraries that were popular in the VB6 days, including ADODB, DAO, and RDO. Most notably, ADO.NET promotes the so-called “disconnected” approach to database processing: data is brought from the server to the client, where it can be processed locally, and then all (and only) the changes are uploaded back to the database. Another important difference is that ADO.NET doesn’t offer any kind of support for server-side cursors, such as keysets and dynamic cursors.

ADO.NET は ADODB や DAO、RDO のような VB6 時代によく使われていたどんなデータベース接続ライブラリとも異なります。もっとも顕著なのは、ADO.NET がいわゆる「disconnected」アプローチをデータベース処理に用いていたことです。すなわち、サーバーからクライアントに伝達されたデータはローカルで処理され、変更されたデータ(または差分)はデータベースに戻されます。次に重要な違いは ADO.NET がキーセットや動的カーソルのようなサーバーサイドのカーソルを何もサポートしないということです。

Of all the three database object models available to VB6 developers, only ADODB supports a programming model that is “enough” similar to ADO.NET’s disconnected approach. ADODB allows you to open a client-side recordset and use batch updates commands with optimistic lock, client-side disconnected recordsets are akin to ADO.NET’s DataTables, and server-side forwardonly-readonly (FO-RO) recordsets are similar to ADO.NET DataReader objects.

VB6 開発者対して有効な三つのデータベースオブジェクトモデルの中で、ADODB のみ ADO.NET disconnected アプローチに「匹敵」するプログラミングモデルをサポートします。ADODB によってクライアント側のレコードセットを開き、楽観的ロックによる一括更新コマンドを使用することができますし、クライアント側の disconnected レコードセットは ADO.NET のデータテーブルと同種のものであり、サーバー側の forwardonly-readonly(FO-RO)レコードセットは ADO.NET の DataReader オブジェクトと似ています。

Thanks to this similarity, converting a VB6/ADODB application to VB.NET/ADO.NET is *relatively* simple, at least according to many articles about VB6 migration. The truth is different, however, for at least two reasons. First, few commercial VB6 applications use disconnected recordsets and FO-RO recordsets exclusively, and many of them relies on server-side cursors, such as keysets. Second, the ADODB and ADO.NET object models differ in so many details – parameterized commands, store procedure invocation, field attributes, just to name a few – that the

number of necessary manual adjustments is very high and defeats the convenience of automatic migration software.

この類似性のおかげで、VB6 と ADODB のアプリケーションを VB.NET と ADO.NET に移行するのは、少なくとも VB6 移行についての多くの記事によれば、*比較的シンプル*です。しかし、少なくとも二つの理由によって、真実は異なります。まず、disconnected レコードセットや FO-RO レコードセットのみを使う商用の VB6 アプリケーションはほとんどありませんし、ほとんどのアプリケーションはキーセットのようなサーバーサイドカーソルに依存しています。次に、ADODB オブジェクトモデルと ADO.NET オブジェクトモデルは詳細において多くの相違—ほんの少し名前を挙げるだけでも、パラメタライズドコマンド、ストアドプロシジャ呼び出し、フィールド属性など—がありますし、そのために必要となる手作業による調整工数は自動移行ツールの利便性を圧倒するものです。

We at CodeArchitects decided to solve the problem of ADODB-to-ADO.NET conversion in a new, revolutionary way.

コードアーキテクツ社において私たちは新しく革新的な方法で ADODB から ADO.NET への移行の問題を解決することに決めました。

Rather than using our code converter to generate code that compensates for the many differences between the two object model – an approach that other vendors have adopted and that has proven to be largely insufficient for any real-world application – CodeArchitects has authored a .NET library named **ADOLibrary**, that is basically a hierarchy of ADO.NET objects that has exactly the same object model as and is functionally equivalent to ADODB.

弊社の移行ツールを使用して、ふたつのオブジェクトモデルの多くの相違を相殺するソースコードを生成するよりもむしろ—他のベンダーが適用していたアプローチであり、実際のアプリケーションに対しては非常に不十分であることが証明されているアプローチ—コードアーキテクツ社は **ADOLibrary** という名の.NET library を作成しました。このライブラリは基本的に ADO.NET の上位に位置づけられ、完全に同じオブジェクトモデルであり、機能的に ADODB と同等です。

For example, the ADOLibrary contains a class named ADOConnection, which exposes the same properties, methods, and events as the ADODB.Connection object. Likewise, the ADORecordset object has the same programming interface – and, more important, *the same behavior* – as the ADODB.Recordset object, and so forth.

例えば、ADOLibrary は ADOConnection というクラスを含んでおり、このクラスは ADODB.Connection オブジェクトと同じプロパティとメソッドとイベントを持っています。同様に、ADORecordset オブジェクトは ADODB.Recordset オブジェクトと同じプログラミングインターフェース—より重要なことは*動作が同じ*であることです—を持っています。

Microsoft ADODB library and CodeArchitects' ADOLibrary are very similar, except that

マイクロソフト社の ADODB ライブラリとコードアーキテクツの ADOLibrary は次の点を除いてとても似ています。

a. The name of each class is different (the ADODB.*classname* type corresponds to the CodeArchitects.ADOLibrary.ADO*classname* type)
   各クラスの名前が異なります（ADODB.*classname* は CodeArchitects.ADOLibrary.ADO*classname* に対応します)。

b. ADOLibrary is a “pure” .NET library and has no dependency on COM.
ADOLibrary は「純粋な」.NET ライブラリであり COM に依存しません。

Thanks to feature a), once you have migrated a VB6 project to VB.NET, you can use a global Find & Replace command inside Visual Studio to replace all references to ADODB into references to ADOLibrary. Unless your project uses some ADODB features that ADOLibrary doesn't support (see next sections), in the end you obtain a VB.NET project that behaves exactly like the original VB6 program except it uses ADO.NET objects exclusively.

a の特徴のお陰で、一旦 VB6 プロジェクトを VB.NET に変換すれば、ADODB に対する参照をすべて ADOLibrary に対する参照に置換するために Visual Studio のソリューション全体の検索と置換を使用することができます。ADOLibrary がサポートしないいくつかの ADODB の機能（次の章を参照）をプロジェクトが使用していない限り、ADO.NET オブジェクトのみ使用している場合を除いて、最終的には、正確に元の VB6 プログラムのように動作する VB.NET プロジェクトを作成することができます。

Notice that ***ADOLibrary doesn't depend on VB Migration Partner*** and can be successfully used also if you have converted from VB6 manually, or by using Microsoft Upgrade Wizard, or by using a conversion tool from another vendor.

ADOLibrary が VB Migration Partner に依存していないと、VB6 から手作業で移行した場合やマイクロソフト社のアップグレードウィザードや他社の移行ツールを使用して移行した場合にも問題なく使用できることは注目に値します。

Even if these two software components are independent from each other, ADOLibrary integrates perfectly with CodeArchitects' VB Migration Partner. If convert from VB6 using our software, you don't even need to perform any Find & Replace command, because all ADODB types are converted to types in ADOLibrary automatically, if you wish so.

たとえ、これら二つのソフトウェアコンポーネントがお互いに独立していても、ADOLibrary はコードアーキテクツ社の VB Migration Partner と完全に統合します。弊社のソフトウェを使って VB6 から移行する場合、そう望めば、ADODB のすべての型は ADOLibrary の型に自動的に変換されるので、検索と置換コマンドを使う必要さえありません。

A.1. Features and Limitations／機能と制限

ADOLibrary version 1.0 replicates and supports many of the most important ADODB features, including:

ADOLibrary のバージョン 1.0 は以下の ADODB の最も重要な機能の多くを複製、サポートしております。

- Forwardonly-readonly (FO-RO) recordsets
  Forwardonly-readonly(FO-RO)レコードセット
- Client-side recordsets with optimistic batch updates
  楽観的一括更新を備えたクライアントサイドのレコードセット
- Server-side cursors, including keysets, static and dynamic cursors *(Microsoft SQL Server only)*
  キーセット、静的・動的カーソルを含むサーバーサイドカーソル *（Microsoft SQL Server のみ）*

- Standalone, disconnected recordsets (e.g. Dim rs As New Recordset) with custom Fields collection
  カスタムフィールドコレクションを備えたスタンドアロン型 disconnected レコードセット（例　Dim rs As New Recordset）
- Parameterized commands
  パラメタライズドコマンド
- Stored procedure calls, with input and output parameters
  入手力パラメータを伴うストアドプロシジャ呼び出し
- Most dynamic properties, e.g. UpdateCriteria, UniqueTable and ResyncCommand
  UpdateCriteria、UniqueTable、ResyncCommand など、多くのダイナミックプロパティ

ADOLibrary preserves one of the best ADODB features, namely the ability to interact with different databases by simply changing the ConnectionString property of the Connection object. Internally, ADOLibrary can use the most appropriate ADO.NET data provider. In version 1.0 the following providers are supported

ADOLibrary は最高の ADODB の特徴の一つを引き継いでいます。すなわち、Connection オブジェクトの ConnectionString プロパティを変更するだけで異なるデータベースと相互に接続する機能です。内部で、ADOLibrary はもっとも適切な ADO.NET データプロバイダを使用します。バージョン 1.0 では次のプロバイダがサポートされています。

- System.Data.OleDb
- System.Data.SqlClient

Support for Oracle and ODBC providers will be added in future releases. (Current version can access Oracle databases by using the OleDb provider, though.)

Oracle と ODBC providers に対するサポートは将来のリリースで追加される予定です。（とはいえ、現在のバージョンは OleDb provider を使って Oracle のデータベースに接続することができます。）

Interestingly, many objects in the ADOLibrary expose additional properties and methods that aren't found in ADODB and that allow you to leverage the best features of ADO.NET. For example, the ADORecordset class – in addition to all the members "inherited" from the ADODB.Recordset class – exposes the **DataTable** property, which returns the System.Data.DataTable object containing all the rows read from the database and transferred to the client-side cursor. Thanks to this property you can then read and write such rows either by using the MoveNext method (the ADODB way) or by directly accessing the DataTable object, as in following example:

興味深いことに、ADOLibrary の多くのオブジェクトに ADODB にはないプロパティとメソッドが追加されており、ADO.NET の機能を最大限に引き出すことができます。例えば、ADORecordset クラスはデータベースから読んだ行を含む System.Data.DataTable オブジェクトを返す **DataTable** プロパティを―ADODB.Recordset クラスから「継承」されたメンバーとは別に―持っています。このプロパティのお陰で、MoveNext メソッドの使用または DataTable オブジェクトを直接アクセスのどちらかによって以下の例のように行の読み書きができます。

```
Sub ReadRows(ByVal cn As ADOConnection, ByVal sql As String)
    Dim rs As New ADORecordset
    rs.CursorLocation = ADOCursorLocationEnum.adUseClient
```

```
        rs.Open(sql, cn, ADOCursorTypeEnum.adOpenStatic, _
                ADOLockTypeEnum.adLockBatchOptimistic)

        ' access individual rows using ADODB-like approach
        Do Until rs.EOF
            Debug.WriteLine(rs(0).Value)
            '...
            rs.MoveNext()
        Loop

        ' access all rows together via the DataTable property
        For Each dr As DataRow in rs.DataTable.Rows
            Debug.WriteLine(dr(0))
            '...
        Next

        ' you can also use the DataTable property to bind the recordset to a
        ' standard DataGridView control, or any other bindable .NET control
        DataGridView1.DataSource = rs.DataTable
        rs.BindingManager = Me.BindingManager
    End Sub
```

