EasyFormula

User Guide

Contents

1. Introduction

What is EasyFormula

EasyFormula is a software development component to create dynamic calculation formula. By using EasyFormula, your application can let user to create their own formula, so they can choose to how to process the data, how to organize or present the data instead of decided by the application developers. For example, if you are developing a billing system, by using EasyFormula, the end user can create a formula of client name to choose the client name should be printed as “David Williams” or “Williams, David”. You can archive this without using expensive reporting program.

The application developers can be benefited from EasyFormula from these 2 perspectives.

- Flexible

  EasyFormula give your application flexibility to meet variant requirements from different users. Users always have different requirements on how to calculate data, how to present their data in reports, etc. i.e. If you are developing a payroll system, as your client, each company may have their own formula to calculate commission for salesman. By using EasyFormula to let your user create a dynamic formula by themselves, you don’t have to make customization to meet the specific requirement from individual client.

- Better application architecture

  Your application probably has adopted beautiful multi-tier architecture to separate presentation layer, business logic layer, and data layer. But in some complex business environment, there are too many and too complex calculations at business layer itself. It’s worth to think separate these calculations from hardcoded source code. EasyFormula can help to make your business calculation more organized and easy to maintain, and you can also change your calculation logic right the way without having to rebuild the system again.

  EasyFormula also make a lot of easy for your developer to communicate with business analyst, as the business analyst can read the formula in plain text format by themself, they don’t have to go through source code to figure out you calculation logic.

Formula Syntax

Operant:

EasyFormula supports 5 types of operants:

5. String. Enclosed by double quota
i.e. “abc”, “David”

6. Number. Both integer and real
i.e. 3, 23.4, -4

7. Function. A list of built-in functions,see chapter 4
i.e. SubStr, DateDiff

8. Named formula, leading by @ sign
i.e. @ClientName, @Address

9. Context data. Data Provided by client application. Enclosed by curly bracket.
i.e. {Age}, {Price}

Operators:

+ plus and string concatenation
-
*
/
&& logical AND
|| logical OR
>
<
>=
<=
= logical equal

Some example of formula:

{Price}*0.16-10
@ClientName+@Address
{LastName}+”,”+{MiddleName}+” “+{LastName}

Formula := **<Operant>** [<**Operator**> <**Operant**>] [….n]

<**Operant**> :=[(] *String* | <**Function**> | *Number* | <**Formula Reference**> |
<**Context Data**> [)]

*<**Operator**> :=* ‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘&&’ | ‘||’ | ‘>’ | ‘<’ | ‘>=’ | ‘<=’ | ‘=’

<**Function**> := *Function Name* (<Operant>[,…n])

<Formula Reference> := @*Formula Name*

<Context Data> :={*Context data name*}

String

String constants enclosed by double quota, i.e. "Client Name"

Number

Interger or floating number

Function Name

Is one of supported built-in functions.

Formula Name

Name of another formula objects, the name to formula object mapping is maintained by client application, which can be assessed by DataAdapter.

Context data name

A name link to the data provided by client application. I.e. Firstname,LastName

Availability and Platforms

EasyFormula currently available on Win32 platform only, Java version will be released soon. If you are interested in Java version, please register at www.jtech.ca , we will notice you as long as it is released.

EasyFormula for Win32 have 2 editions:

- **EasyFormula for VCL**, which support for Borland C++/Delphi development environment.
- **EasyFormula for MFC**, which support for Microsoft Visual Studio environment.

Both edition comes with 2 DLL, one is native C++ library, the other is a ActiveX/COM library which can be used by other programming languages.

They are available for free evaluation at:
http://www.jtech.ca/Download/ProductList.asp.

Note: A random evaluation message will be returned by Calculate method, please apply the purchased serial code to disable evaluation message.

Terminology

Client Application:	The application in which the EasyFormula component is used.
User :	The end user of client application.
Developer :	The developer of client application.

2. C++ Class

Each EasyFormula package comes with 2 DLL, one is used for native C++ development, the other is an ActiveX component that can be used by other development environment, like VB, ASP. Both the classes of C++ library and interface of ActiveX component share similar methods.

