고급 COM 기술의 활용 (I)

(Using Advanced COM Techniques I.)

이번 장에서는 비교적 고급이라고 할 수 있는 COM 에서 컬렉션을 구현하는 방법과 콜백 함수를 이용하여 인터페이스간 통신을 하는 방법, 그리고 연결점(Connection Point) 인터페이스를 사용하여 이벤트를 구현하는 방법을 예제를 통해 익히도록 한다.

컬렉션 객체의 구현

델파이의 컴포넌트들 중에는 여러 개의 서브 아이템을 소유하는 클래스들이 많다. 대표적인 것이 TStringList 로 이 클래스는 리스트 박스, 콤보 박스, 메모 컴포넌트 등에서 Lines 또는 Items 프로퍼티로 접근할 수 있도록 되어 있다. TStringList 는 문자열들을 인덱스로 접근할 수 있도록 허용한 일종의 컬렉션 클래스라고 말할 수 있다.
이 밖에도 TList, TTreeNodes 등의 클래스가 컬렉션의 형태로 이루어져 있다.

- IEnumXXXX 인터페이스

그렇다면, COM 에서 이런 컬렉션을 구현하려면 어떻게 하면 될까 ? COM 에서 컬렉션을 구현하기 위해서는 IEnumXXXX 라는 인터페이스를 구현해야 한다.
이런 IEnumXXXX 와 같은 인터페이스를 열거 인터페이스라고 하며, 대표적으로 구현된 예는 이벤트를 구현하기 위해 사용되는 IConnectionPoint 와 IConnectionPointContainer 에서 이용하는 IEnumConnectionPoints, IEnumConnections 인터페이스를 들 수 있다.
IEnumXXXX 인터페이스는 진정한 인터페이스라고는 할 수 없고, 모든 환경 인터페이스의 요청을 지시하기 위한 문서화 도구(documentation device)이다. 기본적으로 이런 열거형 인터페이스는 Next, Skip, Reset, Clone 이라는 4 가지 메소드를 지원한다. 다음의 코드는 예제에서 사용할 IEnumVariant 인터페이스의 선언부이다.

```
IEnumVariant = interface(IUnknown)
  ['{00020404-0000-0000-C000-000000000046}']
  function Next(celt: Longint; out elt;
    pceltFetched: PLongint): HResult; stdcall;
  function Skip(celt: Longint): HResult; stdcall;
  function Reset: HResult; stdcall;
  function Clone(out Enum: IEnumVariant): HResult; stdcall;
```

```
end;
```

클라이언트는 항목을 가지고 있는 배열을 할당하고, 이것을 배열의 크기와 함께 Next 메소드로 전달한다. 여기에서 배열의 이름이 들어가는 파라미터가 elt 이다. celt 는 아마도 'count of elements of T'의 약자일 것으로 생각되는데, 여기서 T 는 데이터 형을 의미하며 배열의 크기를 지정한다. pceltFetched 파라미터는 카운터 변수로 사용된다. 이런 클라이언트의 요구가 있으면 서버는 지정된 항목의 수만큼 배열을 채우고자 시도할 것이고, 카운터에 실제로 들어간 항목의 수를 반환하게 된다.
Reset 메소드는 클라이언트가 첫번째 항목으로부터 다시 열거를 시작하게 할 때 사용되며, 열거하는 도중에 항목을 건너뛸 때에는 Skip 메소드를 사용한다. 마지막으로 Clone 메소드는 해당되는 IEnumXXXX 인터페이스에게 현재의 열거자 객체의 복사본을 생성해준다.
그러면 실제로 예제를 통해 이를 익히도록 하자.

- 컬렉션을 구현한 자동화 서버

이번에 작성할 예제 자동화 서버는 필자가 인터넷에서 구한 Coll_Demo 라는 프로그램을 참고하여 작성한 것인데, 아쉽게도 작성자에 대한 정보가 없어서 이를 구체적으로 알리지 못한다는 것을 미리 알려둔다.
이 예제는 파일을 찾아주는 역할을 하는 자동화 서버를 제작하는데, 해당되는 파일들의 정보를 IFileObject 라는 인터페이스에 저장하고 이들의 컬렉션을 관리하는 IFileObjects 인터페이스와 메인 인터페이스 역할을 하는 IFileFinder 인터페이스의 3 가지 인터페이스를 이용한다.
예제를 직접 작성하기 전에, 먼저 이들 인터페이스를 디자인하도록 한다.

```
IFileObject = interface(IDispatch)
  function Get_Name: WideString; safecall;
  function Get_FullName: WideString; safecall;
  function Get_Size: Integer; safecall;
  property Name: WideString read Get_Name;
  property FullName: WideString read Get_FullName;
  property Size: Integer read Get_Size;
end;
```

