White Paper
Micro Focus Visual COBOL

COBOL – .NET プログラミング言語の新たな選択肢

.NET Framework は 2002 年の初版リリース以降、マイクロソフト社が技術トレンドを取り込みつつ機能充実を図ってきたアプリケーション開発用フレームワークです。2017 年 4 月にはバージョン 4.7 がリリースされ機能が一層熟成されました。開発者は同フレームワーク上で開発をすることで、高機能な Web アプリ、モバイルアプリ、クラウドアプリ等、幅広いアプリケーションを開発することができます。企業アプリケーションにとって必須要件であるセキュリティ関連機能も充実しており、エンタープライズの分野でも採用が進んでいる技術です。

COBOL はエンタープライズアプリケーションにおける主要な開発言語であり、.NET Framework 上でも C# や VB.NET と同様に .NET 言語として利用可能な言語です。広範な機能を提供する .NET Framework 上で、60 年近い実績を持つビジネスロジック記述力に優れた COBOL 言語を利用できることには大きな意味があります。.NET Framework に装備された先端技術を取り入れて、機能を刷新しながら企業固有のビジネスロジックを継続利用することも可能となります。本書では、どのように .NET Framework 上で COBOL を利用でき新たな価値創造につなげられるかを解説します。

MICRO FOCUS

目次

はじめに

2002 年の初版リリース以降 .NET Framework は改版の度に機能強化が加えられ 2017 年にはバージョン 4.7 が発表されました。また Window Azure の成熟度の高まりも相俟って新時代に突入したといえます。開発者は、Windows 8.1、Windows 10、Windows Server 2012、Windows Server 2016 の 32/64Bit 環境で実行される、サービス指向かつセキュアなアプリケーションを、一貫したプロセスと高い生産性で開発することができ、さらにクラウド環境にディプロイすることができます。

.NET Framework は、その登場以来のユニークで強力な特徴として、プログラミング言語に関する透過性を備えています。すべてのプログラミング言語は、共通言語ランタイム (CLR) を共通エンジンとする実行環境上で稼動するマイクロソフト中間言語 (MSIL) にコンパイルされ、共通のローディング・呼び出しメカニズムによって運用されます。このため、C#、VB.NET などのプログラミング言語はその文法の違いを超えて自由にインタフェースをもつことができます。あるプログラミング言語で実装されたクラスは、外部から見ればそれがどの言語で記述されているかには無関係に、透過的に参照することができます。

この特徴は、アセンブリの管理やガーベジコレクションなどの運用機能に関しても、言語を超越したサービスを提供します。これによってアプリケーション開発者は多くの作業から解放され、自身に興味のあるビジネスロジックに専念することができます。

あまり多くを語られてはいませんが、COBOL も .NET Framework 上のプログラミング言語として非常に初期から活躍してきています。上に述べたような .NET Framework のメリットは、すべて COBOL プログラマーにも享受できるものとなっています。Micro Focus Visual COBOL は、Visual Studio 2013、Visual Studio 2015、2017 等を IDE として採用し、.NET Framework 4.7 までのアプリケーション開発に活用することができます。

本書では、.NET Framework でのアプリケーション開発に COBOL 言語を使用する意義や技法について、いくつかの観点から考察します。

本書で示されているチュートリアルは、Micro Focus Visual COBOL を使用して実行されています。

1. オープン環境における COBOL

.NET Framework について語るに先立ち、オープン環境のシステム開発において COBOL が受け持っている役割について、最近の動向を概観しておきます。

1959 年に CODASYL によって仕様開発された COBOL 言語は、60 年近く使われ続けています。急速な技術革新の進むコンピュータサイエンスの中で、COBOL は現在も日々全世界の主要なビジネス業務を駆動しています。その理由は、単に過去に記述された膨大なプログラム資産があるからというだけではありません。事務処理専用言語として誕生した COBOL は、金融業務で必須とされる 38 桁までの十進固定小数点変数の四則演算を言語レベルでサポートしています。給与、保険料、利率といった金融業務では計算結果の誤差は許されませんので、数多くの企業の基幹システムにおいて COBOL は業務の心臓部であるビジネスロジック部分で活用されています。制御系やインターフェイス系の部分では .NET が大活躍している今、COBOL で書かれたビジネスロジックをコンポジットアプリケーション開発の中で活用することが重要な技術テーマになっている所以です。

日本国内について、マイクロフォーカスのお客様の動向を見ますと、レガシーマイグレーションとして既存 COBOL コードの活用は幅広くなされていますが、まったくの新規開発でも、非常に大規模なプロジェクトで COBOL が採用されています。マイクロフォーカスは、COBOL が使われ続けている動機付けとして以下の 3 点を重視しています。

1）リスクを最小化しなければならない大規模開発プロジェクトにとって、管理しやすい開発言語であること

他人が書いたコードでも読みやすい、生産されるコードの品質にばらつきが少ないといった管理面の良さは、COBOL が長く持っている強みです。短納期の大規模開発においては、このような管理しやすさは非常に重要です。多くの実績のある開発方法論は COBOL のような言語をベースとして実装されています。

2) 最新のテクノロジにすばやくキャッチアップしていること

もし COBOL が時代遅れの言語であるならば、.NET Framework や Java EE のアプリケーション開発の中で使い込んでゆくことはできないはずです。実際には、COBOL は .NET の歴史の最初から参加している言語ですし、Java EE ベースの開発のみならず多くの .NET 開発事例で COBOL が部分的に活用されています。昨今ではマイクロサービスを実現するための重要なテクノロジを COBOL が提供しています。
COBOL の将来に対する不安感がもしあるとすれば、それには実際の根拠がありません。

3）コスト・性能の要件を満たす現実的なソリューションであること

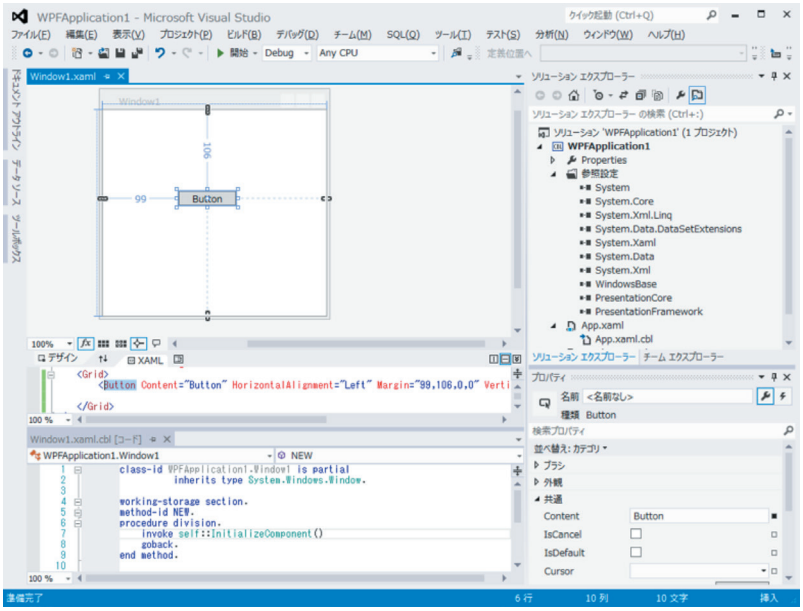
現状 COBOL で書かれているレガシー資産をベースにして、.NET Framework のような新しいテクノロジの上に業務を再構築しようとした場合、同じ言語を使用することは多くの面でコストメリットを生み出します。また、今でもシステム構成上大きな位置を占めるバッチ処理においては COBOL がもっとも効率的に処理を実行すると言われています。

2. .NET 開発での COBOL の意義

COBOL が .NET 対応されていることの意義は、以下の二つの異なる観点から見ることができます：
● .NET Framework 上のアプリケーション開発で COBOL 言語が使用できるということ
● COBOL 言語による開発で .NET Framework や Visual Studio が利用できるということ
以下、この二つの異なる観点から、.NET と COBOL の関連について説明します。

2.1 .NET のプログラミング言語としての COBOL

最初の観点は、言うまでも無く C# や VB.NET と並ぶ .NET プログラミング言語として COBOL があるということです。まず下のスクリーンショットを見てください。



これは XAML ベースの Windows Presentation Foundation のアプリケーションを Visual Studio 2015 を使用して作成しているところを示していますが、ここで使われている言語は COBOL です。アプリケーションで行っていることは、C# で WPF アプリケーションを作成するのとなんら変わることがありませんが、言語の文法は COBOL です。確かにここで使われている COBOL 言語は伝統的なバッチ処理で使われてきたものと若干異なってはいます。しかし、これは決して勝手に拡張された .NET 構文ではありません。ISO 国際標準で 2002 年に規格化されたオブジェクト指向 COBOL 構文をベースとした構文が採用されており、規格準拠のプログラミングで実現されています。ただし、.NET Framework に特有なオブジェクトモデルをサポートするために、一部固有の言語拡張もなされています。パーシャルクラスを記述するための CLASS-ID. クラス名　IS PARTIAL. などはこれに相当します。これらによって、COBOL は .NET アプリケーションを記述するために必要な機能をほぼすべて装備しています。COBOL だから特別であるということはほとんどありません。プログラマーは単に自分が文法を知っている言語というだけの理由で C# ではなく COBOL を選択することができます。

それに加えて COBOL には、他の言語にはない豊富なビジネス向けの機能があります。プログラマー定義の任意桁数の 10 進データを、自然な算術式を使用して四則演算することができ、その結果をさまざまな金額編集形式に宣言的に変換することができます。このような優れた機能を .NET 開発の中でも享受することができるのです。

2.2 従来型 COBOL 開発における .NET の活用

.NET にも Java にもまったく興味がなく、COBOL で純粋なバッチプログラムだけを書いているという方にとっても、実行環境として .NET Framework を採用することは検討に値するものです。

実行環境としての .NET Framework は高い信頼性を提供します。.NET Framework は Windows オペレーティングシステムの上位レイヤーとしてアプリケーションの実行を管理します。.NET ではないネイティブな Windows アプリケーションは、OS を直接利用して実行されるため、プログラムの誤動作や意図された不正動作などによって、メモリリークやセキュリティホールなどのさまざまな問題を起こし得ます。共通言語ランタイム上で動作するマネージコードではこの問題が緩和されます。このため .NET Framework は定期的なリブートを必要とせずに 365 日間運用が可能な環境といわれています。

.NET が提供する Web アプリケーション関連技術 ASP.NET が稼動する基盤 Internet Information Services は .NET のマネージドコードのスケーラブルなサーバーサイドの実行環境として動作します。これまでメインフレームで利用されてきたような大規模なアプリケーションを同環境へ展開し、より柔軟且つ利便度の高いアプリケーションへ進化させることも可能です。バッチアプリケーションであれば Windows Communication Foundation の技術を用いてマイクロサービス化しアプリケーションアーキテクチャーを改善させることもできます。オンラインアプリケーションであれば Web アプリケーションとして刷新させユーザビリティの高いアプリケーションへ進化させることも可能です。

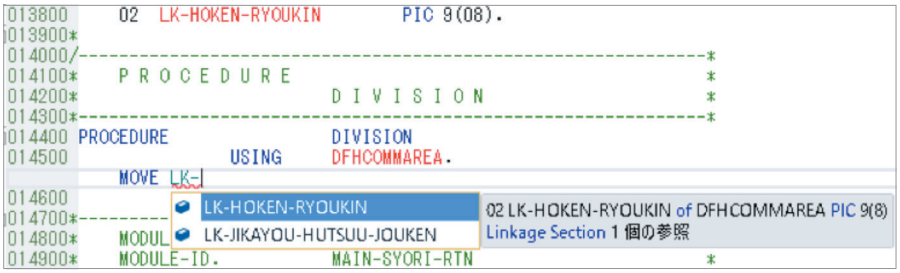
2.3 COBOL 開発における Visual Studio IDE の活用

Micro Focus Visual COBOL は Visual Studio IDE 上での COBOL アプリケーション開発を可能としています。同製品はこの IDE 向けにプログラマー個人の生産性向上を支援する機能、並びにチーム開発の効率性向上を支援する機能を提供しています。これらを駆使することで .NET 開発においても COBOL アプリケーションをスピーディー且つコーディング規約に沿った高品質なアプリケーションとして開発できます。本項ではこれらのうち代表的な支援機能を紹介します。