The native C++ library includes 2 classes and a couple of built-in functions. EFormula is the prime class that is used to construct a formula, parse and perform calculation.

DataAdapter is a helper class to interpret a context related operant in a formula, it provides a interface to access data from client application, i.e. A formula is defined as: {age}+5, where {age} is data provided by your client application, while EasyFormula is called to perform calculation, the DataAdapter object will be called to fetch this local context related data from client application. DataAdapter is a virtual class, developers have to implement a subclass to inherit from DataAdapter and overwrite 3 methods of it, these 3 methods will be called by EFormula to get your context data.

EFormula

Constructors:	
EFormula()	
	Construct a EFormula object without setting formula string, the formula string can be set by SetFormulaStr method later.
EFormula(const char* strFormula)	
	Construct a EFormula object with a formula string
Methods:	
`bool Compile(int &ErrPos, string &ErrorMsg)`	

	Compile formula string into internal binary format. It will scan the string, if an unknow element or unrecongnized syntax is encounted, it returns false with error position and error message. Return: true if compile succeed False if error encounted Parameters: ErrPos: out parameter. The position of first character that cause the error. ErrPos is 0 if this method returns true. ErrorMsg: out parameter. The error message if this method returns false.
`Variant Calculate(int &ErrPos, string &Msg)`	
	Perform calculation, the return value is the calculation result. Return: Calculation result ErrPos: out parameter. The position of first character that cause the error. ErrPos is 0 if this method returns true. ErrorMsg: out parameter. The error message if this method returns false. Note: a random evaluation message will be returned by evaluation copy, you need apply a purchased serial code to disable this message.
`char *GetFormulaStr()`	
	Return current formula string.
`EFormula& operator=(const EFormula& src)`	
	Assign operator, return a reference of current object.
`void SetDataAdapter(DataAdapter* pAdapter)`	
	Set DataAdapter pointer, it will be called by Calculate method to get data of context related operant. If this method is not called, a default DataAapter will be used which will return empty string for all context related operant. Parameters: PAdapter: in parameter, pointer to DataAdapter object.
`void SetFormulaStr(const char* strFormula)`	

	Set formula string. Parameters: StrFormula: pointer to a string represents the formula.
`void SetSerialCode(const char* Serial)`	
	Set the serial code after purchasing a license. Parameter: Serial: the serial code string

DataAdapter

Methods:	
virtual Variant GetOperantValue (const char *pOperantName)	
	Get context value for a context related operant. Return: value for the operant Parameters: pOperantName: In paramter, pointer to the name of operant.
virtual string GetFormulaByName(const char *pFormulaName)	
	Get formula string referred by a formula name. The client application can give a name to a formula component, this named formula can be referred by other formula components. Return: formula string Parameters: PFormulaName: In parameter. Pointer to formula name.
virtual bool IsKnown(const char *pOperantName)	
	Check if this operant can be interpreted by local context. Return: true if can be interpreted. False if cannot be interpreted.

3. COM Wrapper

In order to support development environment other than C++, we also build a ActiveX/COM library, and implemented 2 automation interfaces, so it can also be used in VB, Delphi, even Script development.

These 2 interface IFormula and IDataAdapter have similar methods as native C++ library.

Methods:	
Compile([in, out] VARIANT *ErrPos, [in, out] VARIANT *MSG)	
	Compile formula string into internal binary format. It scans the string, if an unknow element or unrecongnized syntax is encounted, the ErrPos is set to a positive value. Parameters: ErrPos: out parameter. The position of first character that cause the error. ErrPos is 0 if this method returns true. ErrorMsg: out parameter. The error message if this method returns false.
Calculate([in,out]VARIANT *pErrPos,[in,out]VARIANT *pMSG,[out,retval] VARIANT *pResult)	
	Perform calculation, the return value is the calculation result. Return: pResult, Calculation result. ErrPos: out parameter. The position of first character that cause the error. ErrPos is 0 if this method returns true. ErrorMsg: out parameter. The error message if this method returns false. Note: a random evaluation message will be returned by evaluation copy, you need apply a purchased serial code to disable this message.
GetFormulaStr([out, retval] BSTR * pFormula)	
	Return current formula string.
SetDataAdapter([in] IDataAdapter * IAdapter)	