가장 기본적인 요소가 되는 인터페이스가 IFileObject 이다. 3 가지 프로퍼티를 지원하는데, 이들은 모두 읽기 전용이다. 파일의 전체 경로를 포함한 이름을 저장하는 FullName, 패스 정보를 제외한 파일 이름인 Name, 파일의 크기 정보인 Size 프로퍼티를 가진다.

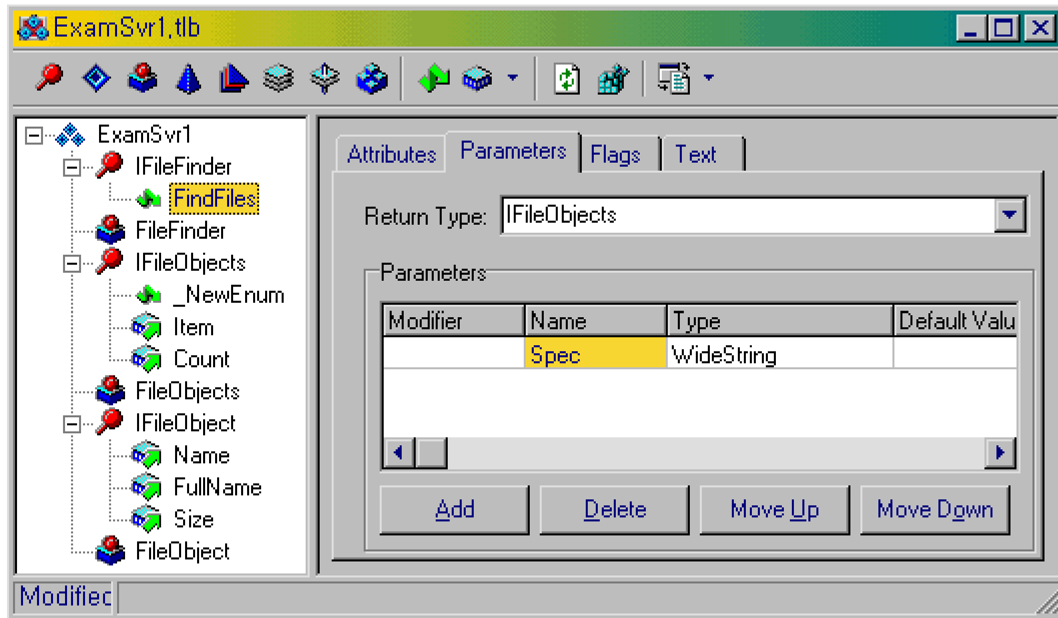
```
  IFileObjects = interface(IDispatch)
    function _NewEnum: IUnknown; safecall;
    function Get_Item(Index: Integer): IFileObject; safecall;
    function Get_Count: Integer; safecall;
    property Item[Index: Integer]: IFileObject read Get_Item;
    property Count: Integer read Get_Count;
  end;
```

IFileObjects 인터페이스는 기본적으로 IFileObject 인터페이스를 요소로 한 컬렉션 역할을 하게 된다. _NewEnum 메소드가 여기서 중요한 역할을 하는데, IEnumVariant 인터페이스를 구현한 클래스를 생성해서 여기에 접근할 수 있도록 IUnknown 인터페이스로 형변환하여 결과값을 리턴한다. 참고로 IEnumVariant 인터페이스는 ActiveX.pas 유닛에 선언되어 있으므로 이를 따로 선언할 필요는 없다.
Item 프로퍼티는 인덱스를 가진 프로퍼티로, 해당 인덱스의 IFileObject 인터페이스를 반환한다. Count 프로퍼티는 전체 IFileObject 인터페이스의 수를 반환한다.

```
  IFileFinder = interface(IDispatch)
    function FindFiles(const Spec: WideString): IFileObjects; safecall;
end;
```