①入力補完

Visual Studio IDE における COBOL エディター上で何かを入力した後に Ctrl ＋ Space を打鍵すると入力された文字列で始まる COBOL の予約語、もしくは DATA DIVISION や ENVIRONMENT DIVISION で定義した変数等のユーザー定義語がリストされます。Ctrl ＋ Space を打鍵した時点で入力途中の文字列に続く予約語、ユーザー定義語がユニークになっていればその語で自動入力補完されます。本機能を活用することで効率的且つ正確にコーディングをすることができます。

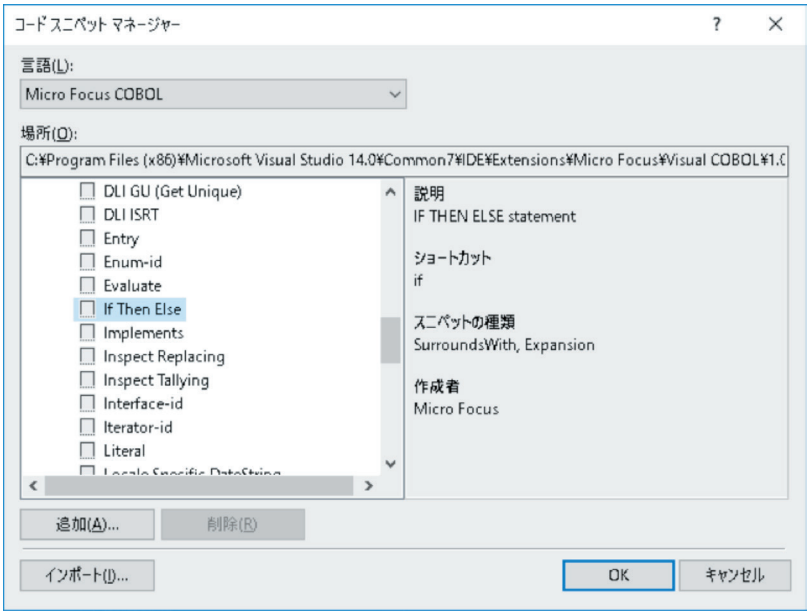
入力補完の実行イメージ：



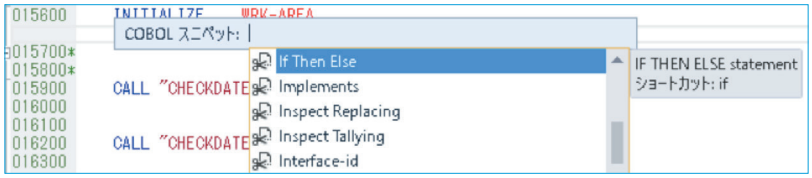
②コードスニペット

コードスニペットは定型的なコードの固まりをテンプレートとして登録し、必要に応じてソースへ挿入する機能です。Visual COBOL は COBOL 言語でよく利用されるであろうスニペットをビルドインで登録しています。カスタマイズしたスニペットをリストに追加することも可能です。追加する際は XML ベースの構成ファイルへスニペットを登録し、それを利用する環境で取り込みます。そのため、例えばコード規約に準じたコードスニペットを作成後、開発プロジェクトメンバーへ頒布し利用を徹底させれば、プロジェクト標準の記法を効率良く遵奉させることもできます。

コードスニペットマネージャーのイメージ：



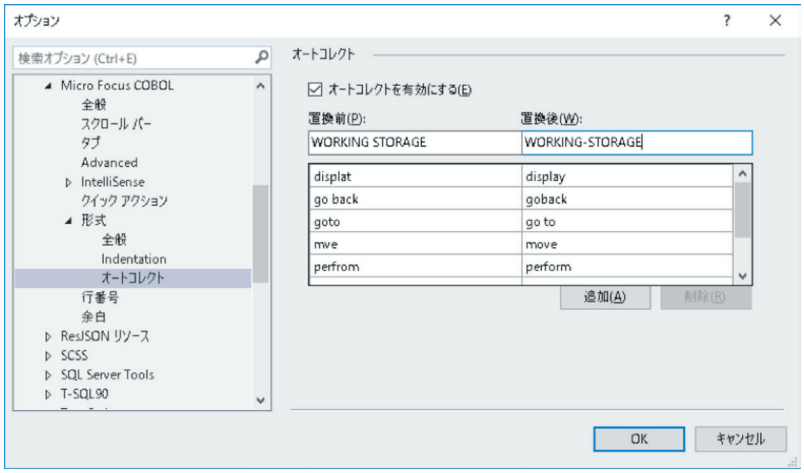
スニペット挿入イメージ：



③オートコレクト

オートコレクトは、事前に定義されたリストに基づいてスペルミス等を自動で補正する機能です。本機能も補正パターンをカスタマイズして追加することも可能ですので、プログラマーが誤入力しやすいパターンを予め登録しておけば、よりコーディングの生産性を向上させることができます。

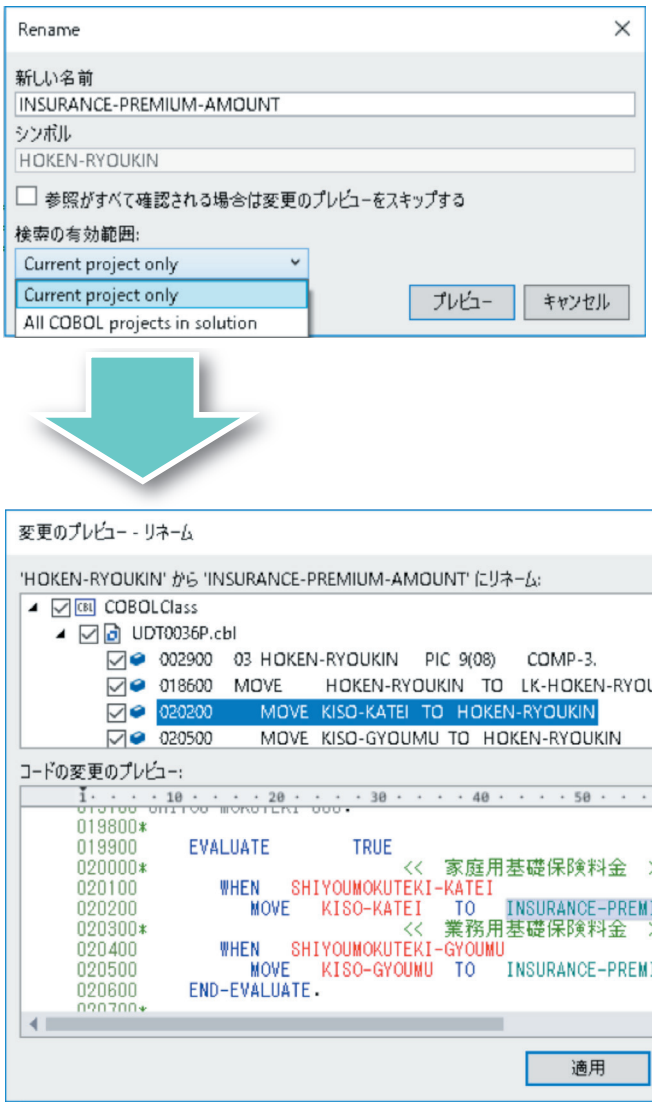
オートコレクトの設定画面イメージ：



④リファクター

変数名、ファイル名、段落名、節名等、ユーザー定義語をリネームする際は、定義位置のみならずそれらの参照箇所も併せて更新する必要があります。Visual COBOL 3.0J より実装されるリファクター機能を利用すればユーザー定義語の定義並びに参照名を指定されたスコープに基づいて自動で一括更新します。これにより、反映漏れ等を防止することができます。

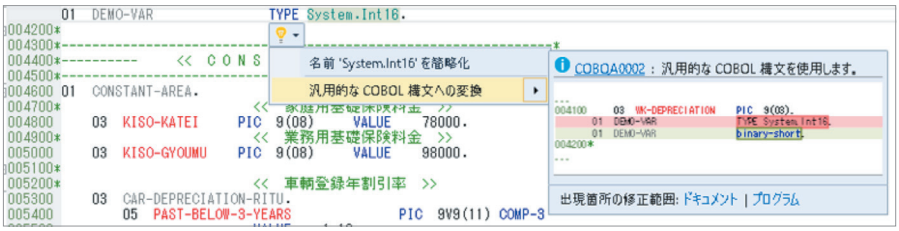
リファクターの実行イメージ：



⑤クイックアクション

クイックアクションは定められた開発作業の自動支援機能です。本機能は Microsoft が C# 等の開発用に Visual Studio 2015 より搭載し、Micro Focus も Visual Studio 2015 版の Visual COBOL をリリースするタイミングで COBOL 開発向けのクイックアクション機能を提供しました。本機能を利用することで例えば、任意のコードを COPY ファイルへ抽出する、.NET 形式の定義を対応する COBOL の予約語を使った定義があればそれに置き換える、Interface を実装する等の作業を自動化できます。

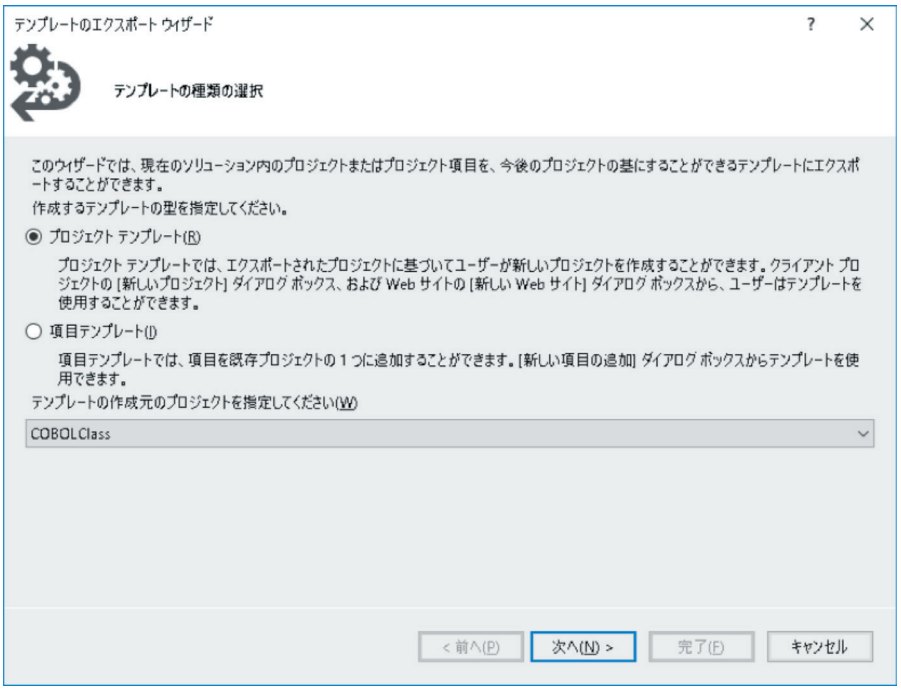
クイックアクション機能を利用して .NET 型の定義を COBOL 型に変換するイメージ：



⑥プロジェクトのカスタマイズ

Visual Studio では作成したプロジェクトをテンプレートとしてエクスポートし、開発者間で共有させることが可能です。Visual COBOL で作成したプロジェクトについても例外ではありません。コンパイラー指令、アセンブリの参照等のプロジェクト設定並びにソースコードの雛形をまとめてプロジェクトテンプレートとして切り出し、開発者間で共有させることが可能です。これによりプログラマーに属人化されたプロジェクトが作成されることを防ぎ、開発標準がより効率良く徹底させられることが見込めます。