	Set IDataAdapter pointer, it will be called by Calculate method to get data of context related operant. If this method is not called, a default IDataAapter will be used which will return empty string for all context related operant. Parameters: pAdapter: in parameter, pointer to IDataAdapter object.
SetFormulaStr([in] BSTR FormulaString)	
	Set formula string. Parameters: FormulaString: In parameter, pointer to a string represents the formula.
SetSerialCode([in] BSTR SerialCode)	
	Set the serial code after purchasing a license. Parameter: SerialCode: the serial code string

IDataAdapter

GUID: {29FC75B5-5BC8-40B3-90BD-3DFD3D4381A0} (VCL version)
GUID: {69DB05FC-34B3-4209-8249-E7FB05225A89} {MFC version)

Methods:	
GetOperantValue (in] BSTR OperantName, [out, retval] VARIANT * Value)	
	Get context value for a context related operant. Return: value for the operant Parameters: OperantName: In paramter, the name of operant.
GetFormulaByName([in] BSTR FormulaName, [out, retval] BSTR * bstrFormula)	
	Get formula string referred by a formula name. The client application can give a name to a formula component, this named formula can be referred by other formula components. Return: formula string Parameters: PFormulaName: In parameter. Pointer to formula name.

IsKnown([in] BSTR OperantName, [out, retval] VARIANT * bKnown)	
	Check if this operant can be interpreted by local context. Return: true if can be interpreted. False if cannot be interpreted.

4. Built-in Functions

String

Length(OrgStr)

Return the length of a string. OrgStr is the input string

Trim(OrgStr)
Trim leading and trailing spaces of a string. OrgString is the input string.

TrimLeft(OrgStr)
Trim leading spaces of a string. OrgString is the input string.

TrimRight(OrgStr)
Trim trailing spaces of a string. OrgString is the input string.

UpperCase(OrgStr)
Convert a string to all capitals. OrgString is the input string.

LowerCase(OrgStr)
Convert a string to all lower case. OrgString is the input string.

Reverse(OrgStr)
Reverse a string. OrgString is the input string.

SubStr(OrgStr, StartPos, EndPos)
Extract a substring from a given string. OrgStr is the given string, StartPos is the index from which the substring starts, the first character is 1. EndPos is the index where the substring ends.

FindSub(OrgStr, Sub)
Returns the index at which a specified substring begins, where 1 is first character. If the substring is not fund, it returns 0.

Replace(OrgStr, Replaced, Replacing)
Replace a substring from a given string with a new substring. OrgStr is the original string, Replaced is the substring to be replaced, Replacing is the new substring. I.e. Replace(“Have lunch”,”lunch”,”dinner”) returns “Have dinner”.

Char(ASCII)
Return a character given by it’s ASCII code, i.e. Char(32) returns space.

Date/Time

Today()
Return a date value of today.

Now()
Return a date value of now.

AddDays(date, days)
Add number of days onto a given date.

AddWeeks(date, weeks)
Add number of weeks onto a given date.

AddMonths(date, months)
Add number of months onto a given date.

AddYears(date, years)
Add number of years onto a given date.

DateDiff(dateFrom, dateTo, Unit)
Returns the difference of two given date in specified unit.

Value of Unit

"Day" or "D": return difference in days.
"Month" or "M": returns difference in months.
"Year" or "Y": returns difference in years.
"Week" or "W": return difference in weeks.

i.e. DateDiff("10/15/2003","01/05/2003","D")

IsLeapYear(date)
Check if a give date is in leap year.

GetMonth(date)
Return month of a given date.

GetYear(date)
Return year of a given date.

DayOfMonth(date)
Return day of the month of a given date.

DayOfWeek(date, type)
Return day of the week of a given date.

Math

Min(num1, num2)

Returns lesser number between num1 and num2

Max(num1, num2)

Returns greater number between num1 and num2

Mod(num1, num2)

Modulus operation, returns the remainder of integer division , cannot be used with float data.

Round(num, digits)

Rounds off a floating-point number (decimal) to a specified number of decimal places. Num is the number to be round, digits is the number of decimal places.