IFileFinder 인터페이스는 FindFiles 메소드를 호출할 때, 파라미터로 'c:₩*.*'와 같이 파일을 열거할 조건을 문자열로 넘겨주면 IFileObjects 인터페이스를 반환하는 역할을 한다.
그러면, 이들을 실제로 구현해 보도록 하자.
먼저 File|New 메뉴의 ActiveX 탭에서 ActiveX Library 아이콘을 더블 클릭하여 새로운 DLL 프로젝트를 시작한다. 그리고, 자동화 객체를 선언하도록 한다. File|New 메뉴의 ActiveX 탭에서 Automation Object 아이콘을 더블 클릭하고 클래스 이름으로 FileFinder를 입력하고 OK 버튼을 클릭하면 IFileFinder 인터페이스가 타입 라이브러리 에디터에 추가될 것이다. 마찬가지로 다시 File|New 메뉴의 AcitveX 탭에서 Automation Object 아이콘을 더블 클릭하고 클래스 이름을 FileObjects 를 입력하고 OK 를 클릭하고, 다시 한번 반복하여 FileObject 를 추가한다. 이렇게 하면 IFileFinder, IFileObjects, IFileObject 인터페이스와 이들에 대한 CoClass 들이 선언될 것이다.
그러면, 인터페이스에 메소드와 프로퍼티를 추가해보자.
IFileFinder 인터페이스를 선택하고 New Method 버튼을 클릭한다. 메소드의 이름을 FindFiles 로 설정하고, Parameters 탭을 선택한다. Return Type 콤보 박스에서 IFileObjects 인터페이스를 선택하고, Add 버튼을 클릭하여 파라미터를 추가한다. 추가된

파라미터의 Name 을 'Spec'으로 설정하고, Type 을 콤보 박스에서 WideString 으로 설정한다.
IFileObjects 인터페이스에서는 _NewEnum 메소드와 Item, Count 프로퍼티를 추가해야 한다. 마찬가지 방법으로 추가하면 되는데, 이때 Item 과 Count 프로퍼티는 모두 읽기 전용으로 설정해야 하므로 Attributes 탭의 Invoke Kind 콤보 박스에서 property Get 을 선택해야 한다. Item 프로퍼티는 IFileObject, Count 프로퍼티는 Integer 로 Type 을 설정한다.
_NewEnum 메소드는 Parameters 탭에서 Return Type 으로 IUnknown 을 선택한다.
이제 마지막으로 IFileObject 인터페이스의 프로퍼티를 추가하도록 하자. New Properties 버튼을 클릭하여 Name, FullName, Size 프로퍼티를 추가한다. 이들은 모두 읽기 전용이므로 Attributes 탭의 Invoke Kind 콤보 박스에서 property Get 을 선택해야 한다. Name, FullName 프로퍼티는 WideString, Size 프로퍼티는 Integer 로 Type 을 설정한다. 이렇게 해서 작성된 타입 라이브러리 에디터의 형태는 다음과 같을 것이다.



타입 라이브러리 에디터를 닫으면 자동화 객체에 대한 유닛이 추가되는데, 이들을 모두 적당한 이름으로 저장하도록 하자.
먼저 IFileFinder 인터페이스를 먼저 구현하도록 하자. FindFiles 메소드만 구현하면 되는데, FindFiles 메소드는 Spec 파라미터의 내용을 바탕으로 IFileObjects 인터페이스를 반환하는 역할을 하게 되므로 IFileObjects 인터페이스를 구현한 유닛을 uses 절에 추가해야 한다. 여기서는 U2_ExamSvr1Impl.pas 유닛을 추가한다.
FindFiles 메소드는 다음과 같이 구현한다.

```
function TFileFinder.FindFiles(const Spec: WideString): IFileObjects;
```

```
begin
  Result := TFileObjects.Create(Spec);
end;
```

IFileObjects 인터페이스의 구현 방법을 알아보기 전에, 먼저 구현하기 쉬운 IFileObject 인터페이스를 먼저 구현하도록 하자.
IFileObject 인터페이스를 구현한 TFileObject 클래스에 프로퍼티의 정보를 저장할 FName, FFullName, FSize 변수를 private 섹션에 추가하고, constructor 인 Create 메소드를 public 섹션에 다음과 같이 추가한다.

```
  TFileObject = class(TAutoObject, IFileObject)
  private
    FFullName: String;
    FName: String;
    FSize: Integer;
  public
    constructor Create(const sr: TSearchRec);
  protected
    function Get_FullName: WideString; safecall;
    function Get_Name: WideString; safecall;
    function Get_Size: Integer; safecall;
  end;
```