プロジェクトテンプレートのエクスポートイメージ：



3. .NET での COBOL 再利用技法

メインフレームなどの基幹システムで長年運用されてきた COBOL ロジックは、安定した業務の実行が証明された企業資産です。これをそのまま .NET が提供する豊富なテクノロジの中で活用することができます。Web アプリケーション関連技術に目を向けますと、.NET では ASP.NET があり、ASP.NET WebForms、ASP.NET MVC に加え近年では ASP.NET SignalR のようなリアルタイム通信を実現する Web アプリケーション構築も可能になり周辺技術は一層充実化しています。このユーザビリティが増したアプリケーション内で品質が既に担保された企業固有のビジネスロジックを活用できるという点は換えがたいメリットです。Web アプリケーションのみならず、WPF のようなリッチなクライアントテクノロジーや、WCF のような分散アプリケーション技術の中からも活用することが可能です。本章では、COBOL ロジックを活用するに当たっての技法について、いくつかの側面から解説します。

3.1 既存 COBOL ロジックの有効活用

既存の COBOL ロジックが単独で完結したサブルーチンとして存在している場合、これをそのまま .NET のクラスライブラリにビルドできます。これにより、既存の COBOL ロジックプログラムを一行も書き換えることなく .NET 環境上で再利用することが可能です。利用する側は COBOL で書かれたということを意識することなく C# や VB.NET ベースで開発されたクラスライブラリと同じにようにインテリセンスを駆使して効率的に活用することができます。即ちプレゼンテーション部分は Web や WPF ベースのリッチクライアントへモダナイズしつつも、ビジネスロジックへの過去の投資を保護し且つリライトによる品質劣化リスクを回避するアプリケーションを構築することが可能となります。

一例として下記のような簡単な例題を考えます。

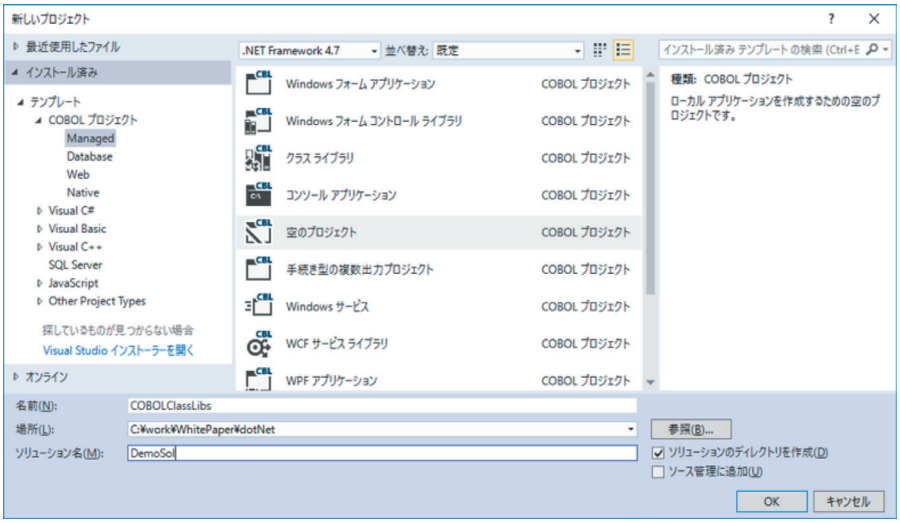
```
       FILE-CONTROL.
           SELECT CUST-MASTER ASSIGN TO CUSTF
                  ORGANIZATION INDEXED RECORD KEY FS-CUSTID
                  ACCESS MODE RANDOM.
       DATA            DIVISION.
       FILE            SECTION.
       FD CUST-MASTER.
       01 CUST-REC.
           05 FS-CustId      PIC 9(5)  COMP-3.
           05 FS-CustName    PIC X(30).
           05 FS-CustCompany PIC X(30).
           05 FS-CustEmail   PIC X(30).
       LINKAGE         SECTION.
       01 Cust-Param.
          03 CustId      PIC 9(5)  COMP-3.
          03 CustName    PIC X(30).
          03 CustCompany PIC X(30).
          03 CustEmail   PIC X(30).
       PROCEDURE       DIVISION
           USING Cust-Param.
       1.
           OPEN        I-O CUST-MASTER.
           MOVE        CustId     TO FS-CustId.
           READ        CUST-MASTER INVALID CONTINUE
                        END-READ.
           CLOSE       CUST-MASTER.
           MOVE        FS-CustName    TO CustName.
           MOVE        FS-CustCompany TO CustCompany.
           MOVE        FS-CustEmail  TO CustEmail.
           EXIT        PROGRAM.
```

この簡単な COBOL プログラムは、 第一パラメーターとして顧客番号を受け取り、 索引編成の顧客マスターファイルから該当するレコードを乱 READ してその顧客名、会社名、メールアドレスを返すというものです。ここではこのプログラムの記述は維持しつつも ASP.NET ベースの Web アプリケーションからシームレスに利用する具体的な手順をご紹介します。 本手順で紹介するアプリケーションは既存 COBOL ロジックの再利用手法の紹介を目的としており、マルチユーザーからの同時アクセス等が考慮されていません。 マルチユーザーからの同時アクセスを伴うアプリケーション開発技法については5章で紹介しています。

3.1.1 既存の COBOL プログラムを .NET クラスライブラリとしてビルド

1）テキストエディター等で上のプログラムをコピーし READCUST.cbl という名前で保存します。

2）Visual Studio で
[ファイル] メニュー > [新規作成] > [プロジェクト] を選択し、
[COBOL プロジェクト] > [Managed] > [空のプロジェクト]
テンプレートを選択します。



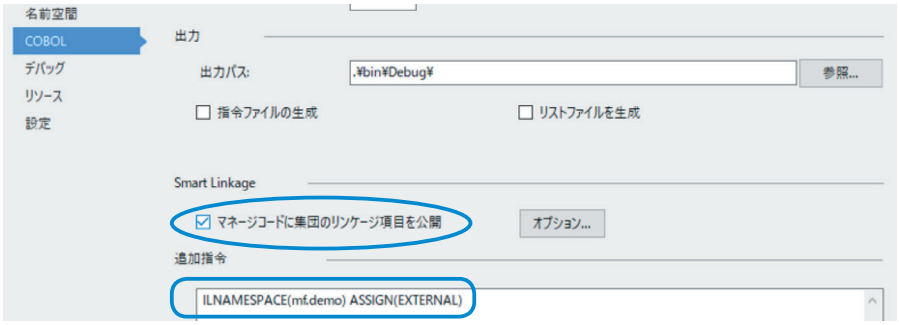
3）ソリューションエクスプローラーにてプロジェクトを右クリックから
[追加] > [既存の項目]
を選択します。

4）エクスプローラーより 1) で作成したプログラムを選択します。

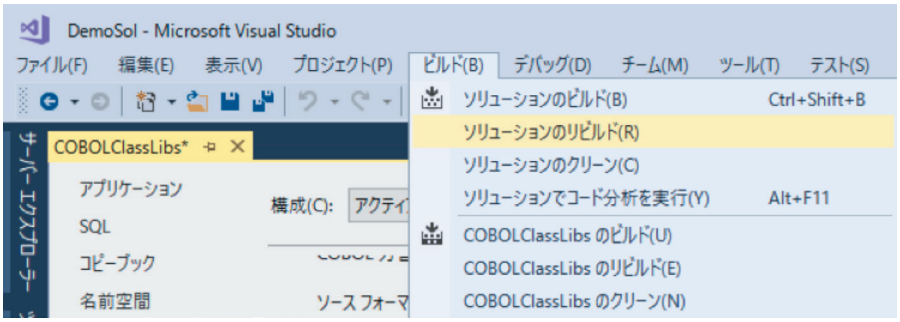
5）ソリューションエクスプローラーにて Properties をダブルクリックします。

6）アプリケーションページにおける出力の種類欄で「クラスライブラリ」を選択します。

7）COBOL ページにて [マネージコードに集団のリンケージ項目を公開] にチェックを入れ、[追加指令] 欄に「ILNAMESPACE(mf.demo) ASSIGN(EXTERNAL)」を指定します。



8）[ビルド] メニュー > [ソリューションのリビルド] を選択し COBOL プログラムを .NET クラスライブラリにビルドします。



メモ：SmartLinkage

[マネージコードに集団のリンケージ項目を公開] にチェックを入れると SmartLinkage 機能が有効になります。

COBOL はパック 10 進、固定長のバイナリデータ等 .NET にはないデータ型を利用します。そのため、.NET から COBOL のライブラリを利用するにはこの互換性を考慮する必要があります。SmartLinkage はこの言語間の差分を自動で吸収します。本機能を有効化しますとプログラムに対応する CLASS に加えて、LINKAGE SECTION の 01 レベルの変数の単位でそれらに応じた CLASS を自動で生成します。この CLASS には COBOL のパラメーターを .NET の基本型や String 等にマッピングした Public Property が実装されます。C# や VB.NET 等の .NET 言語側からはこの Public Property を通じて COBOL のクラスライブラリとデータ授受します。下表は本例で自動生成されたクラスと主なメソッド / プロパティの一覧となります。

クラス	メソッド / プロパティ
ReadCust	int ReadCust(CustParam value)
CustParam	uint Custid
CustParam	string Custname
CustParam	string Custcompany
CustParam	string Custemail

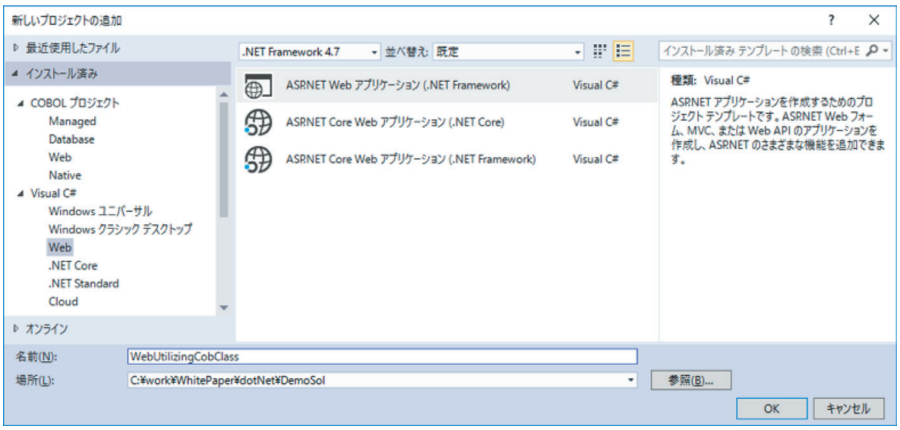
メモ : ILNAMESPACE 指令

ILNAMESPACE 指令は COBOL プログラムをビルドする際、指定された値の名前空間配下にクラスライブラリ生成します。これにより、COBOL プログラム中で名前空間の指定をすることなく任意のクラス構造を構築することができます。本例では、mf.demo という名前空間配下に ReadCust クラスを生成させています。また SmartLinkage により自動で生成された CustParam クラスも mf.demo 名前空間の下に生成されます。

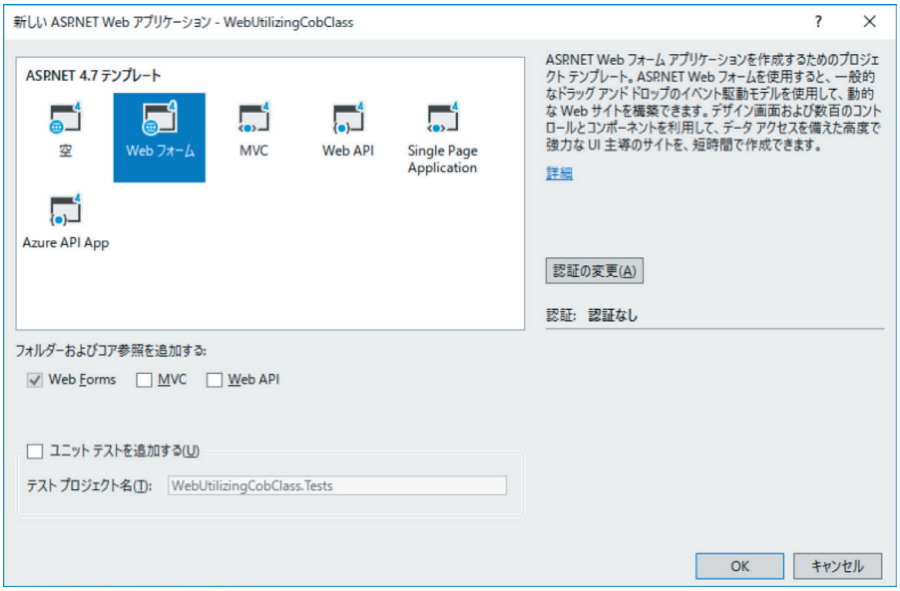
3.1.2 COBOL のクラスライブラリを呼び出す ASP.NET アプリケーションを作成

