CodeZine
開発者のための実装系Webマガジン

特集記事

C++/CLIで、機能拡張を簡単に実現するフレームワーク「MEF」を試してみた

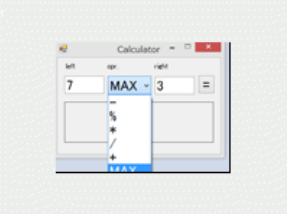
re-buildなしに機能拡張！

C++ .NET Framework

επιστημη[著] 2014/06/05 14:00

ダウンロード ↓ サンプルファイル (29.4 KB)

今回のお題は「MEF：Managed Extensibility Framework」。.NET Framework 4（Visual Studio 2010）で追加された、アプリケーションに拡張性を持たせることを目的としたフレームワークです。「re-buildなしにアプリケーションの機能拡張ができないか？」との相談を受け、MEFを試してみることにしました。



MEFは.NETライブラリですから、ピュアなC++では使えないけど「ほぼC++」なC++/CLIなら何とかなるんじゃないかと。MSDNで見つけたドキュメントを手掛かりに MEF：はじめの一歩をC++/CLIで踏み出しました。

DLL、COM、そしてMEF

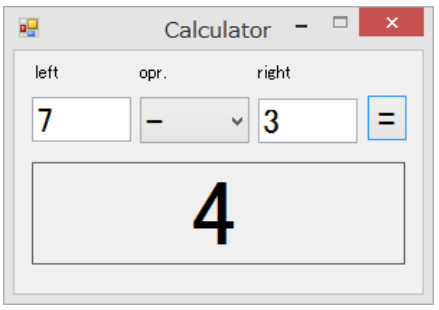
re-buildなしに機能拡張する手段はずっと以前からあります、WindowsではDLLがおなじみですね。DLLの中には関数名を埋め込むことができ、LoadLibrary()でDLLを読み込んだ後GetProcAddress()に関数名を与えれば、当該関数のアドレスが手に入り/呼び出すことができます。ただし手に入るのは関数のアドレスだけで、引数や戻り値に関する情報はもらえないので決め打ちにするしかありませんし、関数/変数をひとまとめにしたオブジェクト（クラス）を定義して呼び出すこともできません。そこでオブジェクトを定義し呼び出せるCOMの登場となるわけです。が、COMってシロモノは作る側も使う側もかなりややこしく、決して"お手軽"とは言い難い。

MEFによる機能拡張では、拡張したい機能をinterfaceで定義します。機能を提供する側はそのinterfaceを実装したクラスを作ってExport（外に出す/公開する）し、対して機能を利用する側はinterfaceの受け皿を用意して、そこにImport（取り込む）します。MEFは提供（Export）側と利用（Import）側との仲介を行い、両者を結びつけてくれます。Export/Importはクラスや変数にアトリビュート：[Export(～)]/[Import(～)]を付記するだけなので、COMに比べて圧倒的に簡単です。