Floor(num)

Returns an integer equal to, or the next integer less than, the number passed to it. i.e. Floor(-3.79) is –4, Floor(6.23) is 6

Power(base, exponent)

Raises Base to any power.

IsZero(num)

Determine if a float number can be regarded as zero.

Log2(num)

Calculate log base 2.

Log10(num)

Calculate log base 10.

LogN(B, ex)

Returns the log base B of ex.

Cos(X)

Returns the cosine of X.

Sin(X)

Returns the sine of X.

Tan(X)

Returns the tangent of X.

5. Examples

5.1. VCL

Using Native C++ library

- **Implement the data adapter**

```
Variant DataAdapter::GetOperantValue(const char*   pOperantName)
{
// This is the method to get client application data, in this example,
// 2 local data {Name} and {Tel} is provided.

      Variant Value;

    if(AnsiString(OperantName) == "Name")
        Value = Variant("David");
    else if(AnsiString(OperantName) == "Tel")
        Value = Variant("123456");
    else
        Value = Variant("");

    return(Value);
}

bool DataAdapter::IsKnown(const char    *pOperantName)
{
    bool isKnown = false;

    if(AnsiString(OperantName) == "Name")
         isKnown = true;
    else if(AnsiString(OperantName) == "Tel")
         isKnown = true;
    else
         isKnown = false;

    return(isKnown);
}

string DataAdapter::GetFormulaByName(const char *pFormulaName)
{
      // In this example, the data adapter hardcoded a formula
      // for ‘Contact’ as {Name}+’ ’+{Tel}
      //  In your real project, you may store the formula into
      //  database, in that case this method should check against
      //  database to fetch formula string

      string strFormula(“”);
      if(AnsiString(FormulaName) == "Contact")
            strFormula=string("{Name}")+string("+':'
      +")+string("{Tel}");

      return (strFormula);
}
```

- **Client Program**

```
      // Get formula string from a form
    AnsiString strFormula = Formula->Text;

      //Create EasyFormula object
    EFormula gFormula(strFormula.c_str());

      // Create data adapter class to get data from client application
    DataAdapter myAdapter;

    int ErrPos;
    string Msg;

      // pass the pointer of data adapter to formula object
   gFormula. SetDataAdapter(&myAdapter);

      // Check formula syntax
    if(gFormula.Compile(ErrPos,Msg))
    {
MessageDlg("Syntax Error",mtError,TMsgDlgButtons()<<mbOK, 0);
return;
    }
      // Call Calculate method to get result
    Variant value = gFormula.Calculate(ErrPos,Msg);

      // Check error message
    if(!ErrPos)
      MessageDlg((AnsiString)value,mtInformation,TMsgDlgButtons()<<mbOK
, 0);
    else
      MessageDlg("Calculation Error",mtError,TMsgDlgButtons()<<mbOK,
0);
```

Using COM Automation object

- **Implement the data adapter**

Type library definition:

```
[
  uuid(1B4586DA-D864-41FE-B0A5-05FBB9A60FFC),
  version(1.0),
  helpstring("TestAdapter Library")

]
library TestAdapter
{

  importlib("stdole2.tlb");
  importlib("EasyFormula.tlb");

  [
    uuid(E382618A-1682-4FE1-AC41-D1BD7384F6F9),
    version(1.0),
    helpstring("Dispatch interface for MyAdapter Object"),
    dual,
    oleautomation
  ]
   interface IMyAdapter: IDispatch
  {
  };

  [
    uuid(DCFC8E41-A2C3-47CF-A8BC-AAEA3293D6F6),
    version(1.0),
    helpstring("MyAdapter Object")
  ]
  coclass MyAdapter
  {
    interface IMyAdapter;
    [default] interface IDataAdapter;
  };

};
```