1）ソリューションエクスプローラーでソリューションを右クリックし、
[追加] > [新しいプロジェクト]
を選択します。

2）[Visual C#] > [Web] > [ASP.NET Web アプリケーション (.NET Framework)]
を選択します。

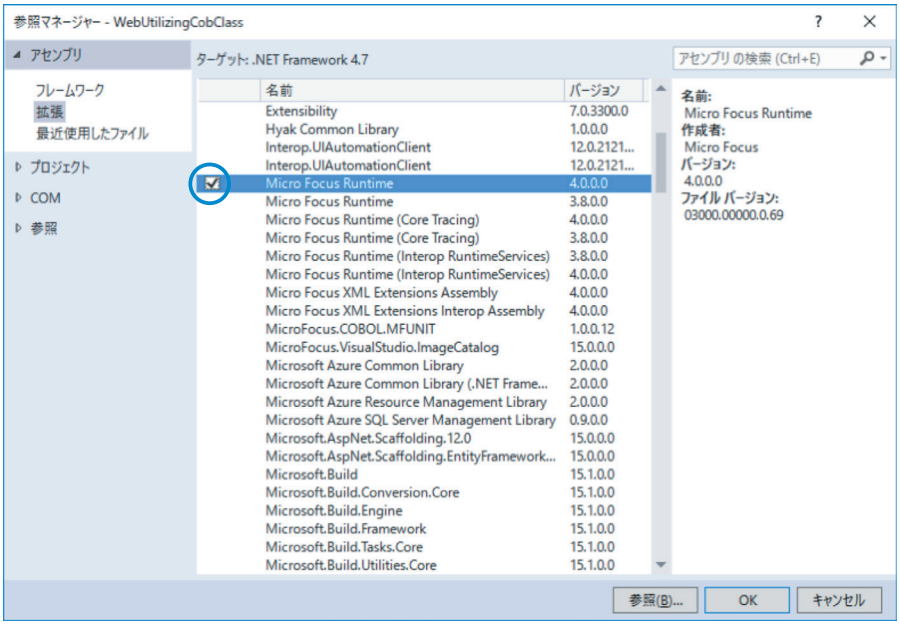


3）次に表示される画面にて [Web フォーム] を選択します。

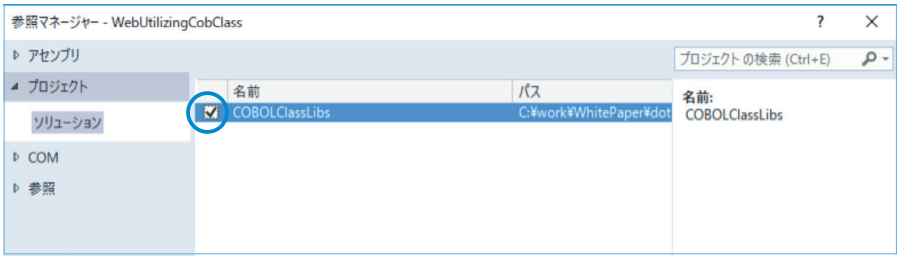


4）ソリューションエクスプローラーにて作成した Web プロジェクト配下の
[参照設定] を右クリックし [参照の追加] を選択します。

5）アセンブリページ配下の拡張ページを選択し、Micro Focus Runtime バージョン 4.0.0.0 を選択します。



6）ロジェクトページを選択し COBOL のクラスライブラリを作成したプロジェクトにチェックを入れます。



7）About.aspx のテンプレートの一部を本例のロジックに対応したかたちに書き換えます。

変更前 :

```
                    :
<h3>Your application description page.</h3>
<p>Use this area to provide additional information.</p>
                    :
```

変更後：

```
                        :
    <h3>顧客 ID を入力してください :</h3>
    <asp:TextBox ID="txtID" runat="server" />
    <asp:Button ID="btnGo" runat="server" Text=" 検索 " OnClick="btnGoClicked" />
    <asp:Panel ID="pnlRes" runat="server" Visible="false">
    <ol class="round">
        <li class="one">
            <h5>CustName</h5>
            <asp:Label ID="lblName" runat="server" />
        </li>
        <li class="two">
            <h5>CustCompany</h5>
            <asp:Label ID="lblCompany" runat="server" />
        </li>
        <li class="three">
            <h5>CustEmail</h5>
            <asp:Label ID="lblEmail" runat="server" />
        </li>
    </ol>
    </asp:Panel>
                        :
```

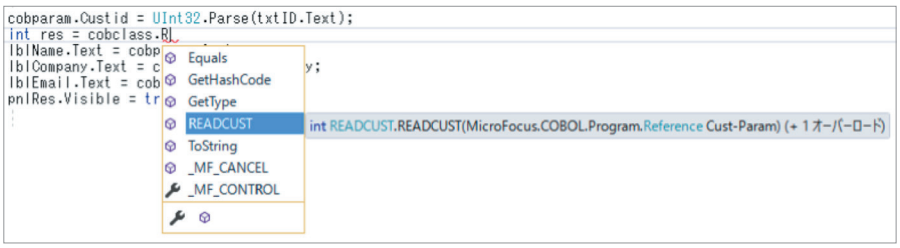
8）検索ボタン押下に対するイベントハンドラを作成し、COBOL から作成したクラスを利用し顧客データを取得するロジックを埋め込みます。

About.aspx.cs に以下のイベントハンドラを埋め込みます。

```
                        :
using mf.demo;
                        :
        protected void btnGoClicked(object sender, EventArgs e)
        {
            READCUST cobclass = new READCUST();
            CustParam cobparam = new CustParam();


            cobparam.Custid = UInt32.Parse(txtID.Text);
            int res = cobclass.READCUST(cobparam);
            lblName.Text = cobparam.Custname;
            lblCompany.Text = cobparam.Custcompany;
            lblEmail.Text = cobparam.Custemail;
            pnlRes.Visible = true;
        }
                        :
```

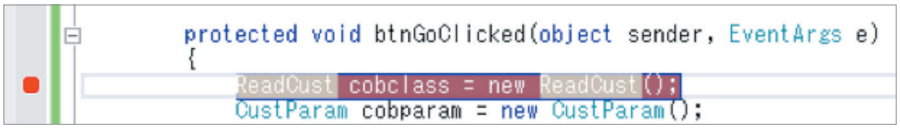
COBOL プログラムから生成されたクラスライブラリは通常のクラスライブラリと同様にシンボルを生成するため、下図のようにインテリセンスを駆使して開発することが可能です。



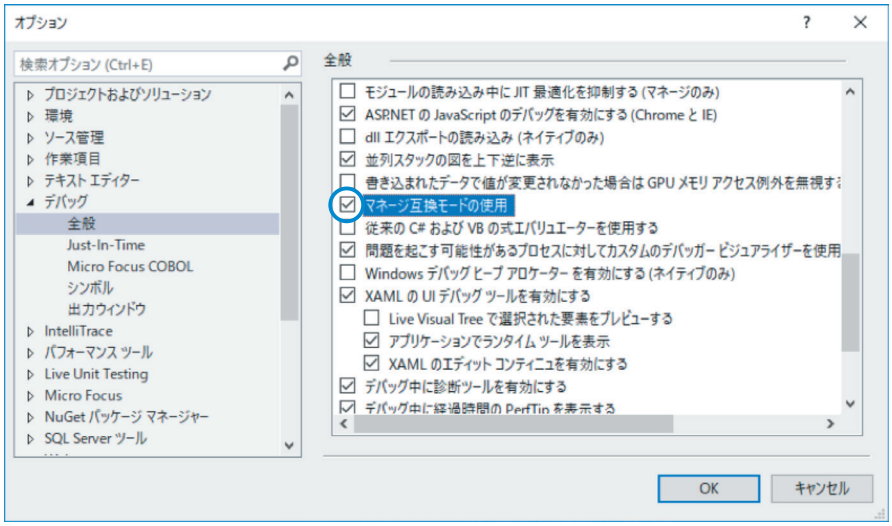
9）Web アプリケーションの構成ファイル Web.config に COBOL ファイル CUST.dat のパス情報を指定します。
以下のようなコードを Web.config ファイル中の configuration タグの直下に追加します。

```
<configuration>
  <configSections>
    <sectionGroup name="MicroFocus.COBOL.Application">
      <section name="Switches" type="System.Configuration.NameValueSectionHandler" />
      <section name="Environment" type="System.Configuration.NameValueSectionHandler" />
    </sectionGroup>
    <sectionGroup name="MicroFocus.COBOL.Runtime">
      <section name="Tunables" type="System.Configuration.NameValueSectionHandler" />
      <section name="Switches" type="System.Configuration.NameValueSectionHandler" />
    </sectionGroup>
  </configSections>
  <MicroFocus.COBOL.Application>
    <Switches />
    <Environment>
      <add key="CUSTF" value="< 予めテストデータが登録されている CUST.dat が格納されるパス >¥CUST.dat" />
    </Environment>
  </MicroFocus.COBOL.Application>
                :
```

10）イベントハンドラの一行目にブレークポイントを指定します。

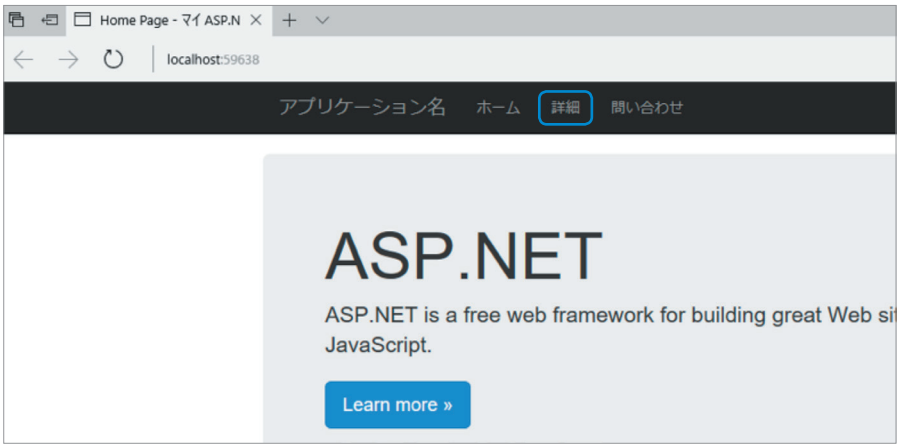


11）[ツール] > [オプション] を選択しオプションウィンドウを起動し、
[デバッグ] > [全般]
で表示される画面にて [マネージ互換モードの使用] にチェックを入れます。