能書きはこれくらいにして、サンプルをお見せしましょう。

二項電卓：Windows Formアプリケーションの作り方

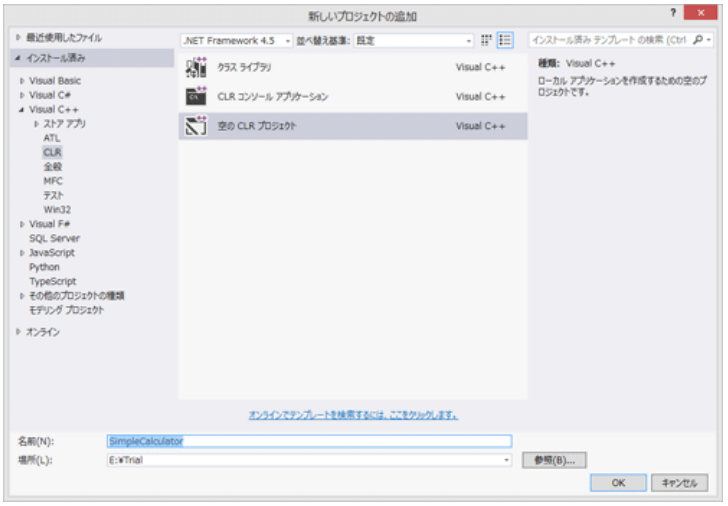
サンプルに用意したのは、こんな外観の電卓アプリケーション。



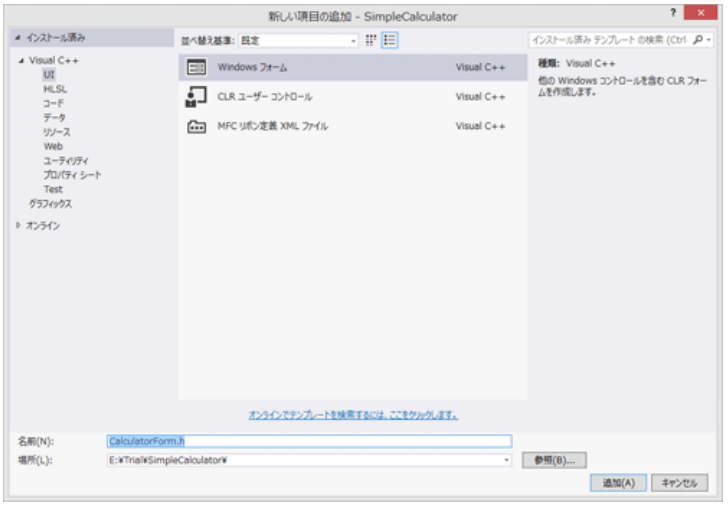
int値を2つ入力し、演算子を選んで「=」ボタンを押すと答えが表示される、あまりのショボさに電卓と呼ぶのも恥ずかしい二項電卓をWindows Formsで作りました。ComboBoxに列挙される演算子をMEFで追加しようという魂胆です。

Visual Studio 2012以降、C++/CLIによるCLRフォームアプリケーションのひな型がプロジェクトテンプレートから消えてしまいました。けども作れなくなったわけではありません。ちょっと寄り道になりますが、フォームアプリケーションの作り方を紹介しておきます。

まず、プロジェクトテンプレートでは「空のCLRプロジェクト」を選択します。プロジェクト名は"SimpleCalculator"としましょう。



つぎにメニュー：プロジェクト/新しい項目の追加...でWindowsフォームを追加します。フォーム名は"CalculatorForm"で。

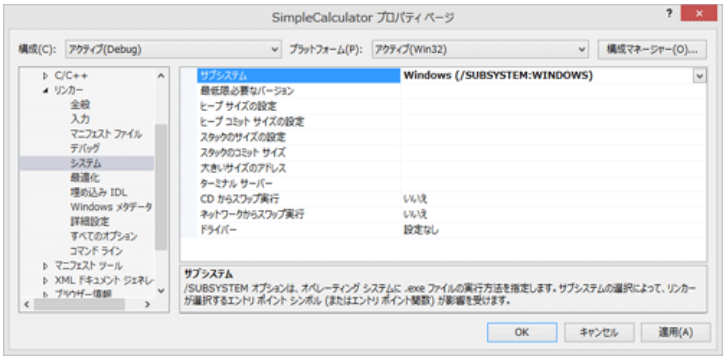


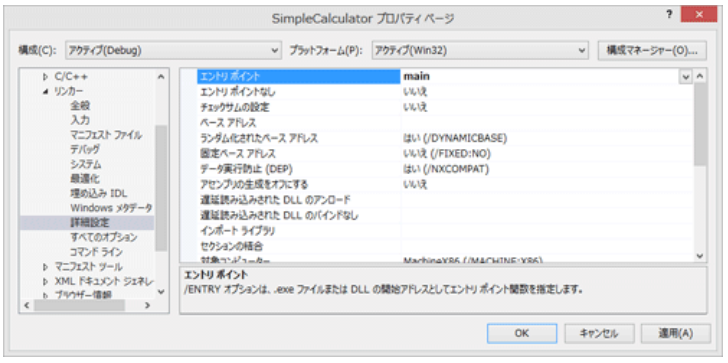
さらにメニュー：プロジェクト/新しい項目の追加...でC++ファイル（.cpp）、名前を"SimpleCalculator.cpp"としてコードを追加します。これがアプリケーションの入り口であるmain()です。