Limitations／機能制限

ADOLibrary doesn't support all ADODB features. Here is the list of the main features that aren't supported or are supported only partially:

ADOLibrary は ADODB の機能すべてをサポートしているわけではありません。以下はサポートされていない機能と部分的にサポートしている機能の一覧です。

- Hierarchical recordsets aren't supported.
  階層型レコードセットはサポートされません。
- Server-side cursors (other than FO-RO cursors) are supported only for Microsoft SQL Server databases.
  （FO-RO カーソル以外の）サーバーサイドカーソルは Microsoft SQL Server のみサポートされます。
- Multiple, semicolon-delimited SQL statements are supported in **Recordset.Open** methods, but not in the**Execute** method of the Connection and Command objects.
  複数のセミコロンで区切られた SQL 文は **Recordset.Open** メソッドでサポートされますが、Connection オブジェクトと Command オブジェクトの **Execute** メソッドではサポートされません。
- Asynchronous execution is allowed for the **Open** method of Connection and Recordset objects, but not for the **Execute** method of the Connection and Command objects.
  非同期実行は Connection オブジェクトと Recordset オブジェクトの **Open** メソッドに対して許可され

ていますが、Connection オブジェクトと Command オブジェクトの **Execute** メソッドには許可されていません。

- The **UpdateBatch** method of the Recorset object requires that the SQL source statement includes the key fields of all involved tables and doesn't work with old-syntax JOIN statements, with nested SELECT statements, and with derivate tables.
  Recorset オブジェクトの **UpdateBatch** メソッドは SQL 文が内包するテーブルすべてのキー列を含んでいる必要があり、古い構文の JOIN ステートメントやネストした SELECT 文や派生テーブルを使用していると動作しません。
- The SELECT statement used to open server-side keyset and dynamic cursors must include at least one non-nullable key column.
  サーバーサイドのキーセットや動的カーソルを呼び出すために使用される SELECT 文は少なくともひとつの null を許可しないキー列に含まれていなければなりません。
- The **Move** method doesn't work and raises an exception with serverside dynamic cursors.
  サーバーサイドの動的カーソルを伴う場合、**Move** メソッドは動作せず、例外が発生します。
- The **Index**, **Seek**, **MarshalOption**, **StayInSync** members of the Recordset class aren't implemented and are marked as obsolete.
  Recordset クラスの **Index**、**Seek**、**MarshalOption**、**StayInSync** は実装されておらず、既に使用されていないメンバーとして認識されます。
- The **PageSize**, **Clone**, and **Find** members of the Recordset class aren't implemented for keysets and other server-side cursors.
  Recordset クラスの **PageSize**、**Clone**、**Find** はキーセットや他のサーバーサイドカーソル用には実装されていません。
- The **CommandStream**, **Dialect**, **NamedParameters**, and **Prepared** members of the Command class aren't implemented and are marked as obsolete.
  Command クラスの **CommandStream**、**Dialect**、**NamedParameters**、**Prepared** は実装されておらず、既に使用されていないメンバーとして認識されます。
- The **ReadText**, **WriteText**, and **SkipLines** methods of the Stream class aren't implemented and are marked as obsolete.
  Stream クラスの **ReadText**、**WriteText**、**SkipLines** は実装されておらず、既に使用されていないメンバーとして認識されます。
- A few dynamic properties of the Connection or Recordset objects aren't supported.
  Connection オブジェクトと Recordset オブジェクトの２、３のダイナミックプロパティはサポートされません。
- The ADODB.Record class isn't supported.
  ADODB.Record クラスはサポートされません。

More details about these limitations are provided later in this document.
これらの詳細についての詳細は後述します。

A.2. Installing and Using ADOLibrary／ADOLibrary のインストールと使用方法

You can download ADOLibrary from VB Migration Partner's Download page, which you can reach from the Help menu. (If you purchased ADOLibrary separatedly, you will receive the URL to this page via email.)

ADOLibrary は VB Migration Partner のダウンロードページからダウンロードすることができます。また、ダウンロードページにはヘルプメニューから移動することもできます。(ADOLibrary だけを購入した場合、URL が email で送付されます。)

If you are converting a VB6 application using VB Migration Partner, you can enable ADODB-to-ADO.NET conversion using the ADOLibrary in either of the following ways:

VB Migration Partner を使用して VB6 アプリケーションを移行している場合、次のいずれかの方法で ADOLibrary を使用した ADODB から ADO.NET への変換を有効にすることができます。

A. You copy the CodeArchitects.ADOLibrary.dll file into VB Migration Partner's setup folder.
   VB Migration Partner のセットアップフォルダに CodeArchitects.ADOLibrary.dll をコピーします。
B. You use an **AddLibraryPath** pragma to point to the folder where you have stored the CodeArchitects.ADOLibrary.dll file, as in:
   次のように、CodeArchitects.ADOLibrary.dll ファイルを保存したフォルダを指定する **AddLibraryPath** プラグマを使います。

```
'## AddLibraryPath "c:¥adolib"
```

Notice that the AddLibraryPath pragma can be only inserted in a *.pragmas file.
AddLibraryPath プラグマは*.pragmas にのみ挿入することができることに注意してください。

C. You use an **ImportTypeLib** pragma to alias the ADODB type library with the CodeArchitects.ADOLibrary.dll assembly:
   CodeArchitects.ADOLibrary.dll アセンブリとともに ADODB タイプライブラリに別名をつける **ImportTypeLib** プラグマを使います。

```
'## ImportTypeLib msado27.tlb "@c:¥adolib¥CodeArchitects.AdoLibrary.dll"
```

Notice that the ImportTypeLib pragma can be only inserted in a *.pragmas file.
ImportTypeLib プラグマは*.pragmas ファイルにのみ挿入することができることに注意してください。

If you apply strategy A), then *any* VB6 project that references the ADODB type library will be converted to a VB.NET project that references the ADOLibrary instead. Strategies B) and C) allows you to be more granular, therefore they are the approaches we recommend if you want to preserve ADODB in some of your converted projects.

手順 A が行える場合、ADODB タイプライブラリを参照する VB6 プロジェクトであれば *何でも*、代わりに ADOLibrary を参照する VB.NET プロジェクトに変換されます。手順 B と C の場合、より粗くなりまので、移行されるプロジェクトのいくつかで ADODB を使用したい場合にこれらの手順をお勧めしています。

If you are not using VB Migration Partner – for example, you have migrated from VB6 either manually, or using Microsoft Upgrade Wizard (included in Visual Studio 2005 or 2008), or using a converted from another vendor – you can still adopt ADOLibrary. In this case, the manual procedure is as follows:

VB Migration Partner を使用していない場合でも、—例えば、手作業で VB6 から移行された場合や、(Visual Studio 2005 か 2008 に含まれる)Microsoft Upgrade Wizard を使用された場合や、他社の移行ツールをご使用になられた場合—ADOLibrary を使用することができます。この場合の手順は次のようになります。

1. Load the converted VB.NET project inside Visual Studio
   Visual Studio に変換する VB.NET プロジェクトを読み込んでください
2. Open the References tab of the My Project designer
   My Project プロパティの参照タブを開いてください
3. Drop the reference to the ADODB.dll file *(this will cause many compilation errors to appear)*
   参照する ADODB.dll ファイルをドロップしてください。*(これによって多くのコンパイルエラーが発生します)*
4. Add a reference to the CodeArchitects.ADOLibrary.dll file
   CodeArchitects.ADOLibrary.dll ファイルへの参照を追加してください
5. Add "CodeArchitects.ADOLibrary" as a project-level imported namespace
   プロジェクトレベルのインポートされた名前空間に CodeArchitects.ADOLibrary を追加してください
6. Using Visual Studio's Find and Replace command, replace all occurrences of "ADODB." (dot included) into "ADO" *(this step should solve all compilation errors caused at point 3)*
   Visual Studio の検索と置換コマンドを使って、見つかった「ADODB.」(.も含みます) をすべて「ADO」に置換してください。*(この段階で手順 3 で発生したコンパイルエラーはすべてなくなります)*

The effect of the last action is to replace class names such as "ADODB.Recorset" into "ADORecordset", so that all statements that were using ADODB objects will now use the corresponding object in ADOLibrary.

最後の手順の効果は、クラス名を「ADODB.Recorset」から「ADORecordset」のように置換することです。これによって、ADODB オブジェクトを使用するすべてのステートメントが ADOLibrary の対応するオブジェクトを使用するようになります。

A.3. ADOLibrary Reference／ADOLibrary リファレンス

This section describes each class in the library, with details on all the properties, methods, and events that might work differently from "classic" ADODB or that were added to the library to compensate for the differences between the two environments.

この章では、「古い」ADODB とは異なる動作をしたり、両者の相違を埋め合わせるためにライブラリ追加された各クラスのプロパティ、メソッド、そしてイベントについて解説します。

The ADOConfig class／ADOConfig クラス

ADOLibrary exposes a special ADOConfig class, which allows you to change the behavior of objects in the library and take full advantage of ADO.NET versatility, with just minor edits in source code. ADOConfig exposes static (Shared in VB.NET) properties exclusively.

ADOLibrary は特別な ADOConfig クラスを持っており、ライブラリのオブジェクトの動作を変更することができ、ソースコードに小さな編集を加えるだけで ADO.NET の多様性を完全に利用することができます。ADOConfig は静的プロパティ（VB.NET では Shared）のみ持ちます。

BatchUpdatesSetAllValues property／BatchUpdatesSetAllValues プロパティ

If this property is set to False (the default value) then UPDATE statements generated by UpdateBatch methods assign only the columns that have been modified. For example, if you have read Name, Company, and City fields from a given table and you later modify only the Name and Company fields, the UPDATE statement generated for that row when you issue the UpdateBatch method will assign only these two fields and won't modify the value of the City field currently stored in the database.

このプロパティに Flase をセットすると UpdateBatch メソッドによって生成された UPDATE 文は修正された列だけを対象とします。例えば、テーブルから Name、Company、City フィールドを読み込んだ後で、Name と Company フィールドだけを修正した場合、UpdateBatch メソッドを使用しているときにその行に対して生成される UPDATE 文はこれらの二つのフィールドのみを対象とし、現在データベースに保存されている City フィールドの値は修正しません。

Even if this behavior perfectly mimics what ADODB does, many developers consider it a design flaw of ADODB and would prefer the UpdateBatch method to assign all the fields in the row, regardless of whether they were modified by the client application. You can achieve this safer behavior by simply assigning True to the BatchUpdatesSetAllValues property:

たとえこの動作が ADODB の動作を完全に複製したとしても、ほとんどの開発者はそれを ADODB のデザイン上の欠点だと考え、それらがクライアントアプリケーションによって修正されるかどうかに関わらず、行のフィールドすべてを対象とする UpdateBatch メソッドを好むでしょう。利用者は BatchUpdatesSetAllValues プロパティに True を設定するだけでより安全に動作させることができます。

```
ADOConfig.BatchUpdatesSetAllValues = True
```

Notice that the ADORecordset class also exposes the BatchUpdatesSetAllValues property, therefore you can change the behavior for each individual Recordset.