이들 메소드는 다음과 같이 비교적 쉽게 구현할 수 있다. 참고로 TFileObject 클래스의 constructor 는 IFileObjects 인터페이스에 의해서 호출되므로, 파라미터인 sr 등을 조작할 필요는 없다.

```
constructor TFileObject.Create(const sr: TSearchRec);
begin
  inherited Create;
  FFullName := sr.Name;
  FName := ExtractFilename(FFullName);
  FSize := sr.Size;
end;

function TFileObject.Get_FullName: WideString;
```

```
begin
  Result := FFullName;
end;

function TFileObject.Get_Name: WideString;
begin
  Result := FName;
end;

function TFileObject.Get_Size: Integer;
begin
  Result := FSize;
end;
```

이제 가장 구현하기 어려운 IFileObjects 인터페이스를 구현하도록 한다.
IFileObjects 인터페이스를 구현하기 위해서는 IEnumVariant 인터페이스를 구현해 주어야 한다. 이를 위해서 uses 절에 ActiveX.pas 유닛을 추가한다. 그 밖에도 이들을 구현하기 위해서 여러가지 함수를 사용하게 되는데 여기에 필요한 SysUtils.pas, Windows.pas 유닛과 TList 클래스를 사용하기 위해 Classes.pas 유닛을 uses 절에 추가해야 된다. 그리고, 앞에서 구현한 TFileObject 클래스를 이용하게 되므로 IFileObject 인터페이스를 구현한 유닛을 uses 절에 추가한다 (여기서는 U3_ExamSvr1Impl.pas).
먼저, IEnumVariant 인터페이스를 구현할 TEnumVariant 클래스를 다음과 같이 선언한다.

```
  TEnumVariant = class(TInterfacedObject, IEnumVariant)
  private
    FIndex: Integer;
    FList: TList;
    FParent: IUnknown;
  protected
    function Next(celt: Longint; out elt; pceltFetched: PLongint): HResult; stdcall;
    function Skip(celt: Longint): HResult; stdcall;
    function Reset: HResult; stdcall;
    function Clone(out enum: IEnumVariant): HResult; stdcall;
  public
    constructor Create(aParent: IUnknown; aList: TList);
  end;
```

여기서 열거자 역할을 하는 메소드는 protected 섹션에 선언된 4 개의 메소드이다. 내부적으로 사용하기 위해 FIndex, FParent, FList 변수를 private 섹션에 추가하고, public 섹션에 constructor 로 Create 메소드를 추가한다.
IFileObjects 인터페이스를 선언하는 TFileObjects 클래스에는 TList 클래스의 객체를 저장할 FList 변수를 private 섹션에 추가하고 constructor 와 destructor 로 사용할 Create, Destroy 메소드를 public 섹션에 다음과 같이 추가한다.

```
  TFileObjects = class(TAutoObject, IFileObjects)
  private
    FList: TList;
  public
    constructor Create(const aFileSpec: String);
    destructor   Destroy; override;
  protected
    function _NewEnum: IUnknown; safecall;
    function Get_Count: Integer; safecall;
    function Get_Item(Index: Integer): IFileObject; safecall;
  end;
```

그러면, 먼저 TEnumVariant 클래스를 구현해 보도록 하자. constructor 인 Create 메소드에서는 TEnumVariant 클래스의 Parent 가 되는 클래스와 열거할 항목을 저장할 TList 클래스 변수의 값을 다음과 같이 설정한다.

```
constructor TEnumVariant.Create(aParent: IUnknown; aList: TList);
begin
  inherited Create;
  FParent := aParent;
  FList := aList;
end;
```

그리고, 가장 중요한 Next 메소드는 다음과 같이 구현한다.

```
function TEnumVariant.Next(celt: Longint; out elt; pceltFetched: PLongint): HResult;
type
  TVariantArray = packed array[0..0] of OleVariant;
```

```
var
  i: Integer;
begin
  for i := 0 to celt - 1 do
  begin
    VariantClear(TVariantArray(elt)[i]);
  end;
```

앞에서도 설명한 바 있지만, 여기서 elt 는 배열 이름을 가리키며 celt 는 배열의 항목의 수가 된다. 그러므로, 이 코드는 선언한 TVariantArray 라는 OleVariant 의 배열의 값을 초기화하는 역할을 한다.

```
  i := 0;
  while (celt > 0) and (FIndex < FList.Count) do
  begin
    TVariantArray(elt)[i] := IUnknown(FList[FIndex]) as IDispatch;
    Inc(i);
    Dec(celt);
    Inc(FIndex);
  end;
```