SimpleCalculator.cpp

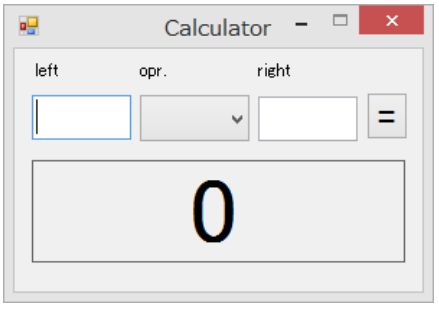
```
#include "CalculatorForm.h"
using namespace System;

[STAThread]
int main(array<String^>^ args) {
  using namespace System::Windows::Forms;
  Application::EnableVisualStyles();
  Application::SetCompatibleTextRenderingDefault(false);
  Application::Run(gcnew SimpleCalculator::CalculatorForm());
  return 0;
}
```

残るはプロジェクト・プロパティの設定。構成プロパティ/リンカー/システム/サブシステムをWindows(/SUBSYSTEM:WINDOWS)に、構成プロパティ/リンカー/詳細設定/エントリポイントをmainとします。

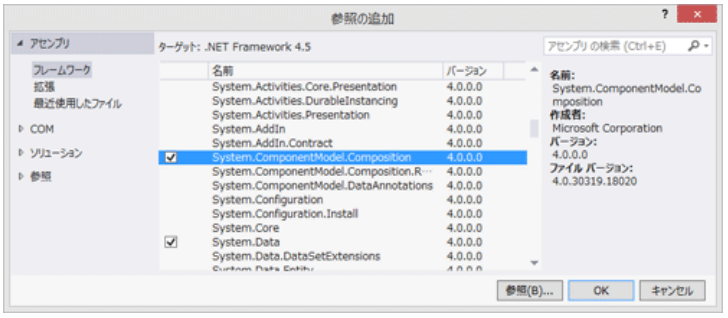




ここでビルド/実行し、空っぽのCalculatorFormが現れたら出来上がり。2つのTextBoxとComboBox、Buttonそして大きなLabelを貼り付け、Button-Clickハンドラを用意して電卓フォームの完成です。



最後にもう一つ、MEFを提供するアセンブリ：System.ComponentModel.Compositionへの参照を追加しておきます。



interfaceの定義とその実装

それではMEFを使った拡張機能をこしらえましょうか。

まずはinterfaceから。定義するinterfaceは、

- intを2つ与えるとintを返す演算：IOperation
- 演算に付けられた演算子（の名前）：IOperationData
- 2つのintと演算子を与えると演算子に応じた演算を行って結果を返す計算機：ICalculator

の3つです。

Interface.h

```
#ifndef INTERFACE_H_
#define INTERFACE_H_

namespace SimpleCalculator {

  public interface class IOperation {
    int Operate(int left, int right);
  };

  public interface class IOperationData {
    property System::String^ Symbol { System::String^ get(); }
  };

  public interface class ICalculator {
    int Calculate(int left, System::String^ opr, int right);
    System::Collections::Generic::IEnumerable<System::String^>^ Symbols();
  };

}

#endif
```

IOperationの実装は、ひとまず加算：Addと減算：Subtractの2つを用意しましょう。

Operations.cpp

```
#include "Interface.h"

using namespace System::ComponentModel::Composition;

namespace SimpleCalculator {

  [Export(IOperation::typeid)]
  [ExportMetadata(L"Symbol", L"+")]
  ref class Add : IOperation {
  public:
    virtual int Operate(int left, int right) {
      return left + right;
    }
  };

  [Export(IOperation::typeid)]
  [ExportMetadata(L"Symbol", L"-")]
  ref class Subtract : IOperation {
  public:
    virtual int Operate(int left, int right) {
      return left - right;
    }
  };

}
```