ADORecordset クラスは BatchUpdatesSetAllValues プロパティも持っていますので、個々のレコードセットに対して動作を変更することができます。

EnforceConstraintsOnLoad property／EnforceConstraintsOnLoad プロパティ

When used to create a client-side static cursor, the Open method of the ADORecordset reads rows from the database and loads them into a private, temporary DataSet object whose EnforceConstraints property is set to False, and only later data is moved into a DataTable. This step is necessary to perfectly replicate the ADODB

behavior, which never raises an error if an incoming row doesn't match all required constraints. (For example, ADODB doesn't raise an error if the incoming row contains a NULL value for a non-nullable column.)

クライアントサイドの静的カーソルが使用される場合、ADORecordset の Open メソッドはデータベースから行を読み、EnforceConstraints プロパティが False にセットされた private で temporary な DataSet オブジェクトにそれらを読み込んだ後にのみ、DataTable にデータを移します。このステップは ADODB の、取り込まれる行が要求される全制約に該当しない場合、エラーを発生させないという動作を完全に複製するために不可欠です。(例えば、取り込まれる行が非 NULL 列に NULL を含む場合、ADODB はエラーを発生させません。)

On the other hand, loading data into a temporary DataSet slightly degrades performances. If you are sure that incoming data doesn't violate any database constraint you can skip this intermediate step by setting the EnforceConstraintsOnLoad property to True:

一方、temporary の DataSet にデータを読み込むことはわずかにパフォーマンスを低下させます。取り込みデータがデータベースの制約に違反することが確実な場合、EnforceConstraintsOnLoad プロパティを True にセットして中間ステップをスキップすることができます。

```
ADOConfig.EnforceConstraintsOnLoad = True
```

Notice that the ADORecordset class also exposes the EnforceConstraintsOnLoad property, therefore you can change the behavior for each individual recordset.

ADORecordset クラスが EnforceConstraintsOnLoad プロパティも持っているため、個々のレコードセットに対する動作を変更できることに注意してください。

IgnoreDataSchema property／IgnoreDataSchema プロパティ

If this property is set to False (the default value) and you open a Recordset in batch optimistic mode, ADOLibrary queries the data source and retrieves schema information about individual fields. This information is used to correctly implement a few properties of the Field object, namely DefinedSize, NumericScale, Precision and the Properties collection. If you are sure that your code doesn't rely on this metadata information, you can speed up Recordset.Open methods by setting this property to True.

このプロパティに(初期値)False をセットし、一括楽観モードでレコードセットをオープンする場合、ADOLibrary はデータソースに照会し、個々のフィールドに関するスキーマ情報を取り出します。この情報は Field オブジェクトの2、3のプロパティを正しく実装するために使用されます。すなわち、DefinedSize、NumericScale、Precision と Properties コレクションです。コードがこのメタデータ情報に依存しないことが確実である場合、このプロパティに True をセットして Recordset.Open メソッドの速度を向上させることができます。

```
ADOConfig.IgnoreDataSchema  = True
```

Notice that this assignment affects all instances of the ADORecordset class. You can be more granular by leaving this global property set to False and assign True to the IgnoreDataSchema property of individual ADOConnection, ADORecordset, and ADOCommand objects.

この設定が ADORecordset クラスの全インスタンスに影響を及ぼすことに注意してください。利用者はこのグローバルプロパティに False をセットしたり、ADOConnection 、ADORecordset 、ADOCommand の各オブジェクトの IgnoreDataSchema プロパティに True をセットすることでより粗くすることができます。

LibraryKind property／LibraryKind プロパティ

By default, ADOLibrary internally uses the OleDb ADO.NET data provider. If you know that the VB.NET application will only access a specific database – for example, Microsoft SQL Server – you can improve performance by assigning a new value to this property:

デフォルトでは、ADOLibrary は OleDb 用 ADO.NET データプロバイダを内部的に使用します。VB.NET アプリケーションが特定のデータベースにのみ接続することがわかっている場合、—例えば、Microsoft SQL Server—このプロパティに新しい値を割り当てて性能を向上させることができます。

```
ADOConfig.LibraryKind = ADOLibraryKind.SqlClient
```

Valid values for the LibraryKind property are: OleDb, SqlClient, Odbc, OracleClient. However, version 1.0 of ADOLibrary only supports the OleDb and SqlClient values. Any other value throws an exception (if ThrowOnUnsupportedMembers is True) or are ignored.

LibraryKind プロパティにに対して有効な値は、OleDb、SqlClient、Odbc、OracleClient です。 ただし、ADOLibrary のバージョン 1.0 は OleDb と SqlClient のみサポートします。その他の値は例外を発生させるか（ThrowOnUnsupportedMembers が True なら）無視されます。

SynchronizingObject property／SynchronizingObject プロパティ

The ADOConnection and ADORecordset objects can raise asynchronous events. By default, these events run in thread other than the main thread of the VB.NET application. This detail can cause problems and unexpected crashes if the code in the event handler accesses one or more user interface elements, such as a form or a control. You can avoid this problem by assigning a form or a control reference to the ADOConfig.SynchronizingObject property. Any form or control will do:

ADOConnection オブジェクトと ADORecordset オブジェクトは非同期イベントを発生させることができます。デフォルトでは、これらのイベントは VB.NET アプリケーションのメインスレッド以外のスレッドで実行されます。イベントハンドラーのコードが、フォームやコントロールのような、一つ以上のインターフェース要素にアクセスする場合、細部において問題が起こったり、予期しないクラッシュが発生する可能性があります。利用者は、ADOConfig.SynchronizingObject プロパティを参照するフォームやコントロールを割り当てることで、この問題を回避することができます。どのようなフォームやコントロールでも次のように記述することができます。

```
' if inside a form class, "Me" is a reference to the current form
ADOConfig.SynchronizingObject = Me
```

Assigning the ADOConfig.SynchronizingObject property affects all the asynchronous events of all the objects instantiated from the ADOLibrary. If only a few event handlers access UI elements you're better off assigning the SynchronizingObject property of individual ADOConnection and ADORecordset instances.

ADOConfig.SynchronizingObject プロパティは ADOLibrary からインスタンス化したオブジェクトの非同期イベントすべてに影響を与えます。ほんの2，3個のイベントハンドラーが UI 要素にアクセスする場合は、ADOConnection と ADORecordset の各インスタンスの SynchronizingObject プロパティを設定した方がいいでしょう。

ThrowOnUnsupportedMembers property／ThrowOnUnsupportedMembers プロパティ

The default behavior of the ADOLibrary is to ignore assignments to properties that aren't supported – for example the Index property of the ADORecordset class – as well as calls to unsupported methods. This behavior is OK during the early stages of the migration, but can be problematic when running more rigorous tests. By setting the ThrowOnUnsupportedMembers property to True you cause any invocation of unsupported members to cause a runtime exception:

ADOLibrary のデフォルト動作は、サポートされないメソッドに対する呼び出しのように、サポートされないプロパティ設定—例えば、ADORecordset クラスの Index プロパティ—を無視します。この動作は移行の初期段階では問題ありませんが、より厳密なテストを行っているときは問題になる可能性があります。ThrowOnUnsupportedMembers プロパティを True に設定することで、ランタイムエラーを発生させるサポートされないメンバーを呼び出しを発生させます。

```
ADOConfig.ThrowOnUnsupportedMembers = True
```

Note for VB Migration Partner users: this property is similar to and has the same effect of the VBConfig.ThrowOnUnsupportedMembers property.

VB Migration Partner の利用者にとって、このプロパティは VBConfig.ThrowOnUnsupportedMembers プロパティと似ており、同じ効果を持つものであることを覚えてください。

The ADOConnection class／ADOConnection クラス

The ADOConnection class is similar and behaves like the ADODB.Connection class, except for the following details.

ADOConnection クラスは、以下の特徴以外については ADODB.Connection クラスに似ており、同じように動作します。

Close method／Close メソッド

Under ADODB, a Connection object is automatically closed when the last reference to it is set to Nothing or just go out of scope. Because of how .NET manages memory, ADOLibrary objects aren't automatically closed when they go out of scope. Instead, it is essential that you explicitly close an ADOConnection object just before setting it to Nothing or letting it go out of scope.

ADODB では、オブジェクトに対する最後の参照に Nothing がセットされるかスコープの外になったとき、Connection オブジェクトは自動的に閉じられます。.NET ではメモリー管理方法が原因で、スコープの外になったとき、ADOLibrary オブジェクトは自動的に閉じられません。その代わり、Nothing かスコープの外になる直前に ADOConnection オブジェクトを明示的に閉じることが必須となっています。

Failing to do so may cause problems in the migrated .NET application, because the connection is closed only sometime later. For example, if the connection was opened in exclusive mode, no other application will be able to re-open it, until the .NET Framework starts a garbage collection some seconds or even minutes later.

コネクションが時々遅く閉じられるため、そうすることに失敗することで、変換された.NET アプリケーションに問題が起こる可能性があります。例えば、コネクションが排他モードで開かれていた場合、.NET Framework が数秒か数分後にガベージコレクションを始めるまでは、他のアプリケーションはそれを再オープンすることができなくなります。

Also, according to .NET programming guidelines, you should never invoke the Close method of an ADOConnection object from inside the Finalize method of a .NET class, or (equivalently) from the Class_Terminate event of the VB6 application being migrated. This practice is to be avoided because the inner ADO.NET object might have been already finalized or disposed of when the .NET Finalize method is invoked.

また、.NET プログラミングガイドラインによると、.NET クラスの Finalize メソッドの内部、または、変換される VB6 アプリケーションの Class_Terminate イベントから ADOConnection オブジェクトの Close メソッドを絶対に呼び出すべきではありません。.NET の Finalize メソッドが呼び出されたとき、ADO.NET オブジェクトが内部的にすでにファイナライズされているかディスポーズされているかもしれないため、このやり方は避けるべきです。

Connection property (.NET only)／Connection プロパティ(.NET のみ)

This readonly property returns the DbConnection object that is used for the current ADOConnection instance. This property allows you to mix ADODB-like statements and ADO.NET statements.

この readonly プロパティは現在の ADOConnection インスタンスに対して使用される DbConnection オブジェクトを返します。このプロパティによって ADODB のようなステートメントと ADO.NET ステートメントを混在させることができます。

```
Sub ExecuteUnderTransaction(ByVal cn As ADOConnection)
    Dim trans As DbTransaction = cn.Connection.BeginTransaction()
    ' ... more statements here
    ここにはもっと多くのステートメントがあります

    trans.Commit()
End Sub
```

The ADOConnection class also exposes other readonly properties named **OleDbConnection**, **SqlConnection**, and **OdbcConnection**, which return a more specific connection object, or Nothing if the ADO.NET data provider used internally doesn't match the property type.