Class Declaration:

```
class ATL_NO_VTABLE TMyAdapterImpl :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<TMyAdapterImpl, &CLSID_MyAdapter>,
  public IDispatchImpl<IMyAdapter, &IID_IMyAdapter,
&LIBID_TestAdapter>,
  DUALINTERFACE_IMPL(MyAdapter, IDataAdapter)
{
public:
  TMyAdapterImpl()
  {
  }

  // Data used when registering Object
  //
  DECLARE_THREADING_MODEL(otApartment);
  DECLARE_PROGID("TestAdapter.MyAdapter");
  DECLARE_DESCRIPTION("");

  // Function invoked to (un)register object
  //
  static HRESULT WINAPI UpdateRegistry(BOOL bRegister)
  {
    TTypedComServerRegistrarT<TMyAdapterImpl>
    regObj(GetObjectCLSID(), GetProgID(), GetDescription());
    return regObj.UpdateRegistry(bRegister);
  }


BEGIN_COM_MAP(TMyAdapterImpl)
  COM_INTERFACE_ENTRY(IMyAdapter)
  COM_INTERFACE_ENTRY2(IDispatch, IMyAdapter)
  DUALINTERFACE_ENTRY(IDataAdapter)
END_COM_MAP()

// IMyAdapter
public:

  STDMETHOD(GetFormulaByName(BSTR FormulaName, BSTR* bstrFormula));
  STDMETHOD(GetOperantValue(BSTR OperantName, VARIANT* Value));
  STDMETHOD(IsKnown(BSTR OperantName, VARIANT* bKnown));
};
```

Adapter Class Implementation:

```
STDMETHODIMP TMyAdapterImpl::GetFormulaByName(BSTR FormulaName,
  BSTR* bstrFormula)
{

    if(AnsiString(FormulaName) == "Contact")
        *bstrFormula = WideString("{Name}") +WideString("+':'
+")+WideString("{Tel}");

    return S_OK;
}

STDMETHODIMP TMyAdapterImpl::GetOperantValue(BSTR OperantName,
  VARIANT* Value)
{

    if(AnsiString(OperantName) == "Name")
        *Value = Variant(WideString("David"));
    else if(AnsiString(OperantName) == "Tel")
        *Value = Variant(WideString("123456"));
    else
        *Value = Variant("");

    return S_OK;
}

STDMETHODIMP TMyAdapterImpl::IsKnown(BSTR OperantName, VARIANT* bKnown)
{

    if(AnsiString(OperantName) == "Name")
        *bKnown = Variant(true);
    else if(AnsiString(OperantName) == "Tel")
        *bKnown = Variant(true);
    else
```

```
        *bKnown = Variant(false);

    return S_OK;
}
```

- **C++ Client Program Using Custom Interface**

```
      // Get formula string from a form
    AnsiString strFormula = Formula->Text;

      /* Using COM */
      // Create an object for wrapper class of IFormula interface
   TFormula *pFormula = new TFormula(NULL);

      // Create an object for wrapper class of IDataAdapter interface
   TMyDataAdapter *pMyAdapter = new TMyDataAdapter(NULL);

      // Get wrapped default interface
   IDataAdapterPtr pIAdapter = pMyAdapter->GetDefaultInterface();


   VARIANT ErrPos;
   VARIANT Msg;

      // Set formula string
   pFormula->SetFormulaStr(WideString(Formula->Text));

      // Set data adapter
   pFormula->SetDataAdapter(pIAdapter);

      // Call Calculate method of IFormula interface
   Variant v = pFormula->Calculate(&ErrPos,&Msg);

      // Check Error
   If((int)Variant(ErrPos) == 0)
     Result->Text = (AnsiString)v;
```

- **Using automation interface by VB Client**

```
    Dim objFormula As Object
    Dim objAdapter As Object
    Dim strFormula As String
    Dim vValue As Variant
    Dim Err As Variant
    Dim Msg As Variant

     ` Create EasyFormula automation object
    Set objFormula = CreateObject("EasyFormula.Formula")

   ` Create DataAdapter object
    Set objAdapter = CreateObject("TestAdapter.MyAdapter")

     ` Get formula string from form
    strFormula = txtFormula.Text
    ' Set formula string into IFormula interface
    objFormula.SetFormulaStr (strFormula)

    ' Set DataAdapter interface into IFormula
    objFormula.SetDataAdapter objAdapter

    ' Call Calculate method of IFormula
    vValue = objFormula.Calculate(Err, Msg)
    ' Check the error and show result
    If Err > 0 Then
        txtResult = Msg
    Else
        txtResult = vValue
    End If
    Set objFormula = Nothing
```