ここでのキモは[]で囲まれたアトリビュートです。「Add/SubtractはIOpertationを実装し、不可情報：Symbolを"+"/"-"としてExport（公開）する」ことを表しています。

そしてICalculatorを実装したCalculator：

Calculator.cpp

```
#include "Interface.h"

using namespace System;
using namespace System::ComponentModel::Composition;
using namespace System::Collections::Generic;
```

```
namespace SimpleCalculator {

  [Export(ICalculator::typeid)]
  ref class Calculator : ICalculator {
  private:
    [ImportMany]
    IEnumerable<System::Lazy<IOperation^, IOperationData^>^>^ operations_;

  public:
    // Symbolが一致するIOperationを見つけ、実行する
    virtual int Calculate(int left, System::String^ operation, int right) {
      for each (Lazy<IOperation^, IOperationData^>^ item in operations_) {
        if ( item->Metadata->Symbol == operation )
          return item->Value->Operate(left, right);
      }
      throw gcnew NotSupportedException(operation);
    }

    // Symbolの列挙を返す
    virtual IEnumerable<System::String^>^ Symbols() {
      auto result = gcnew List<String^>();
      for each (Lazy<IOperation^, IOperationData^>^ item in operations_) {
        result->Add(item->Metadata->Symbol);
      }
      return result;
    }

  };

}
```

メンバ変数operations_は、IOperationとIOperationDataの組を複数個（演算の数だけ）抱えます。アトリビュート[ImportMany]に注目、MEFはこのアトリビュートが付けられた変数に[Export(IOperation::typeid)]な複数のclassを結び付けてくれます。

メソッドOperate()は、operations_に納められたIOperationとIOperationDataの組を使って演算子に応じたIOperationを見つけて演算を行いますし、Symbols()は登録されている演算子の列挙を返します。

フォームに組み込む

もうひといき、ICalculatorの実装Calculatorができたので、これをCalculatorFormに組み込みましょう。

アトリビュート[Import(ICalculator::typeid)]をつけたICalculatorへ calculator_をメンバに追加します。

CalculatorForm.h（抜粋）

```
#pragma once
#include "Interface.h"

namespace SimpleCalculator {

  public ref class CalculatorForm : public System::Windows::Forms::Form
  {
  public:
    CalculatorForm();

  protected:
    ~CalculatorForm();

  private: System::Windows::Forms::TextBox^  tbxLeft;
  private: System::Windows::Forms::ComboBox^  cbxOpr;
  private: System::Windows::Forms::TextBox^  tbxRight;
  private: System::Windows::Forms::Button^  btnExec;
  private: System::Windows::Forms::Label^  lblResult;
```

```
  private:
    System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
....省略
#pragma endregion

  private:
    // 「=」ボタン・クリックのハンドラ
    System::Void btnExec_Click(System::Object^  sender, System::EventArgs^  e);

    [System::ComponentModel::Composition::Import(ICalculator::typeid)]
    ICalculator^ calculator_;
  };

}
```

そして実装。

CalculatorForm.cpp

```
#include "CalculatorForm.h"

using namespace System;
using namespace System::ComponentModel::Composition;
using namespace System::ComponentModel::Composition::Hosting;

namespace SimpleCalculator {

  CalculatorForm::CalculatorForm() {
    InitializeComponent();

    // Import/Export カタログをつくる
    auto catalog = gcnew AggregateCatalog();
    // 自分自身のアセンブリから探して追加する
    catalog->Catalogs->Add(gcnew AssemblyCatalog(CalculatorForm::typeid->Assembly));

    // カタログから作られたコンテナを基にImport/Exportを結びつける
    AttributedModelServices::ComposeParts(gcnew CompositionContainer(catalog), this);

    // 得られた演算子(Symbol)をComboBoxに追加する
    for each ( String^ symbol in calculator_->Symbols() ) {
      cbxOpr->Items->Add(symbol);
    }
    cbxOpr->SelectedIndex = 0;
  }

  CalculatorForm::~CalculatorForm() {
    if (components) {
      delete components;
    }
  }

  System::Void CalculatorForm::btnExec_Click(System::Object^  sender, System::EventArgs^  e) {
    Int32 left;
    Int32 right;
    // フォームから 左辺/右辺/演算子を取り出し、計算して結果を表示する
    if ( Int32::TryParse(tbxLeft->Text, left) && Int32::TryParse(tbxRight->Text, right) ) {
      try {
        int result = calculator_->Calculate(left, cbxOpr->SelectedItem->ToString(), right);
        lblResult->Text = result.ToString();
      } catch ( Exception^ ) {
        lblResult->Text = L"error";
      }
```

```
    } else {
      lblResult->Text = L"?";
    }
  }

}
```