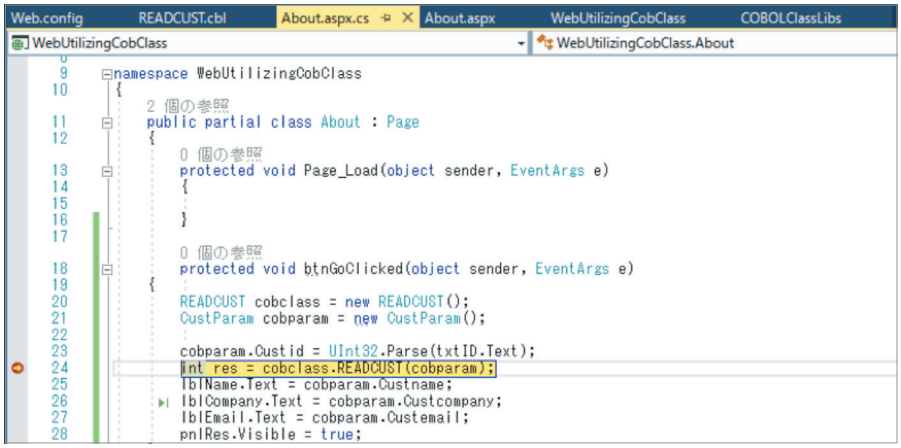
12）F5 キーを打鍵し Web アプリケーションをデバッグ実行します。

① 開始ページにて [詳細] をクリックし About.aspx へ遷移します。

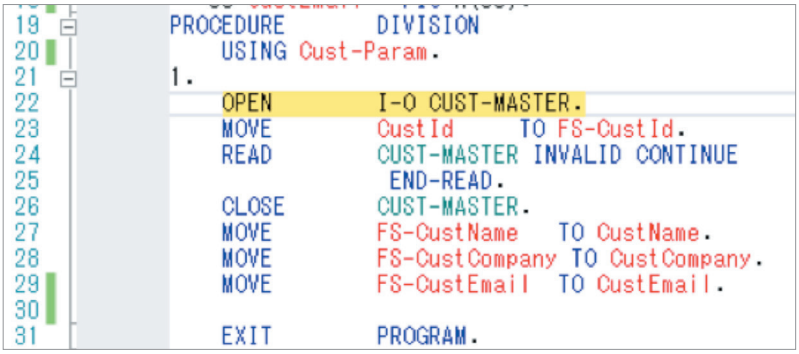


② CUST.dat に登録されている ID を入力の上「検索」ボタンを押下します。

前の手順で追加したブレークポイント位置でデバッガーが一時停止していますので、ここで F11 を打鍵し、ステップインします。



COBOL プログラム中にステップが移動します。C# と COBOL の混合デバッグができることが確認できます。



③ 最後までステップを進め、意図した通りの結果が得られることを確認します。

3.2 COBOL ロジックの .NET 向け最適化

前章では、既存 COBOL ロジックをそのまま維持しつつ再利用するという要件下の手順を見ました。このように単独のサブルーチンをメソッドとして呼び出すのは、COBOL 側から見ればもっとも簡単な方法に見えますが、呼び出し側の .NET アプリケーションから見れば必ずしも最適な利用方法ではありません。それは、この方法が .NET Framework が提供する豊かなオブジェクトモデルを何も使用していないことによります。例えば .NET らしいプログラミングでは以下のような設計がなされています。

● オブジェクトプロパティとしての外部参照
3.1 の例では、プログラムにパラメーターとして値を渡し処理結果をパラメーターとして受け取っています。このように呼び出しの際にのみローカルに使用されるパラメーター領域ではなく、オブジェクトのライフサイクルの間永続的に存在し、外部から参照・更新が可能なデータを、プロパティとして持つことができます。

● コンストラクターによるオブジェクト初期化
オブジェクトが生成される際に行われる初期化処理（ファイルの OPEN、領域の初期値設定など）は、コンストラクターと呼ばれる特殊なメソッドに集約させることができます。こうしておくことによって呼び出し側 .NET プログラミング言語は new 操作を行うことで、COBOL 側の初期化処理を自動的に起動できるようになります。

● 例外スローによるエラー通知
COBOL でのサブルーチン呼び出しでは、呼び出し側にエラーが発生したことを通知するためには、パラメーターにエラーコードを設定して返す方法が一般的です。一方 .NET では、呼び出したメソッドがスローする例外をキャッチして処理する方法が一般的です。.NET の世界で COBOL ロジックを活用する際に、COBOL プログラミングの常識を押し通すことは、呼び出し側の .NET プログラマーにとっては負担になります。

以下、このような .NET らしいプログラミングを、COBOL ロジックの再利用で活用する方法について、チュートリアルで見てみます。

3.1 で使用したのと同じ顧客マスターファイルのアクセスを再び例に取ります。今度は、ファイルのレコードを先頭から順読みしてみたいと思います。3.1 では、ひとつの顧客番号に対するレコードを 1 件だけ返す例でしたので、単独のメソッド呼び出しで実現できました。今度は全件のレコードを読みますので、COBOL だけで書くプログラムとしては以下のようになります。

```
OPEN        INPUT       CUST-MASTER.
PERFORM     UNTIL       EXIT
    READ      CUST-MASTER AT END EXIT PERFORM
              END-READ
    DISPLAY   "顧客番号 : " FS-CustId
    DISPLAY   "顧客名   : " FS-CustName
    DISPLAY   "会社名   : " FS-CustCompany
    DISPLAY   "Email    : " FS-CustEmail
END-PERFORM.
CLOSE       CUST-MASTER.
```

このロジックを .NET クラス化するにあたって以下の方針は妥当なものといえます。

●ファイル定義は 3.1 と同様にメソッドの外に出し、クラスの内部プロパティとする。
●顧客番号、顧客名、会社名、Email は、それぞれクラスの外部プロパティとしてエクスポートする。
●ファイルの OPEN はクラスのコンストラクターの中で行う。
● "ReadNext" というメソッドを用意し、これが呼ばれるごとに 1 件のレコードを順読みする。
● "ReadNext" メソッドは、ファイルの末尾に達したら、ファイルを CLOSE してから、例外をスローする。

そこでこの方針に従ってクラスを実装して呼び出してみます。

1）Visual Studio で [ファイル]>[新規作成]>[プロジェクト] を選択し、COBOL プロジェクトの [Managed]>[クラスライブラリ] テンプレートを選択します。

2）クラス定義ソースのテンプレートが自動生成されます。必要に応じてクラス名は書き換えますが、この例題では「ReadCust2」とします。

3）ソリューションエクスプローラーにてプロパティをダブルクリックします。

4）COBOL ページにおける追加指令欄で「ILNAMESPACE(mf.demo) ASSIGN(EXTERNAL)」を指定します。

5）プログラムを編集します。まず、ファイル定義をメソッドの外に出し、コンストラクタを表す “new” というメソッド名でファイルの OPEN を実装します。索引ファイルのアクセスメソッドは SEQUENTIAL に変わります。これでプログラムは下記のようになります。

```
    FILE-CONTROL.
        SELECT CUST-MASTER ASSIGN TO CUSTF
               ORGANIZATION INDEXED RECORD KEY FS-CUSTID
               ACCESS MODE SEQUENTIAL.
    data division.
    FILE            SECTION.
    FD CUST-MASTER.
    01 CUST-REC.
        05 FS-CustId       PIC 9(5)  COMP-3.
        05 FS-CustName     PIC X(30).
        05 FS-CustCompany  PIC X(30).
         05 FS-CustEmail   PIC X(30).
    working-storage section.


    method-id new.
    procedure division.
        OPEN INPUT CUST-MASTER.
        goback.
    end method.


    end object.
```

6）クラスの外部プロパティはメソッドの外側に WORKING-STORAGE SECTION として記述します。外部プロパティの名前は、COBOL の PROPERTY AS 句で明示的に指定します。

```
    WORKING-STORAGE SECTION.
    01 CustId      Binary-long PROPERTY AS "CustID".
    01 CustName    String PROPERTY AS "CustName".
    01 CustCompany String PROPERTY AS "CustCompany".
    01 CustEmail   String PROPERTY AS "CustEmail".
```

7）ReadNext メソッドを実装します。READ NEXT 文を実行し、レコードが存在すればレコード中の各データ項目を外部プロパティに設定します。AT END 条件が発生したら例外をスローします。
例外は System.Exception オブジェクトを、RAISE 文を使用してスローすることができます。RAISE 文は ISO2002 国際標準で COBOL 言語に追加された COBOL 文法です。
メソッドは以下のとおりとなります。

```
method-id. ReadNext.
procedure division.
    READ CUST-MASTER NEXT
     AT END
      CLOSE CUST-MASTER
      RAISE new System.Exception(" レコード終了 ")
     NOT AT END
      MOVE FS-CustId     TO CustId
      MOVE FS-CustName   TO CustName
      MOVE FS-CustCompany TO CustCompany
      MOVE FS-CustEmail  TO CustEmail
    END-READ.
    goback.
end method.
```

8）プログラムは以下の通りとなりました。

```
 class-id ReadCust2.

 FILE-CONTROL.
    SELECT CUST-MASTER ASSIGN TO CUSTF
           ORGANIZATION INDEXED RECORD KEY FS-CUSTID
           ACCESS MODE SEQUENTIAL.
data division.
FILE            SECTION.
FD CUST-MASTER.
01 CUST-REC.
    05 FS-CustId      PIC 9(5)  COMP-3.
    05 FS-CustName    PIC X(30).
    05 FS-CustCompany PIC X(30).
    05 FS-CustEmail   PIC X(30).
working-storage section.
01 CustId      Binary-long PROPERTY AS "CustID".
01 CustName    String PROPERTY AS "CustName".
01 CustCompany String PROPERTY AS "CustCompany".
01 CustEmail   String PROPERTY AS "CustEmail".
method-id new.
procedure division.
    OPEN INPUT CUST-MASTER.
    goback.
end method.
```

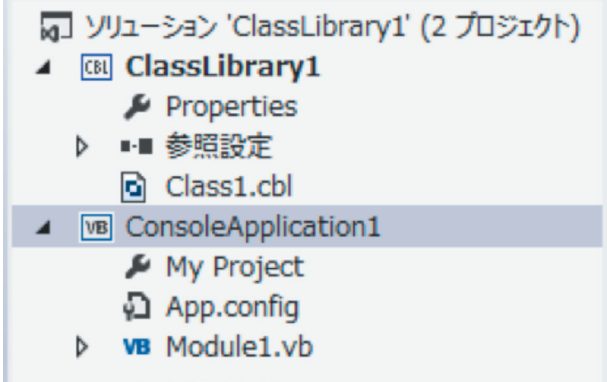
```
 method-id ReadNext.
procedure division.
    READ CUST-MASTER NEXT
     AT END
      RAISE new System.Exception(" レコード終了 ")
     NOT AT END
      MOVE FS-CustId     TO CustId
      MOVE FS-CustName   TO CustName
      MOVE FS-CustCompany TO CustCompany
      MOVE FS-CustEmail  TO CustEmail
    END-READ.
    goback.
end method.

end class.
```

9）アプリケーションをビルドします。これで再利用する COBOL ロジックのクラス化は完了です。

10）次にこのクラスを外部から利用してみます。[ファイル] > [追加] > [新しいプロジェクト] を選択し、例えば VB .NET の [Windows] > [コンソールアプリケーション] テンプレートを選択します。

これで COBOL と VB .NET が混在するソリューションができあがりました。



11）VB .NET から COBOL を呼び出すための参照設定を追加します。VB .NET の ConsoleApplication1 を右クリックし [参照の追加] を選択します。

12）さらに [ソリューション] 配下の [プロジェクト] を選択し、COBOL の ClassLibrary1 を選択します。

13）さらに、[アセンブリ] 配下の [拡張] を選択し、Micro Focus Runtime バージョン 4.0.0.0 を選択して [OK] をクリックします。

14）以下のように COBOL のクラスメソッドを呼び出すコードを書きます。

```
Imports mf.demo

Module Module1
    Sub Main()
        Dim CobClass As ReadCust2 = New ReadCust2
        Dim counter As Integer = 0
        Do While True
            Try
                CobClass.ReadNext()
                counter = counter + 1
                Console.WriteLine(counter & " 件目 ")
                Console.WriteLine(CobClass.CustName)
                Console.WriteLine(CobClass.CustCompany)
                Console.WriteLine("**************************************")
            Catch
                Console.WriteLine(counter & " 件のデータ読み込みが完了しました。")
                Exit Do
            End Try
        Loop
    End Sub
End Module
```