5.2. MFC

Using Native C++ library

- **Implement the data adapter**

```
_variant_t DataAdapter::GetOperantValue(const char*   pOperantName)
{
// This is the method to get client application data, in this example,
// 2 local data {Name} and {Tel} is provided.

      _variant_t Value;

    if(!strcmp(OperantName,"Name"))
        Value = _variant_t ("David");
    else if(!strcmp(OperantName,"Tel"))
        Value = _variant_t ("123456");
    else
        Value = _variant_t("");

    return(Value);
}

bool DataAdapter::IsKnown(const char    *pOperantName)
{
    bool isKnown = false;

    if(!strcmp(OperantName,"Name"))
         isKnown = true;
    else if(!strcmp(OperantName,"Tel"))
         isKnown = true;
    else
         isKnown = false;

    return(isKnown);
}
```

```
string DataAdapter::GetFormulaByName(const char *pFormulaName)
{
      // In this example, the data adapter hardcoded a formula
      // for ‘Contact’ as {Name}+’ ’+{Tel}
      //  In your real project, you may store the formula into
      //  database, in that case this method should check against
      //  database to fetch formula string

      string strFormula(””);

      if((!strcmp(OperantName,"Contact"))
            strFormula=string("{Name}")+string("+':'
      +")+string("{Tel}");

      return (strFormula);
}
```

- **Client Program**

```
// Update data member
UpdateData();

// Create a EasyFormula object with empty formula
EFormula Fm;

_variant_ v;

int pos;
string msg;

int cnt;

// Set formula string into EasyFormula object
Fm.SetFormulaStr((LPCTSTR)m_strFormula);

// Call Calculate method get result
v = Fm.Calculate(pos, msg);

// check error and assign to data member
if(pos==0 && v.vt!=VT_NULL)
{
      m_vResult =v;
}

// Show result
UpdateData(false);
```

Using COM Automation object

- **Implement the data adapter**

IDL Definition

```
import "ocidl.idl";
import "oaidl.idl";
[
	object,
	uuid(69DB05FC-34B3-4209-8249-E7FB05225A89),
	dual,
	nonextensible,
	helpstring("IDataAdapter Interface"),
	pointer_default(unique)
]
interface IDataAdapter : IDispatch{
	[id(1), helpstring("method GetOperantValue")] HRESULT 
GetOperantValue([in] BSTR OperantName,[out,retval] VARIANT* Value);
	[id(2), helpstring("method GetFormulaByName")] HRESULT 
GetFormulaByName([in] BSTR FormulaName,[out,retval] BSTR* Value);
	[id(3), helpstring("method IsKnown")] HRESULT IsKnown([in] BSTR 
OperantName, [out,retval] VARIANT * bKnown);
};
[ uuid(768EA79E-1F0C-4BA2-B216-35BA1224C5EA), version(1.0) ]
library Adapter
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");
	[
		uuid(8C1AA170-C922-4200-8C0D-0A1802B182C6),
		helpstring("MyAdapter Class")
	]
	coclass MyAdapter
	{
		[default] interface IDataAdapter;
	};
};
```

Adapter Class Declaration

```
class ATL_NO_VTABLE CMyAdapter : 
	public CComObjectRootEx<CComSingleThreadModel>,
	public CComCoClass<CMyAdapter, &CLSID_MyAdapter>,
	public IDispatchImpl<IDataAdapter, &IID_IDataAdapter, 
&LIBID_Adapter, /*wMajor =*/ 1, /*wMinor =*/ 0>
{
public:
	CMyAdapter()
	{
	}

DECLARE_REGISTRY_RESOURCEID(IDR_MYADAPTER)


BEGIN_COM_MAP(CMyAdapter)
	COM_INTERFACE_ENTRY(IDataAdapter)
	COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()
```

```
      DECLARE_PROTECT_FINAL_CONSTRUCT()

      HRESULT FinalConstruct()
      {
            return S_OK;
      }

      void FinalRelease()
      {
      }

public:
      STDMETHOD(GetOperantValue)(/*in]*/ BSTR
OperantName,/*[out,retval]*/VARIANT* Value);
      STDMETHOD(GetFormulaByName)(/*[in]*/ BSTR
FormulaName,/*[out,retval]*/BSTR* bstrFormula);
      STDMETHOD(IsKnown)(/*[in]*/ BSTR OperantName,
/*[out,retval]*/VARIANT* bKnown);

};
```