ここでのキモはコンストラクタ、これまでに用意した[Export(～)]と[Import(～)]/[ImportMany]とをコンストラクタで（つまり実行時に）結びつけます。

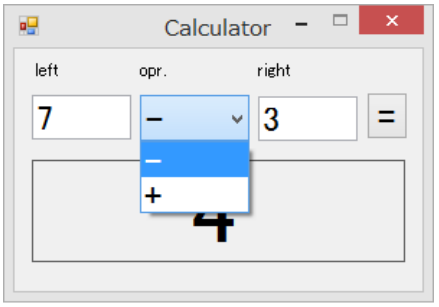
ExportとImportの一覧表に相当するカタログ：AggregateCatalogを生成し、自分自身が属するアセンブリをAdd()することで、アセンブリ内のExport/Importがカタログに登録されます。

カタログを引数に生成されたコンテナ：CompositionContainerとthis（＝CalculatorForm）とをAttributedModelServices::ComposeParts()に与えると、CalculatorForm内から[Import(～)]が付けられた変数と、対応する（同じtypeidを持った）[Export(～)]の追加クラスをコンテナから探し出して結び付けます。

このとき、ICalculatorを実装したCalculator内には[ImportMany]なoperations_があるので、さらにこいつに対応する[Export(～)]もコンテナから探し出して……と、再帰的に結び付けてくれます。

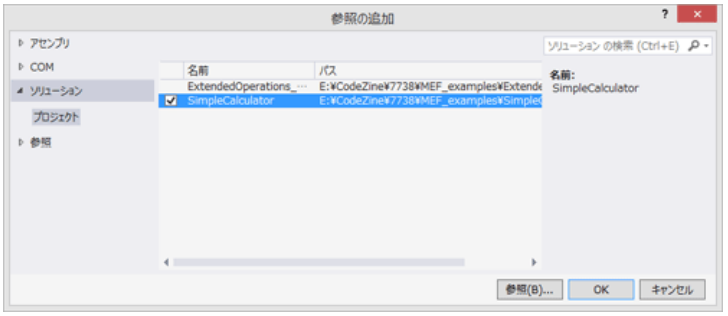
「=」ボタンが押された時のハンドラは簡単。TextBoxとComboBoxからint値2つと演算子を取り出し、operator_に投げて結果を表示すればよし。

ビルド/実行してみましょう。ComboBoxに「+」と「-」が入っていますね。



演算子の追加

では本題、「re-buildなしに機能を拡張」しますか。ソリューションに新しいプロジェクト：CLRクラスライブラリ"ExtendedOpertaions"を起こし、プロジェクト・プロパティの共通プロパティでSimpleCalculatorとSystem.ComponentModel.Compositionの参照を追加します。



あとはSimpleCalculatorで定義したOperations.cppとまったく同じ体裁で乗算/除算を定義します。

ExtendedOperations.cpp

```
#include "stdafx.h"

namespace ExtendedOperations {

  using namespace System::ComponentModel::Composition;
  using namespace SimpleCalculator;
```

```
  [Export(IOperation::typeid)]
  [ExportMetadata(L"Symbol", L"*")]
  ref class Multiple : IOperation {
  public:
    virtual int Operate(int left, int right) {
      return left * right;
    }
  };

  [Export(IOperation::typeid)]
  [ExportMetadata(L"Symbol", L"/")]
  ref class Subtract : IOperation {
  public:
    virtual int Operate(int left, int right) {
      return left / right;
    }
  };

}
```