이 코드는 FList 의 마지막 항목까지의 객체를 IDispatch 로 TVariantArray 배열에 저장하는 역할을 한다. 즉, TVariantArray 배열의 내용과 FList 변수의 내용을 동기화하는 코드이다.

```
  try
    if Assigned(pceltFetched) then
      pceltFetched^ := i;
  except
  end;
```

제대로 작업이 끝났으면 pceltFetched 파라미터의 값을 설정한다.

```
  if celt = 0 then
    Result := S_OK
  else
```

```
    Result := S_FALSE;
end;
```

마지막으로 결과값을 반환하면 된다.
Reset, Skip, Clone 메소드는 구현하기가 비교적 쉽다. 다음과 같이 구현하면 된다.

```
function TEnumVariant.Reset: HResult;
begin
  FIndex := 0;
  Result := S_OK;
end;

function TEnumVariant.Skip(celt: Longint): HResult;
begin
  while (celt > 0) and (FIndex < FList.Count) do
  begin
    Dec(celt);
    Inc(FIndex);
  end;
  if celt = 0 then
    Result := S_OK
  else
    Result := S_FALSE;
end;

function TEnumVariant.Clone(out enum: IEnumVariant): HResult;
var
  r: TEnumVariant;
begin
  r := TEnumVariant.Create(FParent, FList);
  r.FIndex := FIndex;
  enum := r;
  Result := S_OK;
end;
```

이렇게 함으로써 FList 의 아이템에 IDispatch 인터페이스로서 IFileObject 인터페이스를 저

장할 수 있게 되었다. FList 에내용을 저장하고, 이들에 접근할 때 자동으로 IEnumVariant 인터페이스의 메소드를 호출하여 사용하게 된다.
그러면, FList 를 이용하여 IFileObject 인터페이스객체를 관리하는 TFileObjects 클래스를 구현하도록 하자.
가장 중요한 것이 constructor 인 Create 메소드이다. 이 메소드는 IFileFinder 인터페이스의 FindFiles 메소드에 의해서도 호출되며, 실제로 FindFiles 메소드에서 파라미터로 사용된 문자열을 바탕으로 파일을 검색해서 파일의 내용을 바탕으로 TFileObject 클래스를 생성하고, IFileObject 인터페이스를 FList 에 저장한다.
Create 메소드를 다음과 같이 구현한다.

```
constructor TFileObjects.Create(const aFileSpec: String);
var
  r: Integer;
  sr: TSearchRec;
  Obj: IFileObject;
begin
  inherited Create;
  FList := TList.Create;
  r := FindFirst(aFileSpec, faAnyFile, sr);
  try
    while r = 0 do
    begin
      Obj := TFileObject.Create(sr);
      Obj._AddRef;
      FList.Add(Pointer(Obj));
      r := FindNext(sr);
    end;
  finally
    SysUtils.FindClose(sr);
  end;
end;
```

그다지 어렵지 않은 코드이므로 자세한 설명은 생략한다. 주의해야할 부분은 Obj 변수를 IFileObject 형으로 선언한 뒤에 이를 TFileObject.Create 메소드에 찾은 파일의 TSearchRec 데이터 형 데이터를 저장한 sr 변수를 파라미터로 사용하여 호출하는 부분과, IFileObject 인터페이스를 사용하기 때문에 _AddRef 메소드를 호출한 부분이다.

이 부분에서 실수를 하면 COM 의 핵심 부분이라고 할 수 있는 참조 계수관리에 실패하게 된다.

참고: COM 참조계수 관리

델파이는 기본적으로 참조계수 관리를 자동으로 해준다. 그렇지만, 여기에는 일반적인 참조 계수와는 다른 델파이 만의 규칙이 있기 때문에, 이를 잘 숙지하고 있어야 한다.
만약 변수의 데이터 형으로 IUnknown 을 이용한 경우에는 델파이가 자동으로 참조계수 관리를 하므로 AddRef 나 Release 메소드를 사용하면 안된다.
이때 주의할 것은 어떤 식으로 COM 객체를 사용하는지 여부에 따라 참조계수가 증가하기도 하고, 변화를 주지 않을 수도 있다. 다음의 예제 코드를 참고하기 바란다.

```
var
  MyIxxxVariable: ISomeCOMInterface;
  ....
  MyIxxxVariable := TMyCOMObject.Create;          //내부적으로 AddRef 가 호출된다.
  SomeAPIFunc(MyIxxxVariable);
```