ADOConnection クラスは他に **OleDbConnection** や **SqlConnection** や **OdbcConnection** という読み取り専用プロパティも持っています。それらは特定のコネクションオブジェクトを返すか、使用されている ADO.NET データプロバイダがプロパティタイプと内部的に一致しない場合は Nothing を返します。

DefaultDatabase property／DefaultDatabase プロパティ

Unlike the ADODB property, this property doesn't reflect changes in the default database that were enforced by means of a direct **USE dbname** SQL statement.

ADODB プロパティとは異なるこのプロパティは、直接的な **USE dbname** を使用する SQL 文によって強制されたデフォルトデータベースの変更を反映しません。

Execute method／Execute メソッド

This method works as in ADODB, except it throws an exception if multiple, semicolon-delimited SQL statements are passed as an argument.

このメソッドは、複数のセミコロンで区切られた SQL 文が引数として渡された場合に例外を発生させる以外は、ADODB と同様に動作します。

IgnoreDataSchema property (.NET only)／IgnoreDataSchema プロパティ(.NET のみ)

If this property is set to False (the default value) and you open a Recordset in batch optimistic mode, ADOLibrary queries the data source and retrieves schema information about individual fields. This information is used to correctly implement a few members of the Field object, namely DefinedSize, NumericScale, Precision, and Properties. If you are sure that your code doesn't rely on this metadata information, you can speed up Recordset.Open methods by setting this property to True.

このプロパティに(初期値)False をセットし、一括楽観モードでレコードセットをオープンする場合、ADOLibrary はデータソースに照会し、個々のフィールドに関するスキーマ情報を取り出します。この情報は Field オブジェクトの2、3のメンバーを正しく実装するために使用されます。すなわち、DefinedSize、NumericScale、Precision と Properties です。コードがこのメタデータ情報に依存しないことが確実である場合、このプロパティに True をセットして Recordset.Open メソッドの速度を向上させることができます。

```
Sub ReadWithNoSchema(ByVal cn As ADOConnection, ByVal rs As ADORecordset)
    cn.IgnoreDataSchema = True
    rs.Open "Select * From Orders", cn
    ' ...
End Sub
```

The default value of this property when the ADOConnection object is instantiated is equal to ADOConfig.IgnoreDataSchema.

ADOConnection オブジェクトがインスタンス化されたとき、このプロパティの初期値は ADOConfig.IgnoreDataSchema に等しい。

LibraryKind property (.NET only)／LibraryKind プロパティ(.NET のみ)

By default, ADOLibrary internally uses the OleDb ADO.NET data provider. If you know that a given ADOConnection object will only access a specific database – for example, Microsoft SQL Server – you can improve performance by assigning a new value to this property:

デフォルトでは、ADOLibrary は OleDb 用 ADO.NET データプロバイダを内部的に使用します。ADOConnection オブジェクトが特定のデータベースにのみ接続することがわかっている場合、—例えば、Microsoft SQL Server—このプロパティに新しい値を割り当てて性能を向上させることができます

```
Dim cn As New ADOConnection
cn.LibraryKind = ADOLibraryKind.SqlClient
```

Valid values for the LibraryKind property are: OleDb, SqlClient, Odbc, OracleClient. However, version 1.0 of ADOLibrary only supports the OleDb and SqlClient values. Any other value throws an exception (if ThrowOnUnsupportedMembers is True) or are ignored. If this property is left unassigned, it is set equal to the value of ADOConfig.LibraryKind property.

LibraryKind プロパティにに対して有効な値は、OleDb、SqlClient、Odbc、OracleClient です。 ただし、ADOLibrary のバージョン 1.0 は OleDb と SqlClient のみサポートします。その他の値は例外を発生させるか（ThrowOnUnsupportedMembers が True なら）無視されます。このプロパティが設定されない場合、ADOConfig.LibraryKind プロパティの値に等しくなります。

Open method／Open メソッド

This method works exactly as in ADODB, except that it issues the following SQL command immediately after opening a SQL Server connection:

このメソッドは、SQL Server 接続が開始された直後に以下の SQL コマンドが発行される場合を除いて、ADODB と同様に動作します。

```
SET LANGUAGE 'English'
```

This step is necessary because ADOLibrary must interpret error messages coming from the database, in order to raise the corresponding ADODB error message.

対応する ADODB エラーメッセージを呼び出すために、ADOLibrary がデータベースから来るエラーメッセージを解析しなければならないため、このステップは必須です。

OpenSchema method／OpenSchema メソッド

This method accepts only a subset of the enumerated values you can pass to the corresponding ADODB method, namely: adSchemaCatalogs, adSchemaTables, adSchemaColumns, adSchemaProcedures, adSchemaProcedureParameters, adSchemaIndexes. However, an overload that takes the ADO.NET collection name is also provided:

このメソッドは、対応する ADODB メソッドに渡すことができる数値付きの値のサブセットのみ受け取ることができます。すなわち、adSchemaCatalogs、adSchemaTables、adSchemaColumns、adSchemaProcedures、adSchemaProcedureParameters、adSchemaIndexes です。ただし、ADO.NET コレクション名を取るオーバーロードも受け取ることができます。

```
Sub Test(ByVal cn As ADOConnection)
    ' the following two statements are equivalent
```

```
        Dim rs As ADORecordset
        rs = cn.OpenSchema(ADOSchemaEnum.adSchemaCatalogs)
        ' the following statement uses the ADO.NET syntax
        rs = cn.OpenSchema("Catalogs")
    End Sub
```

SynchronizingObject property (.NET)／SynchronizingObject プロパティ(.NET)

The ADOConnection object can raise asynchronous events. By default, these events run in thread other than the main thread of the VB.NET application. This detail can cause problems and unexpected crashes if the code in the event handler accesses one or more user interface elements, such as a form or a control. You can avoid this problem by assigning a form or a control reference to the SynchronizingObject property. Any form or control will do:

ADOConnection オブジェクトは非同期イベントを発生させることができます。デフォルトでは、これらのイベントは VB.NET アプリケーションのメインスレッド以外のスレッドで実行されます。イベントハンドラーのコードが、フォームやコントロールのような、一つ以上のユーザーインターフェース要素にアクセスする場合、細部において問題が起こったり、予期しないクラッシュが発生する可能性があります。利用者は、SynchronizingObject プロパティを参照するフォームやコントロールを割り当てることで、この問題を回避することができます。どのようなフォームやコントロールでも次のように記述することができます。

```
' if inside a form class, "Me" is a reference to the current form
Dim cn As New ADOConnection
cn.SynchronizingObject = Me
```

Version property／Version プロパティ

This property always returns the value “2.8”.

このプロパティは常に 2.8 を返します。

The ADORecordset class／ADORecordset クラス

The ADORecordset class is similar and behaves like the ADODB.Recordset class, except for the following details.

ADORecordset クラスは、以下にあげる機能以外は ADODB.Recordset クラスと似ており、ADODB.Recordset クラスのように動作する。

BatchUpdatesSetAllValues property (.NET only)／BatchUpdatesSetAllValues プロパティ(.NET のみ)

If this property is set to False (the default value) then UPDATE statements generated by UpdateBatch methods assign only the columns that have been modified. For example, if you have read Name, Company, and City fields from a given table and you later modify only the Name and Company fields, the UPDATE statement generated for that row when you issue the UpdateBatch method will assign only these two fields and won’t modify the value of the City field currently stored in the database.

このプロパティに Flase をセットすると UpdateBatch メソッドによって生成された UPDATE 文は修正された列だけを対象とします。例えば、テーブルから Name、Company、City フィールドを読み込んだ後で、Name と Company フィー

ルドだけを修正した場合、UpdateBatch メソッドを使用しているときにその行に対して生成される UPDATE 文はこれらの二つのフィールドのみを対象とし、現在データベースに保存されている City フィールドの値は修正しません。

Even if the abovementioned behavior perfectly mimics what ADODB does, many developers consider it a design flaw of ADODB and would prefer the UpdateBatch method to assign all the fields in the row, regardless of whether they were modified by the client application. You can achieve this safer behavior by simply assigning True to the BatchUpdatesSetAllValues property:

たとえ上述の動作が ADODB の動作を完全に複製したとしても、ほとんどの開発者はそれを ADODB のデザイン上の欠点だと考え、それらがクライアントアプリケーションによって修正されるかどうかに関わらず、行のフィールドすべてを対象とする UpdateBatch メソッドを好むでしょう。利用者は BatchUpdatesSetAllValues プロパティに True を設定するだけでより安全に動作させることができます。

```
Dim rs As New ADORecordset
rs.BatchUpdatesSetAllValues = True
```

When an ADORecordset is created, this property is set equal to ADOConfig.BatchUpdatesSetAllValues property.

ADORecordset が作成されると、このプロパティは ADOConfig.BatchUpdatesSetAllValues プロパティと等しくなります。

BindingManager property (.NET)／BindingManager プロパティ(.NET)

When manually binding an ADORecordset object to a .NET control – such as a DataGridView control – you should also assign this property, as shown in following example:

.NET コントロールに対して—DataGridView コントロールのような—ADORecordset オブジェクトを手作業でバインドする場合、以下の例のように、このプロパティも割り当てるべきです。

```
DataGridView1.DataSource = rs.DataTable
rs.BindingManager = Me.BindingManager
```

Bookmark property／Bookmark プロパティ

This property works as in ADODB, except that you cannot assign it to move to a record that has been deleted. Under ADODB this operation is legal, whereas it throws an exception with the ADOLibrary.

このプロパティは、削除されたレコードを移動させるために割り当てることができなということ以外は、ADODB と同じように動作します。ADODB においてこの操作は有効ですが、ADOLibrary では例外が発生します。

CacheSize property／CacheSize プロパティ

This property preserves any value you assign to it, but is otherwise ignored because ADOLibrary doesn't maintain a row cache.

このプロパティは割り当てた値を保持しますが、さもなければ ADOLibrary は行キャッシュを維持しないため無視されます。

Close method／Close メソッド

Under ADODB, a Recordset object is automatically closed when the last reference to it is set to Nothing or just go out of scope. Because of how .NET manages memory, ADOLibrary objects aren't automatically closed when they go out of scope. Instead, it is essential that you explicitly close an ADORecordset object just before setting it to Nothing or letting it go out of scope.

ADODB では、オブジェクトに対する最後の参照に Nothing がセットされるかスコープの外になったとき、Recordset オブジェクトは自動的に閉じられます。.NET ではメモリー管理方法が原因で、スコープの外になったとき、ADOLibrary オブジェクトは自動的に閉じられません。その代わり、Nothing かスコープの外になる直前に ADORecordset オブジェクトを明示的に閉じることが必須となっています。

Failing to do so may cause problems in the migrated .NET application, because the connection is closed only sometime later. For example, if the recordset had opened a SQL Server server-side cursor, the current application cannot open another server-side cursor until the .NET Framwork starts a garbage collection (unless you are using the MARS feature that comes with Microsoft SQL Server 2005 and later versions).