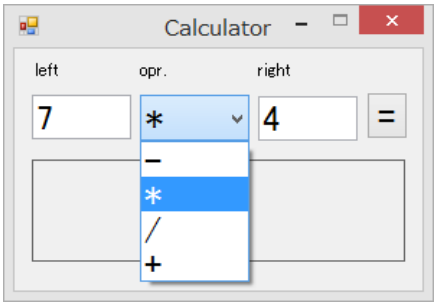
……おっとゴメンナサイ。拡張用アセンブリ（ExtendedOperations.dll）を読み込む部分を忘れてました。SimpleCalculator::CalculatorFormのコンストラクタに追加しなくちゃ。

CalculatorForm::CalculatorForm()

```
  CalculatorForm::CalculatorForm() {
    InitializeComponent();

    // Import/Export カタログをつくる
    auto catalog = gcnew AggregateCatalog();
    // まずは自分自身のアセンブリから
    catalog->Catalogs->Add(gcnew AssemblyCatalog(CalculatorForm::typeid->Assembly));

    // そして自分自身の置かれたディレクトリから見つけてくる（追加ここから）
    String^ myLocation = System::IO::Path::GetDirectoryName(
                           System::Reflection::MethodBase::GetCurrentMethod()
                           ->DeclaringType->Assembly->Location);
    catalog->Catalogs->Add(gcnew DirectoryCatalog(myLocation));
    //（追加ここまで）

    // カタログから作られたコンテナを基にImport/Exportを結びつける
    AttributedModelServices::ComposeParts(gcnew CompositionContainer(catalog), this);

    // 得られた演算子(Symbol)をComboBoxに追加する
    for each ( String^ symbol in calculator_->Symbols() ) {
      cbxOpr->Items->Add(symbol);
    }
    cbxOpr->SelectedIndex = 0;
  }
```

これにより、SimpleCalculator.exeの置かれたディレクトリにあるアセンブリがすべて読み込まれます。

ExtendedOperationsをビルドしたのち、SimpleCalculatorを実行するとComboBoxに「*」と「/」が追加されています。

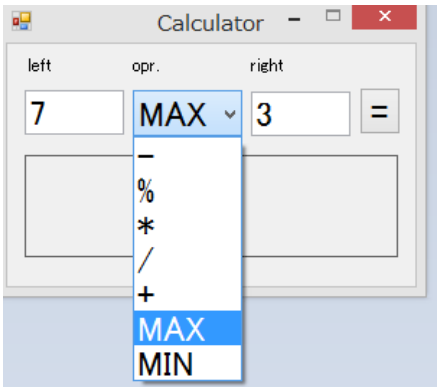


当然ながら .NETアセンブリであれば、C++/CLIに限らずC#やVBで拡張しても構いません。C#クラスライブラリ・プロジェクトを起こし、

ExtendedOperations.cs

```
using System.ComponentModel.Composition;
using SimpleCalculator;

namespace ExtendedOperations {

  [Export(typeof(IOperation))]
  [ExportMetadata("Symbol", "%")]
  class Modulus : IOperation {
    public int Operate(int left, int right) {
      return left % right;
    }
  };

  [Export(typeof(IOperation))]
  [ExportMetadata("Symbol", "MIN")]
  class Minimum : IOperation {
    public int Operate(int left, int right) {
      return left < right ? left : right;
    }
  };

  [Export(typeof(IOperation))]
  [ExportMetadata("Symbol", "MAX")]
  class Maxumum : IOperation {
    public int Operate(int left, int right) {
      return left > right ? left : right;
    }
  };
}
```

できたDLLをSimpleCalculator.exeと同じディレクトリに置けば……ほらね。



……面白いですねぇ、IOerationを定義したAddやSubtractなどは他のどこからも参照されず、ただ定義しExportしただけです。MEFはそれを手掛かりに動的に（実行時に）探し出し、インスタンスを生成してImportした受け皿に乗っけてくれるんですね。アセンブリを配置するだけで機能拡張できるのは、アプリケーションの提供者/利用者の双方に大きなメリットですよね。