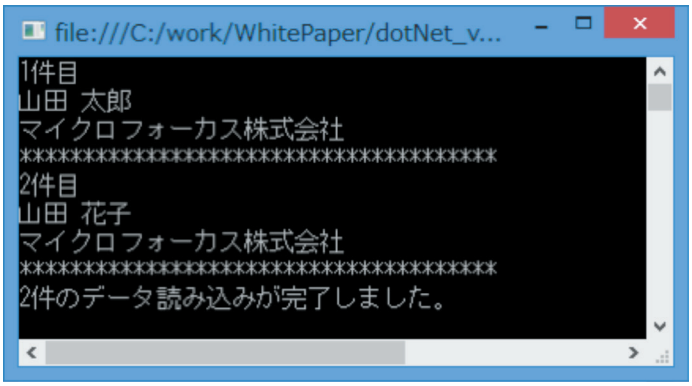
15）アプリケーションの実行環境にファイルのパス情報を指定します。App.config に以下のコードを追加します。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <!--The following code declares a section group for application configuration -->
    <sectionGroup name="MicroFocus.COBOL.Application">
      <section name="Switches" type="System.Configuration.NameValueSectionHandler" />
      <section name="Environment" type="System.Configuration.NameValueSectionHandler" />
    </sectionGroup>
    <!--The following code declares a section group for run-time configuration -->
    <sectionGroup name="MicroFocus.COBOL.Runtime">
      <section name="Tunables" type="System.Configuration.NameValueSectionHandler" />
      <section name="Switches" type="System.Configuration.NameValueSectionHandler" />
    </sectionGroup>
  </configSections>
  <MicroFocus.COBOL.Application>
    <Switches />
    <Environment>
      <add key="CUSTF" value="＜COBOL ファイルまでのパス＞¥CUST.dat" />
    </Environment>
  </MicroFocus.COBOL.Application>
                                :
```

16）VB .NET のプロジェクトをソリューションのスタートアッププロジェクトに設定し、ソリューションをビルドしてデバッグ実行します。

17）ステップ実行すると、ファイルの終わりに達して COBOL の RAISE 文が実行されると、呼び出した VB .NET 側の Catch ブロックに制御が移ることが確認できます。

18）再利用された COBOL プログラムが呼び出され、以下のように VB .NET 側から結果が表示されます。（レコードが 2 件しか存在しない場合の実行例です）

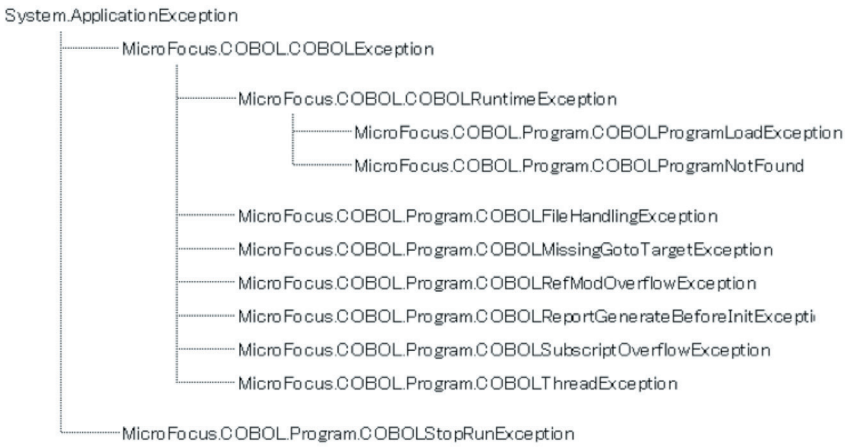


3.3 COBOL ロジック中で発生する例外のハンドリング

.NET アプリケーションは一般的に何等かの実行時エラーにより影響がシステム全体へ波及し防ぐべく例外のハンドリング処理をアプリケーションに組み込みます。COBOL で書かれた .NET アプリケーションに関しても然りであり、COBOL で書かれたアプリケーション内で何等かの問題が発生した場合、例外のハンドリングが考慮されていなければアプリケーション全体のプロセスに影響が及びます。他の .NET アプリケーションでも同様ですが、最悪の場合システム自体が停止することも想定されます。

Visual COBOL は下図のような例外クラスを用意しています。.NET Framework 上で動作する Visual COBOL のランタイムは COBOL アプリケーションに何等かの問題が発生すると問題の内容に合わせて図中の例外をスローします。COBOL から生成された .NET クラスライブラリを利用するアプリケーションはこれらの例外クラスを使って例外処理を実装することができます。

図3- 1 COBOL ロジックよりスローされる主な例外一覧



以下は、3.1 で作成した ASP.NET アプリケーションに COBOL ロジックで発生する例外をハンドリングする処理を加えた例となります。本例のような IIS Server 上で動作させるプログラムやサブプログラムとして使用されるプログラムでは STOP RUN 文は記述しません。従いまして、COBOL プログラム中で発生する実行時例外は MicroFocus.COBOL.COBOLException で全て拾うことができます。ここでは、先ほどの COBOL ロジックを呼び出すプログラムにこの例外クラスを指定した catch ブロックを追加し、COBOL ロジックで発生する例外を考慮したアプリケーションに発展させています。

```
About.aspx.cs:
      :
using mf.demo;
      :
        protected void btnGoClicked(object sender, EventArgs e)
        {
            try
            {
                ReadCust cobclass = new ReadCust();
                CustParam cobparam = new CustParam();

                cobparam.Custid = UInt32.Parse(txtID.Text);
                int res = cobclass.ReadCust(cobparam);
                lblName.Text = cobparam.Custname;
                lblCompany.Text = cobparam.Custcompany;
                lblEmail.Text = cobparam.Custemail;
                pnlRes.Visible = true;
            }
            catch (MicroFocus.COBOL.COBOLException CE)
            {
                string errmsg = "◆ COBOL クラスの実行でエラーが発生しました。"
                              + "システム管理者に連絡してください。"
                              + "<br />Error Code: " + CE.ErrorNumber.ToString()
                              + "<br />Error Message: " + CE.ErrorString;
                lblErr.Visible = true;
                lblErr.Text = errmsg;
            }
        }
      :
```

4. .NET でのバッチ処理

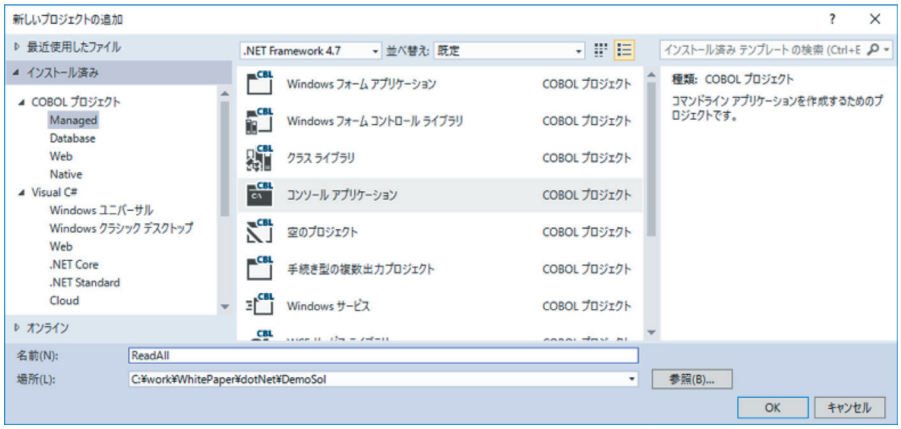
前章では、抽出されたビジネスロジックをクラス化して .NET の中で再利用する方法について見ました。この方法はオンライン処理の再利用で有効です。一方、COBOL で書かれたバッチプログラムは、ひとつの完結した COBOL プログラムが大量のレコード処理を行います。このようなプログラムはクラス化せずに単独のアセンブリとして .NET で運用することができます。

以下にそのような方法の手順を見ます。ここでは例として 3.2 で使用した顧客マスターファイルの全件順読みを行うバッチプログラムを例にとることにします。すなわち、以下のソースファイル READALL.cbl が既に存在するところからスタートします。