例えば、レコードセットが SQL Server のサーバーサイドカーソルをオープンした場合、現在のアプリケーションは、(Microsoft SQL Server 2005 以降で提供される MARS(Multiple Active Result Set)という機能を使用しない限り).NET Framwork がガベージコレクションを開始するまでは他のサーバーサイドカーソルをオープンすることができません。

Also, according to .NET programming guidelines, you should never invoke the Close method of an ADORecordset object from inside the Finalize method of a .NET class, or (equivalently) from the Class_Terminate event of the VB6 application being migrated. This practice is to be avoided because the inner ADO.NET object might have been already finalized or disposed of when the .NET Finalize method is invoked.

また、.NET プログラミングガイドラインによると、.NET クラスの Finalize メソッドの内部、または、変換される VB6 アプリケーションの Class_Terminate イベントから ADORecordset オブジェクトの Close メソッドを絶対に呼び出すべきではありません。.NET の Finalize メソッドが呼び出されたとき、ADO.NET オブジェクトが内部的にすでにファイナライズされているかディスポーズされているかもしれないため、このやり方は避けるべきです。

CommandBehavior property (.NET only) ／CommandBehavior プロパティ(.NET のみ)

The value of this property is used internally and is assigned to the CommandBehavior property of the inner ADO.NET Command object used to open the ADORecordset. For example, if you know that the rowset being read contains only a single row, you can slightly optimize execution as follows

このプロパティの値は内部的に使用され、ADORecordset をオープンするために使用される ADO.NET 内部の Command オブジェクトの CommandBehavior プロパティに対して割り当てられます。例えば、読み込まれる行セットが 1 行だけであることが分かっていれば、次のように若干楽観的な実行が可能となります。

```
Sub TestSingleRow(ByVal rs As ADORecordset, ByVal cn As ADOConnection)
    rs.CommandBehavior = CommandBehavior.SingleRow
    ' notice that following statement can return max row at the maximum
```

次のステートメントが最大での最大行を返すことができることに注意してください

```
        rs.Open("SELECT * FROM Orders WHERE OrderID=1", cn)
        ' ...
    End Sub
```

This property is ignored if assigned to an opened ADORecordset.

このプロパティは、オープンされた ADORecordset に対して割り当てられる場合、無視されます。

Copy method (.NET)／Copy メソッド(.NET)

This method returns a distinct copy of the current ADORecordset and is equivalent to saving and reloading an ADORecordset object to/from a file. The Copy method takes two parameters, a MarshalOptionEnum value and an optional filter string:

このメソッドは現在の ADORecordset のコピーを返し、ファイルへの ADORecordset の保存またはファイルからの ADORecordset 読み出しと同等です。この Copy メソッドは、MarshalOptionEnum 値と任意のフィルター文字列という二つのパラメータを受け取ります。

```
    Sub TestCopy(ByVal rs As ADORecordset)
        Dim copyRs As ADORecordset
        Dim filter As String = "City = 'Boston'"
        copyRs = rs.Copy(ADOMarshalOptionEnum.adMarshalModifiedOnly, filter)
        '...
    End Sub
```

If the second parameter is omitted, then the current value of the Filter property is used. The Copy method is only valid for client-side ADORecordset objects.

第二引数が省略される場合、現在の Filter プロパティの値が使用されます。Copy メソッドはクライアントサイドの ADORecordset オブジェクトに対してのみ有効です。

CursorHandle property (.NET only)／CursorHandle プロパティ(.NET のみ)

This readonly property returns the handle of the server-side cursor used internally when the ADORecordset is used to open a server-side SQL Server cursor. In advanced scenarios you can use this cursor handle to send direct commands to SQL Server.

この読み取り専用プロパティは、ADORecordset がサーバーサイドの SQL Server カーソルをオープンするために使用される場合に、内部的に使用されるサーバーサイドカーソルのハンドルを返します。高度なやり方として SQL Server に直接コマンドを送るために、このカーソルハンドルを使用することができます。

CursorType property／CursorType プロパティ

This property works as in ADODB, except that only the adOpenStatic (when CursorLocation = adUseClient) and adForwardOnly values are guaranteed to work in all cases. All other values refer to server-side cursors and are

valid only when accessing a SQL Server database. If a CursorType value isn't supported, the Open method throws an exception.

adOpenStatic 値（CursorLocation = adUseClient の場合）と adForwardOnly 値がすべての場合において動作することが保証されている以外は、このプロパティは ADODB と同じように動作します。他のすべての値はサーバーサイドカーソルを参照しており、SQL Server に接続するときのみ有効となります。CursorType 値がサポートされない場合、Open メソッドは例外を発生させます。

Please notice that under ADODB, it is possible to specify CursorType=Keyset (or Dynamic) and LockType=BatchOptmisitic. In this case, all updates are cached until a UpdateBatch command is issued. Conversely, the ADOLibrary updates each individual record immediately. To alert the user that the behavior is different, an exception is intentionally thrown when the UpdateBatch command is issued.

ADODB では、CursorType=Keyset（または Dynamic）と LockType=BatchOptmisitic を特定することができることに注意してください。この場合、すべての変更は、UpdateBatch コマンドが発行されるまで、キャッシュされます。反対に、ADOLibrary は各レコードを即座に更新します。動作が異なることを利用者に気付かせるため、UpdateBatch コマンドが発行されたとき、例外が故意に発生します。

Current version of ADOLibrary doesn't fully support server-side dynamic cursors.

ADOLibrary の現在のバージョンはサーバーサイドダイナミックカーソルを完全にはサポートしていません。

CurrentCursorRow property (.NET only)／CurrentCursorRow プロパティ（.NET のみ）

This readonly property returns the DataRow object that contains the current row. It is used only when the ADORecordset object is used to access a SQL Server server-side cursor.

この読み取り専用プロパティは現在の行を含む DataRow オブジェクトを返します。これは、ADORecordset オブジェクトが SQL Server のサーバーサイドカーソルに接続するために使用されるときにのみ使用されます。

DataAdapter property (.NET only)／DataAdapter プロパティ（.NET のみ）

This readonly property returns the DbDataAdapter object that is used internally when the ADORecordset object is used to open a client-side cursor. It can be used to access ADO.NET specific members that have no corresponding member under ADODB.

この読み取り専用プロパティは、ADORecordset オブジェクトがクライアントサイドカーソルをオープンするために使用されるときに内部的に使用される DbDataAdapter オブジェクトを返します。それは ADODB に対応するメンバーがない ADO.NET の特定のメンバーにアクセスするために使用することができます。

The ADORecordset class exposes additional readonly properties, named **OleDbDataAdapter, SqlDataAdapter**, and **OdbcDataAdapter** – which return a strongly-typed DataAdapter object, or Nothing if the ADO.NET data provider used internal doesn't match the property type.

ADORecordset クラスには、**OleDbDataAdapter**、**SqlDataAdapter**、**OdbcDataAdapter** という読み取り専用のプロパティが追加されています—これらのプロパティは強く型付けされた DataAdapter オブジェクトを返しますが、内部的に使用される ADO.NET データプロバイダがプロパティ型と一致しない場合は Nothing を返します。

DataReader property (.NET only)／DataReader プロパティ(.NET のみ)

This readonly property returns the DbDataReader object that is used internally when the ADORecordset object is used to open a forwardonly-readonly (FO-RO) cursor. It can be used to access ADO.NET specific members that have no corresponding member under ADODB, for example using the GetBoolean, GetInteger, etc. methods to read column values in a more efficient way.

この読み取り専用プロパティは、ADORecordset オブジェクトが forwardonly-readonly カーソルをオープンするために使用されるときに内部的に使用される DbDataReader オブジェクトを返します。それは、ADODB に対応するメンバーがない ADO.NET の特定のメンバー—例えば、より効率的な方法でカラムの値を読み込む、GetBoolean や GetInteger などを使用するメソッド—にアクセスするために使用することができます。

The ADORecordset class exposes additional readonly properties, named **OleDbDataReader, SqlDataReader**, and **OdbcDataReader** – that return a strongly-typed DataReader object, or Nothing if the ADO.NET data provider used internal doesn't match the property type.

ADORecordset クラスには、**OleDbDataReader**、**SqlDataReader**、**OdbcDataReader** といった読み取り専用プロパティが追加されています—これらのプロパティは強く型付けされた DataReader オブジェクトを返しますが、内部的に使用される ADO.NET データプロバイダがプロパティ型と一致しない場合は Nothing を返します。

DataSource property／DataSource プロパティ

This property is currently not implemented.

このプロパティは現在実装されていません。

DataTable property (.NET only)／DataTable プロパティ(.NET のみ)

This readonly property returns the DataTable that contains the rows that have been read from the database when the ADORecordset is used to open a client-side cursor. You can use this property to bind data to a .NET control, such as a DataGridView control.

この読み取り専用プロパティは、ADORecordset がクライアントサイドカーソルをオープンするために使用されたときにデータベースから読み込まれた行を含む DataTable を返します。ユーザーはこのプロパティを、DataGridView コントロールのような.NET コントロースにデータをバインドするために使用することができます。

The inner DataTable also used when working with SQL Server server-side cursors, in which case the DataTable contains only the current database row.

この内部的な DataTable は SQL Server のサーバーサイドカーソルを動作させるときにも使用されます。その場合、DataTable は現在のデータベースの行のみを含みます。

DataView property (.NET only)／DataView プロパティ(.NET のみ)

This readonly property returns the DataView that is associated to the inner DataTable and that honors the current Filter and Sort settings. This property is non-Nothing only when the ADORecordset is used to open a client-side cursor. You can use this property to bind data to a .NET control, such as a DataGridView control.

このプロパティは、内部の DataTable と結合している DataView を返すだけでなく、Filter や Sort の設定も引き継ぎます。このプロパティはクライアントサイドカーソルをオープンするために ADORecordset が使用されるときは Nothing になりません。ユーザーは、DataGridView コントロールのような.NET コントロールにデータをバインドするためにこのプロパティを使用することができます。

EnforceConstraintsOnLoad property (.NET only)／EnforceConstraintsOnLoad プロパティ(.NET のみ)

When used to create a client-side static cursor, the Open method of the ADORecordset reads rows from the database and loads them into an private, temporary DataSet object whose EnforceConstraints property is set to False, and only later data is moved into a DataTable. This step is necessary to perfectly replicate the ADODB behavior, which never raises an error if an incoming row doesn't match all required constraints. (For example, ADODB doesn't raise an error if the incoming row contains a NULL value for a non-nullable column.)

クライアントサイドの静的カーソルが使用される場合、ADORecordset の Open メソッドはデータベースから行を読み、EnforceConstraints プロパティが False にセットされた private で temporary な DataSet オブジェクトにそれらを読み込んだ後にのみ、DataTable にデータを移します。このステップは ADODB の、取り込まれる行が要求される全制約に該当しない場合、エラーを発生させないという動作を完全に複製するために不可欠です。(例えば、取り込まれる行が非 NULL 列に NULL を含む場合、ADODB はエラーを発生させません。)

On the other hand, loading data into a temporary DataSet slightly degrades performances. If you are sure that incoming data doesn't violate any database constraint you can skip this intermediate step by setting the EnforceConstraintsOnLoad property to True:

一方、temporary の DataSet にデータを読み込むことはわずかにパフォーマンスを低下させます。取り込みデータがデータベースの制約に違反することが確実な場合、EnforceConstraintsOnLoad プロパティを True にセットして中間ステップをスキップすることができます。

```
Dim rs As New ADORecordset
rs.EnforceConstraintsOnLoad = True
```

When an ADORecordset is created, this property is set equal to ADOConfig.EnforceConstraintsOnLoad property.