Adapter Class Implementation:

```
STDMETHODIMP CMyAdapter::GetOperantValue(/*in]*/ BSTR
OperantName,/*[out,retval]*/VARIANT* Value)
{
      AFX_MANAGE_STATE(AfxGetStaticModuleState());

      _bstr_t rs;
      if(_bstr_t("Name") == _bstr_t(OperantName))
            rs = _bstr_t("David");
      else if(_bstr_t("Tel") == _bstr_t(OperantName))
            rs = _bstr_t("123456");
      else if(_bstr_t("Address") == _bstr_t(OperantName))
            rs = _bstr_t("165 Roanoke Dr.");
      else
            rs = _bstr_t("");

      *Value = _variant_t(rs).Detach();

      return S_OK;
}
STDMETHODIMP CMyAdapter::GetFormulaByName(/*[in]*/ BSTR
FormulaName,/*[out,retval]*/BSTR* bstrFormula)
{
      AFX_MANAGE_STATE(AfxGetStaticModuleState());

      _bstr_t rs;
      if(_bstr_t("Contact") == _bstr_t(FormulaName))
            rs = _bstr_t("{Name}")+ _bstr_t("+\" \"+") +
_bstr_t("{Tel}")+ _bstr_t("+\" \"+") +_bstr_t("{Address}");
      else
            rs = _bstr_t("");
      // TODO: Add your implementation code here
      *bstrFormula = rs.Detach();
      return S_OK;
}

STDMETHODIMP CMyAdapter::IsKnown(/*[in]*/ BSTR OperantName,
/*[out,retval]*/VARIANT* bKnown)
{
      AFX_MANAGE_STATE(AfxGetStaticModuleState());
      bool rs;
      if(_bstr_t("Name") == _bstr_t(OperantName))
            rs = true;
      else if(_bstr_t("Tel") == _bstr_t(OperantName))
            rs = true;
      else if(_bstr_t("Address") == _bstr_t(OperantName))
            rs = true;
      else
            rs = false;

      *bKnown = _variant_t(rs).Detach();
      // TODO: Add your implementation code here

      return S_OK;
}
```

- **Using automation interface by VB Client**

```
    Dim objFormula As Object
    Dim objAdapter As Object
    Dim strFormula As String
    Dim vValue As Variant
    Dim Err As Variant
    Dim Msg As Variant

    ‘ Create EasyFormula automation object
    Set objFormula = CreateObject("EasyFormula.MFormula")

   ‘ Create DataAdapter object
    Set objAdapter = CreateObject("Adapter.MyAdapter ")

    ‘ Get formula string from form
    strFormula = txtFormula.Text

    ‘ Set formula string into IFormula interface
    objFormula.SetFormulaStr (strFormula)

    ‘ Set DataAdapter interface into IFormula
    objFormula.SetDataAdapter objAdapter


    ‘ Call Calculate method of IFormula
    vValue = objFormula.Calculate(Err, Msg)

    ‘ Check the error and show result
    If Err > 0 Then
        txtResult = Msg
    Else
```

```
        txtResult = vValue
    End If
    Set objFormula = Nothing
```

6. Appendix

File list in each package

EasyFormula for VCL

You need Microsoft Visual Studio installed.

File	Description
EasyFormula_VCL.dll	Native C++ dynamic library
EasyFormula_VCL.lib	Native C++ link file
EFormula.h	C++ class header file
VCLCOM.dll	Automation COM library
EasyFormula.tlb	Type library file for COM object

EasyFormula for MFC

You need Borland C++ builder or Delphi installed

File	Description
ESFMFC.dll	Native C++ dynamic library
ESFMFC.lib	Native C++ link file
EFormula.h	C++ class header file
EasyFormula_MFC.dll	Automation COM library
EasyFormula_MFC.idl	IDL file for COM library