```
       FILE-CONTROL.
            SELECT CUST-MASTER ASSIGN TO CUSTF
                   ORGANIZATION INDEXED RECORD KEY FS-CUSTID.
       DATA            DIVISION.
       FILE            SECTION.
       FD CUST-MASTER.
       01 CUST-REC.
           05 FS-CustId      PIC 9(5)  COMP-3.
           05 FS-CustName    PIC X(30).
           05 FS-CustCompany PIC X(30).
           05 FS-CustEmail   PIC X(30).
       PROCEDURE       DIVISION.
       1.
           OPEN        INPUT       CUST-MASTER.
           PERFORM     UNTIL      EXIT
                       READ CUST-MASTER
                            AT END EXIT PERFORM
                       END-READ
             DISPLAY   "顧客番号 : " FS-CustId
             DISPLAY   "顧客名   : " FS-CustName
             DISPLAY   "会社名   : " FS-CustCompany
             DISPLAY   "Email    : " FS-CustEmail
           END-PERFORM.
           CLOSE       CUST-MASTER.
           STOP        RUN.
```

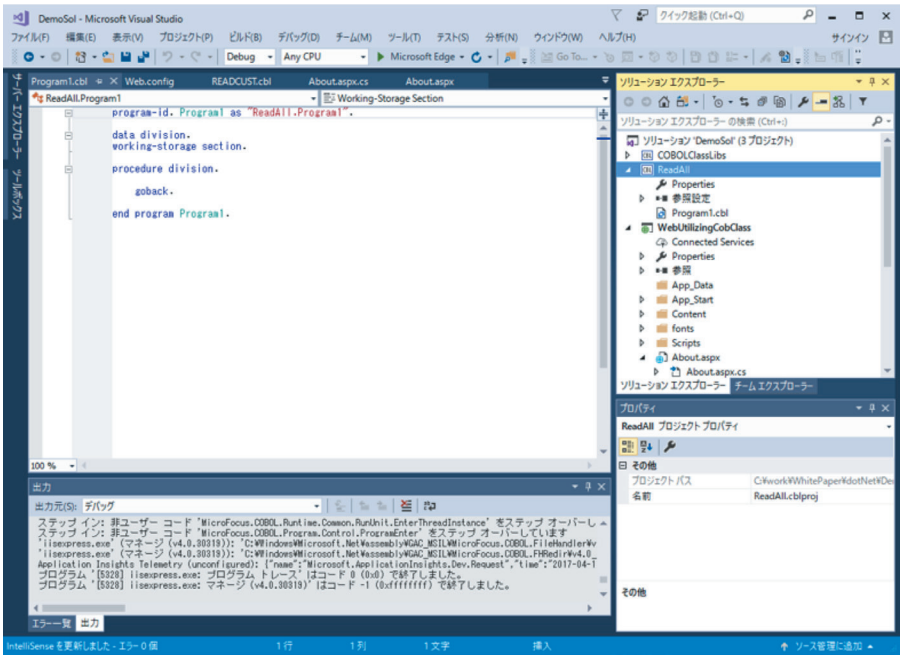
この COBOL ソースファイルを 1 行も修正することなしにそのまま .NET Framework 上のマネージコードとして実行することができます。

1) Visual Studio で [ファイル] > [プロジェクトの新規作成] を選択し、
[COBOL プロジェクト] > [Managed]
配下の [コンソールアプリケーション] を選択します。



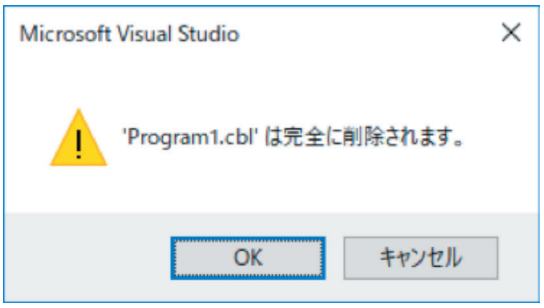
2) プロジェクト名として ReadAll を入力し、[OK] をクリックします。

3) Program1.cbl というコンソールアプリケーションのテンプレートが自動生成されます。



ここではこのテンプレートは使用しないので削除します。ソリューションエクスプローラー上のツリーで、Program1.cbl を右クリックして [削除] を選択します。

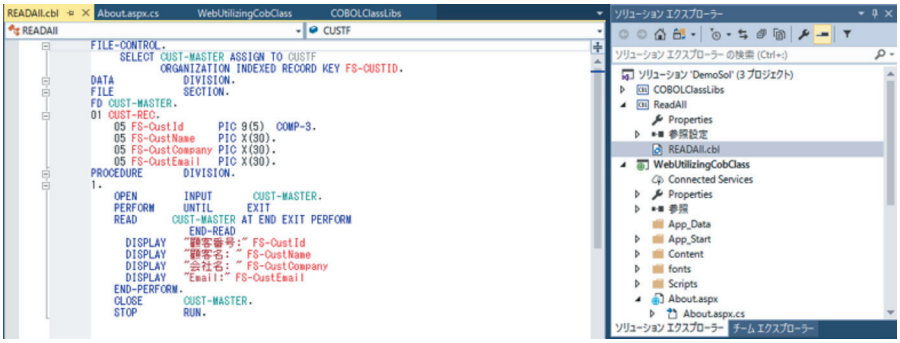
4）以下の確認ダイアログが出たら [OK] をクリックします。



5）ソリューションエクスプローラー上のツリーで、ReadAll プロジェクトを右クリックして [追加] > [既存の項目] を選択します。

6）ファイルダイアログで既存の READAll.cbl ソースファイルを選択し、[追加] をクリックします。

7）以下のようにプロジェクトに READAll.cbl が追加され、エディターで開くことができます。



8）ソリューションエクスプローラーにて Properties をダブルクリックします。

9）COBOL ページにおける追加指令欄で「ASSIGN(EXTERNAL)」を指定します。

10）COBOL ファイルのパスを指定します。ソリューションエクスプローラーにてプロジェクトを右クリックし
[追加] > [新しい項目] > [アプリケーション構成ファイル]
を指定します。

11）ソリューションエクスプローラーにて追加された App.config を右クリックし「編集」を指定します。

12）以下の要領でファイルのパス情報を指定します。

名前欄：　CUSTF
値欄：　ファイル名を含めたパス

13）プロジェクトをビルドし、デバッガーでステップインすると、手続き部の先頭からバッチプログラムをステップ実行することができます。

5. マルチユーザーアプリケーションの開発

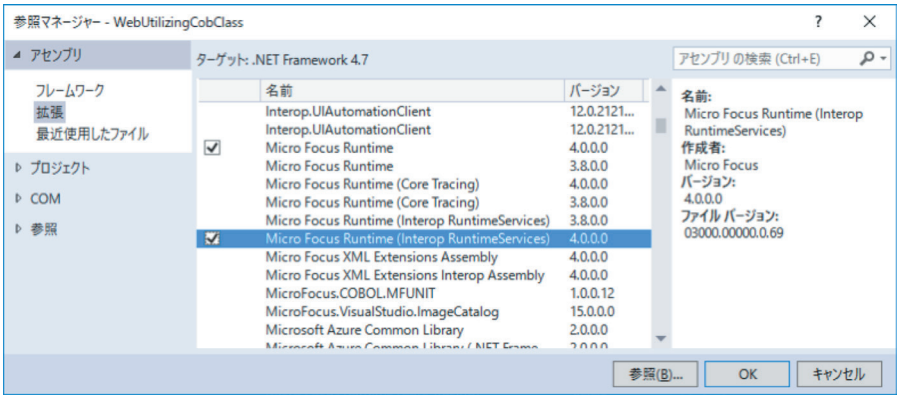
ASP.NET のような Web Application を開発する場合、マルチユーザーからの同時アクセスを考慮する必要があります。ASP.NET アプリケーションはそれぞれのユーザーに対してセッションを持たせる等して処理の一貫性を管理しますが、COBOL アプリケーションの多くはそのような点は考慮されていません。そのため、各スレッドがアドレス空間を共有するデフォルトの状態で COBOL アプリケーションを ASP.NET 内で運用するとセッション毎によるメモリの分離が実現できません。Visual COBOL はこのような条件下での COBOL アプリケーションの運用を想定して RunUnit という機能を提供しています。この RunUnit 機能を活用することで、メモリやファイルロックテーブルをセッション毎に分離することが可能です。これにより、シングルユーザーによる利用を想定して開発された COBOL プログラムであっても ASP.NET のようなマルチユーザーアプリケーション上に容易にマイグレーションできるようになります。尚、RunUnit を使ったマルチユーザーアクセス対応は、COBOL アプリケーションの書き換えを伴いません。

上述したような理由から RunUnit をセッションの開始／終了と連携させることでマルチユーザーアプリケーション処理の一貫性を保てます。ここでは、3.1 で作成した ASP.NET アプリケーションにこの RunUnit のコンセプトを反映させ、マルチユーザーによる同時アクセスを意識したアプリケーションに発展させます。

1）3.1 で作成した ASP.NET アプリケーションを含むソリューションを開きます。

2）ソリューションエクスプローラーにて ASP.NET アプリケーション下の「参照設定」を右クリックし「参照の追加」を選択します。

3）アセンブリページ配下の拡張ページを選択し、Micro Focus Runtime(Interop RuntimeServices) バージョン 4.0.0.0 を追加で選択します。



4）ASP.NET アプリケーションのプロジェクト中にある Global.asax.cs に以下のセッション起動時のイベントハンドラ及びセッション終了時のイベントハンドラを追加します。セッション起動時のイベントハンドラ Session_OnStart には RunUnit の生成処理を実装しセッション変数に登録します。セッション終了時のイベントハンドラ Session_OnEnd には RunUnit を破棄する処理を実装しています。

```
                    :

    public void Session_OnStart()
    {
        // セッションの開始に合わせて RunUnit を起動
        MicroFocus.COBOL.RuntimeServices.RunUnit CobRunUnit =
             new MicroFocus.COBOL.RuntimeServices.RunUnit();
        Session["CobRunUnit"] = CobRunUnit;
    }


    public void Session_OnEnd()
    {
        // セッションの終了に伴い、RunUnit を破棄
        MicroFocus.COBOL.RuntimeServices.RunUnit CobRunUnit =
            (MicroFocus.COBOL.RuntimeServices.RunUnit)Session["CobRunUnit"];
        if (CobRunUnit != null)
        {
            CobRunUnit.StopRun();
        }
    }
                    :
```

5）Default.aspx.cs 中に記述したボタン押下に対するイベントハンドラを以下のように書き換えます。セッション開始時にセッション変数として登録した RunUnit を使って COBOL クラスを実行するためのロジックを追加しています。RunUnit が何等かの理由で存在しない場合は、新たに生成し直します。RunUnit 自体に何等かの問題が発生する場合は、System.ApplicationException の例外がスローされます。そのため、COBOL ロジックの実行時エラーの例外処理に加えてこの例外処理も追加しています。

```
    protected void btnGoClicked(object sender, EventArgs e)
    {
        ReadCust cobclass = new ReadCust();
        CustParam cobparam = new CustParam();


        // セッション開始時にセッション変数として登録した RunUnit を取得
        MicroFocus.COBOL.RuntimeServices.RunUnit CobRunUnit =
                        (MicroFocus.COBOL.RuntimeServices.RunUnit)Session["CobRunUnit"];
        if (CobRunUnit == null)
        {
            // 何等かの理由で RunUnit がセッションに存在しない場合は、RunUnit を新たに起動
            CobRunUnit = new MicroFocus.COBOL.RuntimeServices.RunUnit();
        }

        try
        {
```

```
            //cobclass は CobRunUnit が参照する RunUnit で処理するよう指定
            CobRunUnit.Add(cobclass);

            cobparam.Custid = UInt32.Parse(txtID.Text);

            //cobclass.ReadCust メソッドを CobRunUnit が参照する RunUnit で処理
            int res = cobclass.ReadCust(cobparam);

            lblName.Text = cobparam.Custname;
            lblCompany.Text = cobparam.Custcompany;
            lblEmail.Text = cobparam.Custemail;
            pnlRes.Visible = true;
        }
        catch (MicroFocus.COBOL.COBOLException CE)
        {

            string errmsg = "◆ COBOL クラスの実行でエラーが発生しました。"
                          + "システム管理者に連絡してください。"
                          + "<br />Error Code: " + CE.ErrorNumber.ToString()
                          +  "<br />Error Message: " + CE.ErrorString;
            lblErr.Visible = true;
            lblErr.Text = errmsg;

            //COBOL プログラムに実行時エラーが発生した場合は、RunUnit を破棄
            CobRunUnit.StopRun();
        }
            //Run Unit に何等かのエラーが発生した場合は System.ApplicationException 例外がスローされます。
        catch (System.ApplicationException AE)
        {
            lblErr.Visible = true;
            lblErr.Text = "◆ COBOL ランタイムに何等かの問題が発生しました。"
                        + "システム管理者に連絡してください。<br />"
                        + AE.StackTrace;

            //RunUnit が異常終了しているため、RunUnit を破棄
            CobRunUnit.StopRun();
        }
    }
```

6. COBOL から .NET 資源を活用する

これまでは、COBOL ロジックを .NET プログラミングの中で活用する方法について説明してきました。ここからは反対に COBOL ロジックの中から .NET の世界に存在している豊富な資源にアクセスする方法について説明します。

6.1 基本の例題

まず、基本的な事柄を理解するために、最も簡単な .NET Framework のクラスライブラリを利用する COBOL プログラムを見てみます。

```
namespace ConsoleApplication1
{
   class Rand
   {
       static void Main(string[] args)
       {
           System.Random randObj = new System.Random();
           for (int i = 0; i < 10; i++)
               Console.WriteLine(randObj.Next());
       }
   }
}
```

このコードは、System.Random クラスのオブジェクトをインスタンス化し、これに対して Next() メソッドを 10 回呼び出し、その結果をコンソールに書き出しています。これと同じことを COBOL で行うと以下の通りとなります。

```
IDENTIFICATION DIVISION.
PROGRAM-ID.      RAND.
DATA            DIVISION.
WORKING-STORAGE SECTION.
01 RANDOBJ    TYPE  System.Random.
PROCEDURE       DIVISION.
           SET RANDOBJ TO new System.Random()
           PERFORM 10 TIMES
               DISPLAY RANDOBJ::Next()
           END-PERFORM.
           STOP RUN.
```

この COBOL コードの意味は明瞭ですが、以下にポイントを整理します。

1）データ項目の宣言において、TYPE 句を使用して、オブジェクト参照用途の項目を宣言します。

2）「new クラス名」でクラスのインスタンス化を行い、その結果を SET 文でオブジェクト参照データ項目に格納します。

3）「データ項目名 :: メソッド名」でメソッド呼び出しの返却値を表します。この結果は SET 文で他の項目に転記したり、DISPLAY 文でコンソールに書き出したりすることができます。
これと同じ方法で、プロパティや ENUM の参照も可能です。

Visual COBOL は、オブジェクト指向型言語に親しみのある開発者も抵抗なく COBOL を記述できるよう .NET 向けに方言を拡張しています。この拡張された方言ではクラス参照の変数であれば DECLARE 文を使って PROCEDURE DIVISION 中で変数を定義することができます。上のコードを例えば以下のように元となる C# に記法を近づけることもできます。

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.      RAND.
       PROCEDURE       DIVISION.
           DECLARE RANDOBJ AS TYPE System.Random = new System.Random()
           PERFORM 10 TIMES
               DISPLAY RANDOBJ::Next()
           END-PERFORM.
           STOP RUN.
```

6.2 構造化例外処理

本書の前半のチュートリアルで、COBOL で書いたクラスメソッドから例外をスローする例を見ましたが、その反対に、COBOL から COBOL 以外のメソッドを呼び出す際に、呼び出し先がスローした例外を COBOL でキャッチすることができます。これには TRY 文を使用します。
TRY 文は COBOL の ISO2002 規格には含まれていない、Micro Focus の独自構文ですが、C#, Java, Visual Basic などほとんどすべてのオブジェクト指向言語がサポートしている構文です。COBOL では以下の構文規則で記述します。

```
TRY 無条件文-1
      { CATCH 一意名-1 無条件文-2 }...
      [ FINALLY 無条件文-3 ]
END-TRY
```