ADORecordset が作成されるとき、このプロパティは ADOConfig.EnforceConstraintsOnLoad プロパティと等しくなります。

FetchProgress event／FetchProgress イベント

This event is never raised and is marked as obsolete, because ADOLibrary doesn't support asynchronous fetching. (It does support asynchronous connection and execution, though.)

このイベントは ADOLibrary が非同期フェッチをサポート」しないため発生せず、すでに使用されていないイベントと見なされます。(ただし、非同期接続と非同期実行はサポートされます)

Filter property／Filter プロパティ

Under ADODB this property can be assigned a string containing a WHERE clause, a FilterGroupEnum enumerated value, or an array of bookmarks. The ADOLibrary accepts the first two types, but supports neither bookmark arrays nor the (undocumented) value 4-adFilterPredicate. Moreover, the value 3-adFilterFetchedRecords is ignored,

because ADOLibrary doesn't maintain a row cache. Finally, when setting the value 2-adFilterAffectedRecord after invoking the UpdateBatch method, bear in mind that deleted records that were successfully committed to the database won't be included in the filtered Recordset.

ADODB ではこのプロパティに WHERE 節や FilterGroupEnum 列挙型やブックマークの配列を含む文字列を割り当てることができます。ADOLibrary はその二つの型を受け取りますが、ブックマーク配列と非公式な 4-adFilterPredicate 値はどちらもサポートしません。さらに、ADOLibrary が行キャッシュを保持しないため、3-adFilterFetchedRecords 値は無視されます。最後に、UpdateBatch メソッドを読みだした後で 2-adFilterAffectedRecord 値を設定すると、データベースに対して正常にコミットされた削除済みレコードはフィルタリングされたレコードセットには含まれなくなることに注意してください。

Find method／Find メソッド

This method isn't currently implemented for server-side cursors.

このメソッドはサーバーサイドカーソル用には現在提供されていません。

IgnoreDataSchema property (.NET only)／IgnoreDataSchema プロパティ(.NET のみ)

If this property is set to False (the default value) and you open a Recordset in batch optimistic mode, ADOLibrary queries the data source and retrieves schema information about individual fields. This information is used to correctly implement a few members of the Field object, namely DefinedSize, NumericScale, Precision, and Properties. If you are sure that your code doesn't rely on this metadata information, you can speed up Recordset.Open methods by setting this property to True.

このプロパティに(初期値)False をセットし、一括楽観モードでレコードセットをオープンする場合、ADOLibrary はデータソースに照会し、個々のフィールドに関するスキーマ情報を取り出します。この情報は Field オブジェクトの2、3のメンバーを正しく実装するために使用されます。すなわち、DefinedSize、NumericScale、Precision と Properties です。コードがこのメタデータ情報に依存しないことが確実である場合、このプロパティに True をセットして Recordset.Open メソッドの速度を向上させることができます。

```
Sub ReadWithNoSchema(ByVal cn As ADOConnection, ByVal rs As ADORecordset)
    rs.IgnoreDataSchema = True
    rs.Open "Select * From Orders", cn
    ' ...
End Sub
```

When an ADORecordset is created, this property is set equal to ADOConfig.IgnoreDataSchema property.

ADORecordset が作成されるとき、このプロパティは ADOConfig.IgnoreDataSchema プロパティと等しくなります。

Index property／Index プロパティ

The Index property is unsupported. It always returns an empty string; assigning a different value throws an exception (if ADOConfig.ThrowOnUnsupportedMembers is True).

この Index プロパティはサポートされません。それは常に空文字を返します。また、(ADOConfig.ThrowOnUnsupportedMembers が True の場合)異なる値を割り当てると例外が発生します。

LibraryKind property (.NET only)／LibraryKind プロパティ(.NET のみ)

By default, ADOLibrary internally uses the OleDb ADO.NET data provider. If you know that the ADORecordset object will only access a specific database – for example, Microsoft SQL Server – you can improve performance by assigning a new value to this property:

デフォルトでは、ADOLibrary は OleDb 用 ADO.NET データプロバイダを内部的に使用します。その ADORecordset オブジェクトが特定のデータベースにのみ接続することがわかっている場合、—例えば、Microsoft SQL Server—このプロパティに新しい値を割り当てて性能を向上させることができます。

```
Dim rs As New ADORecordset
rs.LibraryKind = ADOLibraryKind.SqlClient
```

Valid values for the LibraryKind property are: OleDb, SqlClient, Odbc, OracleClient. However, version 1.0 of ADOLibrary only supports the OleDb and SqlClient values. Any other value throws an exception (if ThrowOnUnsupportedMembers is True) or are ignored. If this property is left unassigned, it is set equal to the value of ADOConfig.LibraryKind property or equal to the LibraryKind property of the ADOConnection object used to open the recordset.

LibraryKind プロパティにに対して有効な値は、OleDb、SqlClient、Odbc、OracleClient です。 ただし、ADOLibrary のバージョン 1.0 は OleDb と SqlClient のみサポートします。その他の値は例外を発生させるか（ThrowOnUnsupportedMembers が True なら）無視されます。このプロパティが設定されない場合、ADOConfig.LibraryKind プロパティの値に等しくなるか、レコードセットをオープンするために使用される ADOConnection オブジェクトの LibraryKind プロパティと等しくなります。

LockType property／LockType プロパティ

This property works as in ADODB, except that only the adLockReadOnly and adLockBatchOptimistic values are guaranteed to work in all cases. All other values refer to server-side cursors and are valid only when accessing a SQL Server database. If a LockType value isn't supported, the Open method throws an exception.

このプロパティは、adLockReadOnly 値と adLockBatchOptimistic 値がすべての場合において動作することが保証されていることを除いて、ADODB のように動作します。他のすべての値はサーバーサイドカーソルを参照し、SQL Server に接続するときのみ有効です。LockType 値がサポートされない場合、Open メソッドは例外を発生させます。

MarshalOptions property／MarshalOptions プロパティ

The MarshalOptions property is unsupported. It always returns 0–adMarshalAll; assigning a different value throws an exception (if ADOConfig.ThrowOnUnsupportedMembers is True).

MarshalOptions プロパティはサポートされていません。それは常に 0–adMarshalAll を返します。（ADOConfig.ThrowOnUnsupportedMembers が True の場合は）異なる値を割り当てると例外が発生します。

MaxRecords property／MaxRecords プロパティ

This property works as in ADODB, except it is implemented internally by adding a TOP clause to the SELECT statement sent to the database. If you assign a nonzero value to MaxRecords and the SQL dialect doesn't support the TOP clause, the Open method throws an exception.

このプロパティは、データベースに送信される SELECT ステートメントに TOP 節を追加することで内部的に実装されていることを除いて、ADODB と同じように動作します。MaxRecords に 0 ではない値を割り当てた場合、その SQL は TOP 節をサポートせず、Open メソッドは例外を発生させます。

Move method／Move プロパティ

This method doesn't work and raises an exception when used with server-side dynamic cursors.

サーバーサイド動的カーソルが使用されるとき、このメソッドは動作せず、例外を発生させます。

Open method／Open メソッド

This method fails with server-side keyset and dynamic cursors if the SELECT statement doesn't include at least one non-nullable key column.

SELECT 文が少なくとも一つの非 NULL キー列を含まない場合、このメソッドはサーバーサイドのキーセットと動的カーソルには使用できません。

RecordCount property／RecordCount プロパティ

In current ADOLibrary version, this property returns an inconsistent value when used with server-side cursors and the Filter property is being used to filter rows.

現在の ADOLibrary のバージョンでは、サーバーサイドカーソルが使用される場合、このプロパティは矛盾した値を返しますが、Filter プロパティはフィルター行に使用されます。

Resync method／Resync メソッド

This method works as in ADODB and correctly honors the **Resync Command** dynamic property. However, if a custom resync command is specified in the Resync Command property, then the command must include a single "?" (question mark) placeholder and when the SELECT statement contains a single key column.

このメソッドは ADODB と同じように動作し、動的な **Resync Command** も正しく引き継いでいます。しかし、Resync Command プロパティでカスタム resync コマンドが指定された場合、また、SELECT 文がひとつのキー列を含んでいるとき、そのコマンドは「?」プレースホルダーを含めなければなりません。

RowIndex property (.NET only)／RowIndex プロパティ(.NET のみ)

This property sets or returns the zero-based row index of the current row into the inner DataTable, and is significant only when the ADORecordset is used to open a client-side cursor. Assigning this property is roughly equivalent to a **Move(n)** method.

このプロパティは DataTable 内部に 0 から始まる現在の行の行インデックスをセットまたは返しますが、ADORecordset がクライアントサイドカーソルをオープンするために使用されるときに重要な意味を持ちます。このプロパティを割り当てることは **Move(n)**メソッドと大体の意味において等価です。

Save method／Save メソッド

This method works as in ADODB, except for two details. First, it only works with client-side recordsets (CursorLocatoin=adUseClient). Second, the storage format – both in binary and XML mode – is different from

ADODB and therefore you can't use this method to persist recordsets and share them between VB6 and .NET applications. Moreover, only the binary format (adPersistADTG) is supported when saving to a Stream object.

このメソッドは二つの特徴を除いて、ADODB と同じように動作します。第 1 に、(CursorLocatoin=adUseClient である)クライアントサイトレコードセットに対してのみ動作します。第 2 に、ストレージフォーマット—バイナリと XML の両方—が ADODB とは異なっているため、レコードセットを使い続けるためにこのメソッドを使用することはできませんし、VB6 アプリケーションと.NET アプリケーションの間でそれらを共有することもできません。さらに、Stream オブジェクトに保存する場合、バイナリフォーマット(adPersistADTG)のみサポートされます。

Seek method／Seek メソッド

The Seek method is unsupported. Invoking it throws an exception (if ADOConfig.ThrowOnUnsupportedMembers is True).

Seek メソッドはサポートされません。(ADOConfig.ThrowOnUnsupportedMembers が True の場合は)それを呼び出そうとすると例外が発生します。

State property／State プロパティ

This property works as in ADODB, except it can never return the value 8–adStateFetching, because ADOLibrary doesn't support asynchronous fetching. (It can return the values 2–adStateConnecting and 4–adStateExecuting because ADOLibrary supports asynchronous connections and asynchronous commands.)

ADOLibrary が非同期フェッチをサポートしないため、8–adStateFetching 値を返すことができないことを除いて、このプロパティは ADODB と同じように動作します。(ADOLibrary が非同期接続と非同期コマンドをサポートするので、2–adStateConnecting 値と 4–adStateExecuting 値を返すことはできます。)

Status property／Status プロパティ

The implementation of this property is incomplete in current version of ADOLibrary, in that it can return only one of the following enumerated values: adRecOK, adRecUnmodified, adRecDBDeleted, adRecNew, adRecDeleted, adRecModified, adRecConcurrencyViolaton.

このプロパティの実装は ADOLibrary の現在のバージョンにおいては不完全です。そのため、次の列挙型の値のうちのひとつを返すことのみ可能です。すなわち、adRecOK、adRecUnmodified、adRecDBDeleted、adRecNew、adRecDeleted、adRecModified、adRecConcurrencyViolaton です。

StayInSync property／StayInSync プロパティ

The StayInSync property is unsupported. In always returns True; assigning a different value throws an exception (if ADOConfig.ThrowOnUnsupportedMembers is True).

StayInSync プロパティはサポートされません。常に True が返されます。また、(ADOConfig.ThrowOnUnsupportedMembers が True の場合は)異なる値を割り当てると例外が発生します。

Supports method／Supports メソッド

For highest compatibility, this method returns the same value that would be returned under ADODB. However, because some advanced features aren't currently supported by ADOLibrary, a feature that is returned as

“supported” and yet the VB.NET code can later throw an exception when the corresponding property or method is actually invoked.

最も高い互換性に対して、このメソッドは、ADODB において返す値と同じ値を返します。しかしながら、いくつかの高度な機能は ADOLibrary によって現在サポートされていません。「supported」が返される機能でありながら、対応するプロパティやメソッドが実際に呼び出されると VB.NET のコードが例外を発生させる可能性があります。

SynchronizingObject property (.NET only)／SynchronizingObject プロパティ(.NET のみ)

The ADORecordset object can raise asynchronous events. By default, these events run in thread other than the main thread of the VB.NET application. This detail can cause problems and unexpected crashes if the code in the event handler accesses one or more user interface elements, such as a form or a control. You can avoid this problem by assigning a form or a control reference to the SynchronizingObject property. Any form or control will do:

ADORecordset オブジェクトは非同期イベントを発生させることができます。デフォルトでは、これらのイベントは VB.NET アプリケーションのメインスレッド以外のスレッドで実行されます。イベントハンドラーのコードが、フォームやコントロールのような、一つ以上のユーザーインターフェース要素にアクセスする場合、細部において問題が起こったり、予期しないクラッシュが発生する可能性があります。利用者は、SynchronizingObject プロパティを参照するフォームやコントロールを割り当てることで、この問題を回避することができます。どのようなフォームやコントロールでも次のように記述することができます。

```
' if inside a form class, "Me" is a reference to the current form
Dim rs As New ADORecordset
rs.SynchronizingObject = Me
```

When an ADORecordset is created, this property is set equal to ADOConfig.Synchronizing property.

ADORecordset が作成されると、このプロパティは ADOConfig.Synchronizing プロパティと等しくなります。

UpdateBatch method／UpdateBatch メソッド

This method works as in ADODB, except for the following details:

以下の特徴を除いて、このメソッドは ADODB と同じように動作します。

- the SELECT statement must include the key/identity field of the table (or tables, if it’s a JOIN statement).
  SELECT 文はテーブル（JOIN ステートメントの場合は複数のテーブル）のキーまたは識別用フィールドを含めなければなりません。
- JOIN SQL statements are supported, provided that
  以下の場合、JOIN ステートメントはサポートされます。
  - the key/identity field of each table is included in the field list
    各テーブルのキーまたは識別用フィールドがフィールドリストに含まれている。
  - if * is used to indicate “all fields”, then the joined tables must not have fields with same name
    *が「全フィールド」の意味で使用される場合、結合されたテーブルは同一名のフィールドを持ってはならない。

- old join syntax (e.g. SELECT * FROM Table1,Table2) isn't supported
  古い join 構文（例　SELECT * FROM Table1,Table2）はサポートされません。
- all valid SQL syntax for table names are supported (e.g. database.dbo.table), except
  以下の場合を除いて、テーブル名（例　database.dbo.table）に対するすべての有効な SQL 構文はサポートされます。
- field names and table names cannot include dots (even if names are included between square brackets or double quotes)
  （角括弧や二重引用符で囲まれていたとしても）フィールド名とテーブル名にドットを含むことはできません。
- if a table name in the FROM clause is embedded in square brackets (or double quotes), then the same enclosing delimiters must be used if the table name is used a prefix for the field name. For example:
  FROM 節のテーブル名が角括弧（または二重引用符）で囲まれていた場合、テーブル名がフィールド名の接頭辞として使用されるのなら、デリミタを含んでいる同じ名称が使用されなくてはなりません。例えば、

  SELECT [Order Details].OrderID FROM [Order Details]　　is supported サポートされます
  SELECT [Order Details].OrderID FROM "Order Details"　　isn't supported サポートされません

- derivate tables aren't supported (e.g. SELECT * FROM (SELECT ....)).
  派生テーブルはサポートされません。（例　SELECT * FROM (SELECT ....)）
- the UNION clause isn't supported.
  UNION 節はサポートされません。
- if two UpdateBatch commands are issued on the same recordset and you need to modify one or more fields of a row that has been already batch-updated, it is necessary that you perform the following command after the first UpdateBatch and before modifying the fields:
  二つの UpdateBatch コマンドが同じレコードセットに発行され、既に一括更新された行の一つ以上のフィールドを変更する必要がある場合、最初の UpdateBatch の後とフィールドを変更する前に以下のコマンドを実行する必要があります。

```
myRecordset.DataTable.AcceptTable()
```

Notice that an ADODB UpdateBatch command works even when CursorType=Keyset/Dynamic and LockType=BatchOptimistic. In this case, ADODB postpones all changes in the DB until the UpdateBatch command is executed. In the same circumstances, the ADOLibrary updates each record immediately.

ADODB の UpdateBatch コマンドは CursorType=Keyset/Dynamic や LockType=BatchOptimistic であっても動作することに注意してください。この場合、ADODB は、UpdateBatch が実行されるまで、データベースにおける変更すべてを延期します。同じ状況において、ADOLibrary は各レコードを即座に更新します。

If the table being updated has an auto-increment field or uses a key that is generated by the database server, current version of ADOLibrary can correctly retrieve the auto-generated key only when working with SQL Server databases.

更新されるテーブルに自動インクリメント列がある場合やデータベースサーバーによって生成されるキーを使用する場合、現在のバージョンの ADOLibrary は SQL Server データベースに対して動作するときのみ自動的に生成されたキーを正しく取り出します。

The ADOCommand class／ADOCommand クラス

The ADOCommand class is similar and behaves like the ADODB.Command class, except for the following details.

以下の特徴を除いて、ADOCommand クラスは ADODB.Command クラスと似ており同じように動作します。

Command property (.NET only)／Command プロパティ(.NET のみ)

This readonly property returns the DbCommand object that is used internally by the ADOCommand object. It can be used to access ADO.NET specific members that have no corresponding member under ADODB, for example using the ExecuteScalar method to retrieve single values in a more efficient way.

この読み取り専用プロパティは ADOCommand オブジェクトによって内部的に使用される DbCommand オブジェクトを返します。それは ADODB に対応するメンバーがない ADO.NET の特定のメンバーにアクセスするために使用することができます。例えば、ExecuteScalar メソッドを使用すればより効率的な方法で一つの値を取得することができます。

```
Sub GetSingleValue(ByVal cmd As ADOCommand)
    Dim value As Object
    value = cmd.Command.ExecuteScalar()
    ' ...
End Sub
```

The ADOCommand class exposes additional readonly properties, named **OleDbCommand, SqlCommand**, and**OdbcCommand** – which return a strongly-typed Command object, or Nothing if the ADO.NET data provider used internal doesn't match the property type.

ADOCommand クラスには、**OleDbCommand**、**SqlCommand**、**OdbcCommand** という読み取り専用プロパティが追加されています。これらのプロパティは強く型付けされた Command オブジェクトを返しますが、内部で使用される ADO.NET データプロバイダがプロパティ型と一致しない場合は Nothing を返します。

CommandBehavior property (.NET only)／CommandBehavior プロパティ(.NET のみ)

The value of this property is used internally and is assigned to the CommandBehavior property of the inner Command object. For example, if you know that the rowset being read contains only a single row, you can slightly optimize execution as follows

このプロパティの値は、内部の Command オブジェクトの CommandBehavior プロパティに対して割り当てられ、かつ内部的に使用されます。例えば、読み込まれる行セットが単一行のみを含むことが分かっている場合、次のように若干楽観的な実行を行うことができます。

```
Sub TestSingleRow(ByVal cmd As ADOCommand)
```

```
    cmd.CommandBehavior = CommandBehavior.SingleRow
    Dim rs As ADORecordset = cmd.Execute()
    ' ...
End Sub
```

IgnoreDataSchema property (.NET only)／IgnoreDataSchema プロパティ(.NET のみ)

If this property is set to False (the default value) and you open a Recordset in batch optimistic mode, ADOLibrary queries the data source and retrieves schema information about individual fields. This information is used to correctly implement a few members of the Field object, namely DefinedSize, NumericScale, Precision, and Properties. If you are sure that your code doesn't rely on this metadata information, you can speed up Recordset.Open methods by setting this property to True.

このプロパティに(初期値)False をセットし、一括楽観モードでレコードセットをオープンする場合、ADOLibrary はデータソースに照会し、個々のフィールドに関するスキーマ情報を取り出します。この情報は Field オブジェクトの2、3のメンバーを正しく実装するために使用されます。すなわち、DefinedSize、NumericScale、Precision と Properties です。コードがこのメタデータ情報に依存しないことが確実である場合、このプロパティに True をセットして Recordset.Open メソッドの速度を向上させることができます。

```
Sub ReadWithNoSchema(ByVal cmd As ADOCommand)
    cmd.IgnoreDataSchema = True
    Dim rs As ADORecordset = cmd.Execute()
    ' ...
End Sub
```

The default value of this property when the ADOCommand object is instantiated is equal to ADOConfig.IgnoreDataSchema.

ADOCommand オブジェクトがインスタンス化されたときのこのプロパティの初期値は ADOConfig.IgnoreDataSchema と等価です。

LibraryKind property (.NET only)／LibraryKind プロパティ(.NET のみ)

By default, ADOLibrary internally uses the OleDb ADO.NET data provider. If you know that a given ADOCommand object will only access a specific database – for example, Microsoft SQL Server – you can improve performance by assigning a new value to this property:

デフォルトでは、ADOLibrary は OleDb 用 ADO.NET データプロバイダを内部的に使用します。ADOCommand オブジェクトが特定のデータベースにのみ接続することがわかっている場合、—例えば、Microsoft SQL Server—このプロパティに新しい値を割り当てて性能を向上させることができます。

```
Dim cmd As New ADOCommand
cmd.LibraryKind = ADOLibraryKind.SqlClient
```

Valid values for the LibraryKind property are: OleDb, SqlClient, Odbc, OracleClient. However, version 1.0 of ADOLibrary only supports the OleDb and SqlClient values. Any other value throws an exception (if

ThrowOnUnsupportedMembers is True) or are ignored. If this property is left unassigned, it is set equal to the value of ADOConfig.LibraryKind property.