ここで、一意名 -1 は、System.Exception クラスまたはそのサブクラスのオブジェクトを指定します。 CATCH ブロックの無条件文 -2 は、無条件文 -1 の実行中にスローされた例外に対して実行されます。例外がスローされてもされなくても、FINALLY ブロックの無条件文 -3 は実行されます。
以下の例題プログラムは、.NET Framework の System.DateTime クラスのコンストラクタに対して、実行年 2 月 29 日という日付が指定されています。本稿は 2017 年に執筆されており 2017 年 2 月 29 日という存在しない日付が渡され、例外がスローされます。COBOL プログラムがこれをキャッチし、例外メッセージを DISPLAY している様子がわかります。

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 aDATE                TYPE DateTime.
01 anExptn              TYPE Exception.
PROCEDURE DIVISION.
    TRY
        SET aDATE TO new DateTime(TYPE DateTime::Today::Year, 2, 29)
        DISPLAY TYPE DateTime::Today::Year
    CATCH anEXPTN
        DISPLAY anEXPTN::Message::ToString()
        DISPLAY TYPE DateTime::Today::Year "年はうるう年ではありません。"
    FINALLY
        DISPLAY "処理を完了します。"
    END-TRY.
```

```
c:¥work¥whitepaper¥dotnetsol¥managedconsole¥bin¥debug¥ManagedConsole.exe
Year、Month および Day パラメーターが表現できない DateTime を示しています。
2017年はうるう年ではありません。
処理を完了します。
```

6.3 配列オブジェクトの参照

.NET アプリケーションにおける配列オブジェクトを介したデータ授受は一般的です。一口に配列と言っても .NET Framework では一次元配列、多次元配列、ジャグ配列と幾つかのタイプが用意されています。Visual COBOL は配列のタイプに関わらず COBOL 言語らしい文法で配列にアクセスする方法が用意されています。.NET Framework では配列は System.Array クラスのインスタンスとして実装されます。COBOL で記述する配列も同様であり、C# や VB.NET といった他の .NET 言語がそれらの言語同士でやりとりするかのようにして COBOL とも配列オブジェクトを使ったデータ授受が可能です。またこのクラスに用意されている様々な操作機能を COBOL で利用することも可能です。更に、配列の操作を意識した COBOL らしい拡張記法も Visual COBOL は用意しています。本項ではそれらの技法を紹介します。

6.3.1 一次元配列

1 次元配列は、固定数の要素を直線的に格納し、単一の添え字を用いて任意の 1 項目を識別します。

図6−1 一次元配列のイメージ

.NET の配列オブジェクトを COBOL で表現する場合、変数は 01 レベルで宣言します。要素数については、OCCURS 句で指定します。各要素へのアクセスは添え字を使いますが、C# 等と異なり、COBOL では添え字の開始は1からとなります。配列が MOVE 文に対応した基本型で構成されているのであれば、MOVE 文を使って値を格納することもできます。

■ C# の記述例

```
int[] ary1 = new int[3];
ary1[0] = 10;
ary1[1] = 100;
ary1[2] = 1000;
```

■ COBOL の記述例

```
01 ARY1   BINARY-LONG    OCCURS 3.
PROCEDURE DIVISION.
    SET ARY1(1) TO 10.
    MOVE 100 TO ARY1(2).
    MOVE 1000 TO ARY1(3).
```

.NET では配列の宣言時はサイズを指定せず初期化のタイミング等で与えられた要素数に応じてメモリ領域を割り当てます。COBOL でもこの記法に合わせて宣言時は OCCURS ANY とし、実行時に要素数を確定させることも可能です。下記の例では OCCURS ANY で宣言した配列に SET CONTENT 文を用いて要素数の確定に加えて値の格納も同時に処理しています。

■ C# の記述例

```
String[] ary2 = {"Newybury", "Mountain View", "Atlanta", "Amsterdam"};
```

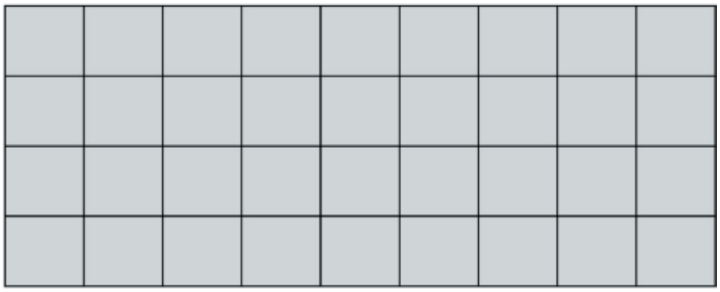
■ COBOL の記述例

```
01 ARY2  string OCCURS ANY.
PROCEDURE DIVISION.
SET CONTENT OF ARY2 TO ("Newbury" "Mountain View"
                                  "Atlanta" "Amsterdam")
```

6.3.2 多次元配列

多次元配列は複数の次元で構成される配列となります。各要素中の配列は同数の要素を保持します。例えば、二次元であれば長方形のようなイメージとなるため、rectangular(長方形の) 配列とも呼ばれます。

図6−2 多次元配列のイメージ（4×8の二次元配列の例）



配列オブジェクトの宣言は
OCCURS n|ANY [n|ANY] …
のように OCCURS 句に続けて次元の数だけ要素数或いは ANY を指定します。各要素へは添え字をスペース区切りで指定してアクセスします。

■ C# の記述例

```
int[, ,] ary3 = new int[4, 3, 2]{{{1,2},{3,4},{5,6}},
                                 {{7,8},{9,10},{11,12}},
                                 {{13,14},{15,16},{17,18}},
                                 {{19,20},{21,22},{23,24}}};
    Console.WriteLine(ary3[2, 1, 0].ToString());
```

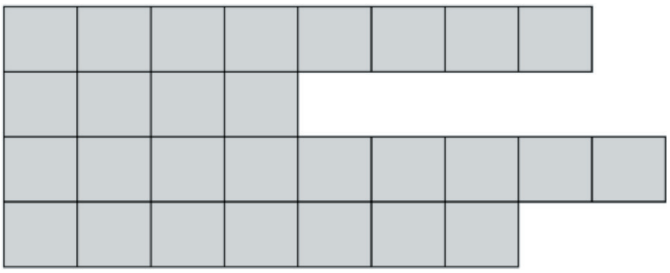
■ COBOL の記述例

```
       01 ARY3 BINARY-LONG OCCURS ANY ANY ANY.
       PROCEDURE DIVISION.
           SET CONTENT OF ARY3 TO (((1 2) (3 4) (5 6))
                                  ((7 8) (9 10) (11 12))
                                  ((13 14) (15 16) (17 18))
                                  ((19 20) (21 22) (23 24))).
           DISPLAY ARY3(3 2 1).
```

6.3.3 ジャグ配列

ジャグ配列は二次元で考えた場合、行ごとに列の数が異なっても構わないタイプの配列です。つまり、各要素も配列となり、「配列の配列」と呼ばれることもあります。このタイプの配列をイメージした場合下図のようにぎざぎざのかたちにもなるため、「jagged（ぎざぎざの）array」と名付けられたと言われています。

図6−3　ジャグ配列のイメージ



配列オブジェクトの宣言は
OCCURS n|ANY [OCCURS n|ANY] [OCCURS n|ANY] …
のように次元の数だけ OCCURS 句を繰り返します。OCCURS 句に続いて数値を指定した場合は、その次元における最大要素数として解釈されます。要素数は SET SIZE 文で指定します。

■ C# の記述例

```
      int[][] ary4 = new int[3][];
      ary4[0] = new int[5];
      ary4[1] = new int[4];
      ary4[2] = new int[2];

      ary4[1][3] = 456;
```

■ COBOL の記述例

```
01 ARY4 BINARY-LONG OCCURS ANY OCCURS ANY.
PROCEDURE DIVISION.
    SET size OF ARY4 TO 3.
    SET size OF ARY4(1) TO 5.
    SET size OF ARY4(2) TO 4.
    SET size OF ARY4(3) TO 2.
    MOVE 456 TO ARY4(1 4).
```

6.3.4 配列の操作

C# や VB.NET にはそれぞれ foreach 文、For Each 文のようにコレクションの数だけループをまわすための構文が用意されています。これに倣い Visual COBOL には PERFORM VARYING … THRU … という拡張 COBOL 構文が追加されています。この構文の PERFORM 文は VARYING のループ変数を、THRU 以下の配列オブジェクトの全要素に渡ってループさせます。

.NET Framework の配列は System.Array クラスのインスタンスとして実装されているため、System.Array クラスに実装されている SORT や REVERSE 等のメソッドを利用することができます。以下はこれらの技術を用いて C# と COBOL を連携させた例題です。本例では、C# で作成した配列の参照を COBOL に渡し、COBOL 側で System.Array クラスの Sort メソッドを使ってソートをします。また、ソート処理の前後で PERFORM VARYING … THRU … 構文を利用し少ないステップで配列の要素数分ループを回しています。

■ C#

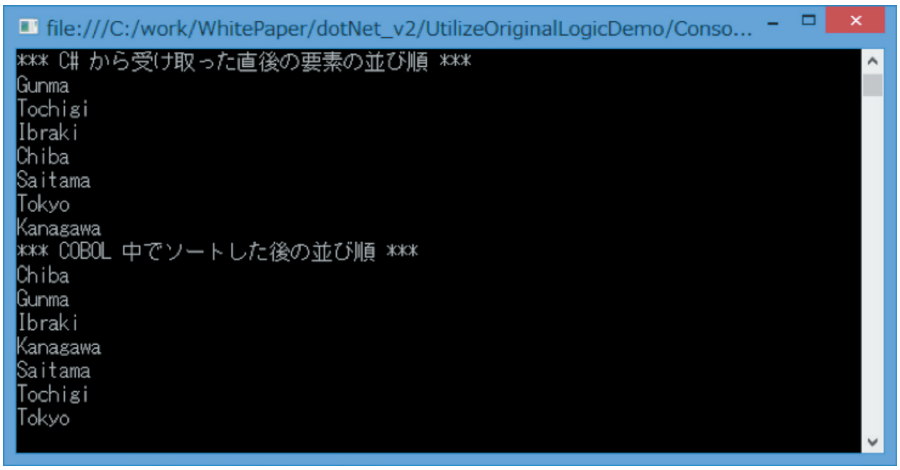
```
class Program
{
    static void Main(string[] args)
    {
        ArrayCOBOLLibrary.COBOLArray coblib = new ArrayCOBOLLibrary.COBOLArray();
        String[] ArrayParam = { "Gunma", "Tochigi", "Ibraki",
                                "Chiba", "Saitama", "Tokyo", "Kanagawa"};
        coblib.COBOLArray(ref ArrayParam);
    }
}
```

■ COBOL

```
PROGRAM-ID. COBOLArray AS "ArrayCOBOLLibrary.COBOLArray".

DATA DIVISION.
LINKAGE SECTION.
01 ArrayParam   string OCCURS ANY.
PROCEDURE DIVISION USING ArrayParam.
    DISPLAY "*** C# から受け取った直後の要素の並び順 ***"
    PERFORM VARYING E1 AS string THRU ArrayParam
       DISPLAY E1
    END-PERFORM.
    INVOKE TYPE System.Array::Sort(ArrayParam).
    DISPLAY "*** COBOL 中でソートした後の並び順 ***"
    PERFORM VARYING E2 AS string THRU ArrayParam
       DISPLAY E2
    END-PERFORM.
    GOBACK.
END PROGRAM COBOLArray.
```

■ 実行結果



7. まとめ

.NET Framework の中で COBOL が主要なプログラミング言語として活躍することの意義と、その技術的な背景について実例を通して考察してきました。.NET Framework 4.7 の新時代においても、最新鋭のアプリケーション開発に COBOL の持つ力をこれまで以上に活用してゆくことをぜひご検討ください。