LibraryKind プロパティにに対して有効な値は、OleDb、SqlClient、Odbc、OracleClient です。 ただし、ADOLibrary のバージョン 1.0 は OleDb と SqlClient のみサポートします。その他の値は例外を発生させるか（ThrowOnUnsupportedMembers が True なら）無視されます。このプロパティが設定されない場合、ADOConfig.LibraryKind プロパティの値に等しくなります。

Name property／Name プロパティ

This property retains the value assigned to it, but is otherwise ignored by ADOLibrary.

このプロパティは割り当てられた値を保持しますが、そうでなければ ADOLibrary によって無視されます。

CommandStream property／CommandStream プロパティ

This property is marked as obsolete always returns Nothing. Assigning a different value to it throws an exception (if ADOConfig.ThrowOnUnsupportedMembers property is True).

このプロパティは既に使用されていないと見なされ、常に Nothing を返します。異なる値を割り当てると（ADOConfig.ThrowOnUnsupportedMembers プロパティが True の場合は）例外が発生します。

Dialect property／Dialect プロパティ

This property is marked as obsolete and always returns the following GUID:

このプロパティは既に使用されていないと見なされ常に次の GUID が返されます。

{C8B521FB-5CF3-11CE-ADE5-00AA0044773D}

which corresponds to the SQL language dialect. Assigning a different value to it throws an exception (if ADOConfig.ThrowOnUnsupportedMembers property is True).

これはその SQL 言語に対応します。異なる値を割り当てると（ADOConfig.ThrowOnUnsupportedMembers プロパティが True の場合は）例外が発生します。

NamedParameters property／NamedParameters プロパティ

This property is marked as obsolete always returns False. Assigning a different value to it throws an exception (if ADOConfig.ThrowOnUnsupportedMembers property is True).

このプロパティは既に使用されていないと見なされ常に False を返します。異なる値を割り当てると（ADOConfig.ThrowOnUnsupportedMembers プロパティが True の場合は）例外が発生します。

Parameters collection／Parameters コレクション

As it happens with ADODB, when an ADOCommand object contains a query with parameters, ADOLibrary has to parse the SQL statement to isolate each and every parameter, corresponding to “?” placeholders. In addition to having an open connection, the following restrictions apply to the SQL SELECT statement assigned to the CommandText property:

ADODB において発生でするように、ADOCommand オブジェクトがパラメータを伴うクエリーを含むとき、ADOLibrary は「?」プレースホルダに対応して各パラーメータを分離するために SQL 文を解析しなければなりません。さらにオープン時接続に加えて、次の制限が CommandText プロパティに割り当てられる SELECT 文に対して適用されます。

- aliased columns are supported (e.g. "LastName AS LN"), but aliased expressions aren't (e.g. "SUBSTRING(title,1,10) AS Title").
  列の別名（例　"LastName AS LN"）はサポートされますが、式の別名（例　"SUBSTRING(title,1,10) AS Title"）はサポートされません。
- aliased tables aren't supported.
  テーブルの別名はサポートされません。
- derived tables aren't supported (e.g. "SELECT a.au_lname AS Name, d1.title_id FROM authors a, (SELECT title_id, au_id FROM titleauthor) AS d1").
  派生テーブルはサポートされません。（例　"SELECT a.au_lname AS Name, d1.title_id FROM authors a, (SELECT title_id, au_id FROM titleauthor) AS d1"）
- parameters can't precede the BETWEEN keyword: for example　"…WHERE fieldname BETWEEN ? AND ?"　is supported but　"... ? BETWEEN 10 AND 20"　is not.
  BETWEEN の前にパラメータを置くことはできません。例えば、「…WHERE fieldname BETWEEN ? AND ?」はサポートされますが、「... ? BETWEEN 10 AND 20」はサポートされません。
- if the statement contains a nested SELECT, parameters can't appear both in the main WHERE clause and in the nested SELECT statements. (Parameters can correspond to fields belonging to different tables only if the tables appear in a JOIN.)
  ステートメントが入れ子になった SELECT を含む場合、パラメータは主となる WHERE 節にも入れ子の SELECT 文にも存在することができません。パラメータは、テーブルが JOIN にある場合のみ、異なるテーブルに属するフィールドに対応することができます。

The following restrictions apply to INSERT SQL statements:

次の制限は INSERT 文に対して適用されます。

- the list of columns must be present (* isn't allowed) and the VALUES keyword must be present.
  列リスト（*は許可されません）と VALUES キーワードは必須です。
- parameters used in the VALUES can't appear in expressions (i.e. must be the only value assigned to a field).
  VALUES に使用されるパラメータは式の中で使うことはできません。（すなわち、フィールドに対して割り当てられた値にのみ使用できます。）

The following restrictions apply to EXEC SQL statements and stored procedure invocations:

次の制限は EXEC 文とストアドプロシージャの呼び出しに適用されます。

- parameters can appear in the list of stored procedure arguments, but can't be part of an expression. For example, "EXEC spname ?, '1abc', ?" is legal, but "EXE spname ?+12" is not.
  パラメータはストアドプロシージャ引数に使用できますが、式の一部にすることはできません。例えば、「EXEC spname ?, '1abc', ?」は適合しますが、「EXE spname ?+12」は適合しません。

Prepared property／Prepared プロパティ

This property is marked as obsolete always returns False. Assigning a different value to it throws an exception (if ADOConfig.ThrowOnUnsupportedMembers property is True).

このプロパティは既に使用されていないと見なされ常に False を返します。異なる値を割り当てると（ADOConfig.ThrowOnUnsupportedMembers プロパティが True の場合は）例外が発生します。

Execute method／Execute メソッド

This method works as in ADODB, except it throws an exception if multiple, semicolon-delimited SQL statements are passed as an argument. Moreover, if the Execute method invokes a stored procedure that returns no records and you are interested in the value returned by the stored procedure, then it is mandatory to specify adExecuteNoRecords in the *options* argument (in ADODB passing this value is optional):

このメソッドは、複数のセミコロンで区切られた SQL 文が引数として渡されると例外を発生させる以外は、ADODB と同じように動作します。さらに、ストアドプロシージャによって返される値に関心があるにも関わらず、Execute メソッドがレコードを返さないストアドプロシージャを呼び出す場合、*options* 引数（ADODB ではこの値は任意です）で adExecuteNoRecords を指定することが必須となります。

```
' cmd points to a stored proc that returns no records
' and has an output parameter
cmd はレコードを返さないストアドプロシジャを指しており、出力用パラメータを一つ持っています
cmd.Execute( , , ADOExecuteOptionEnum.adExecuteNoRecords)
' you can retrieve the stored procedure's return value now
ここでストアドプロシージャの戻り値を取得することができます
Dim retValue As Object = cmd.Parameters(0).Value
```

The ADOField class／ADOField クラス

The ADOField class is similar to and behaves like the ADODB.Field class, except for the following details.

ADOField クラスは、次にあげる特徴以外は ADODB.Field クラスと似ており、同じように動作します。

Attributes property／Attributes プロパティ

The Attributes property works as in ADODB, except it never returns the following bits: adFldMayDefer, adFldUnknownUpdatable, adFldFixed, adFldMayBeNull, adFldRowID, adFldRowVersion, adFldCacheDeferred

Attributes プロパティは次の値を戻さないこと以外は ADODB のように動作します。すなわち、adFldMayDefer、adFldUnknownUpdatable、adFldFixed、adFldMayBeNull、adFldRowID、adFldRowVersion、adFldCacheDeferred は戻されません。

Status property／Status プロパティ

The Status property is marked as obsolete and always returns zero, because it is only useful with Fields belonging to a Record object (which the ADOLibrary doesn't support).

Status プロパティは、(ADOLibrary がサポートしない)Record オブジェクトに属するフィールドにとって便利であるという理由から、既に使用されていないと見なされ常に 0 を返します。

HasDefaultValue (.NET)／HasDefaultValue プロパティ(.NET のみ)

This property (which is missing in ADODB) should be set to True for non-nullable fields for which a default value is defined inside SQL Server. Without this information, ADOLibrary is unable to build the correct SQL string when a new record is added to a server-side keyset and dynamic cursors.

このプロパティは、初期値が SQL Server 内部で定義される非 NULL 列に対しては True をセットされるべきです。この情報なしでは、新しいレコードがサーバーサイドのキーセットや動的カーソルに追加される場合、ADOLibrary は正しい SQL 文をビルドすることができません。

You typically need to set this property only once, immediately after opening a Recordset with CursorType=adOpenKeyset or CursorType=adOpenDynamic, only for those fields that have a default value defined in SQL Server, as in this example:

通常はこのプロパティを、レコードセットが CursorType=adOpenKeyset または CursorType=adOpenDynamic でオープンされた直後に、SQL Server に定義された初期値を持つそれらのフィールドに対してのみ、次の例のように、一回だけセットしなければなりません。

```
rs.Open("SELECT * FROM Customers", myConnection, adOpenKeyset)
' we know that the Country field has a default value set equal to "US"
私たちは Country 列が「US」に等しい初期値を持つことを知っています
rs.Fields("Country").HasDefaultValue = True
...
```

If you open a Recordset other than keyset or dynamic, or if you don't plan to add records to this Recordset, then you can ignore the HasDefaultValue property.

キーセットや動的ではないレコードセットをオープンする場合、もしくは、このレコードセットにレコードを追加するつもりがない場合、HasDefaultValue プロパティを無視することができます。

The ADOParameter class／ADOParameter クラス

The ADOParameter class is similar to and behaves like the ADODB.Parameter class, except for the following details.

ADOParameter クラスは ADODB.Parameter クラスに似ており、次の特徴を除いて同じように動作します。

Attributes, NumericScale, Precision, Type, Size properties／Attributes プロパティ、NumericScale プロパティ、Precision プロパティ、Type プロパティ、Size プロパティ

For improved performance, these properties are read "on demand", when any of them is accessed for the first time. For this reason, the first time you access any of these properties (for any parameter of a given ADOCommand object) it is mandatory that the connection is still open, else an exception is thrown.

性能改善のため、これらのプロパティは、それらが最初にアクセスされるときに、「要求に応じて」読み込まれます。この理由のため、最初に（ADOCommand オブジェクトのいずれかのパラメータ対する）これらのプロパティにアクセスするときは、接続がオープンされていることが必須となり、そうでない場合は例外が発生します。

The ADOStream class／ADOStream クラス

The ADOStream class is similar to and behaves like the ADODB.Stream class, except for the following details.

ADOStream クラスは ADODB.Stream クラスに似ており、以下の特徴を除いて同じように動作します。

ReadText, WriteText, SkipLine methods／ReadText メソッド、WriteText メソッド、SkipLine メソッド

These methods aren't currently implemented

これらのメソッドは現在提供されていません。

The ADORecord class／ADORecord クラス

None of the members in this class is currently implemented.

このクラスのメンバーは現在まったく提供されていません。