이 경우에는 내부적으로 ISomeComInterface 에 대한 참조계수가 증가하기 때문에, 함수를 호출하는데 문제가 없다. 그렇지만 다음의 코드는 사정이 다르다.

```
var
  MyDelphiVariable:  TMyCOMObject;
  ....
  MyDelphiVariable := TMyCOMObject.Create;     //AddRef 가 호출되지 않는다.
  SomeAPIFunc(MyDelphiVariable);
```

이 경우에는 API 함수의 파라미터로 직접 COM 객체를 사용해도 형변환을 하기 때문에 문제가 없지만 AddRef 를 호출하지 않기 때문에, 문제가 된다.
이를 해결하기 위해서는 다음과 같은 코드를 사용하면 된다.

```
var
  MyDelphiVariable:  TMyCOMObject;
  ....
  MyDelphiVariable := TMyCOMObject.Create;
  SomeAPIFunc(MyDelphiVariable as ISomeCOMInterface);
```

as 연산자에 의해 AddRef 가 호출되므로 잘 동작하게 된다.

그런데, IFileObject 인터페이스의 _AddRef 를 호출한 이유는 델파이의 FList 아이템으로 추가할 경우 이들에 의해 인터페이스가 추가, 삭제될 때 참조계수의 변화를 주어야 하기 때문이다. 그러므로, 나중에 IFileObject 인터페이스를 아이템에서 삭제할 경우 _Release 메소드를 호출해야 한다.

그리고, FList 의 Add 메소드를 호출할 때 IFileObject 인터페이스를 저장한 Obj 변수를 Pointer 형으로 형변환하여 저장하면 된다. 즉, 이렇게 FList 에 아이템을 추가할 때 Pointer 형으로 형변환하는 것으로는 _AddRef 가 호출되지 않기 때문에, 그 이전에 _AddRef 메소드를 명시적으로 호출하는 것이다.

TFileObjects 클래스의 _NewEnum 메소드는 다음과 같이 구현한다.

```
function TFileObjects._NewEnum: IUnknown;
begin
  Result := TEnumVariant.Create(Self, FList) as IUnknown;
end;
```

즉, 열거를 담당하는 aParent 로 Self(TFileObjects)를 넘기고 열거할 항목을 저장할 배열로 FList 변수를 지정하는 것이다.

TFileObjects 클래스의 IFileObject 인터페이스를 항목으로 제공하는 Item 프로퍼티와 항목의 수를 제공하는 Count 프로퍼티는 Get_Item, Get_Count 메소드로 다음과 같이 구현할 수 있다.

```
function TFileObjects.Get_Count: Integer;
begin
  Result := FList.Count;
end;

function TFileObjects.Get_Item(Index: Integer): IFileObject;
begin
  Assert((Index > 0) and (Index <= FList.Count));
  Result := IFileObject(FList[Index - 1]);
  Result._AddRef;
end;
```

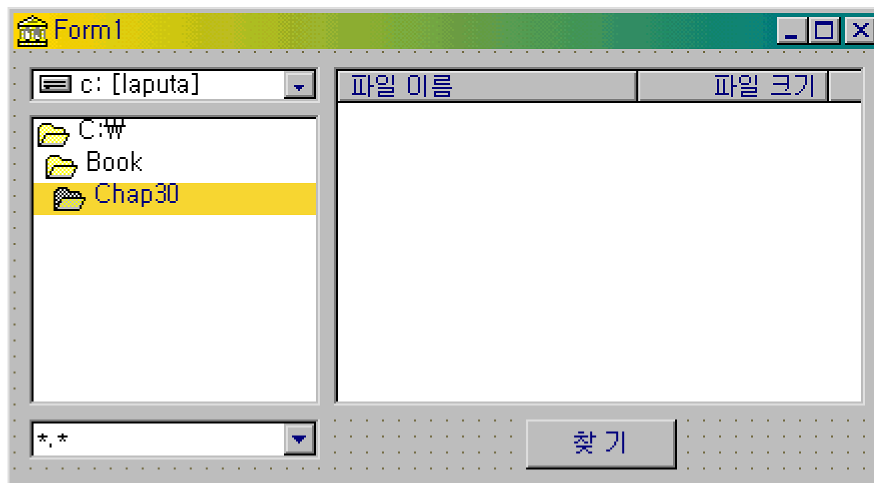
여기에서도 주의할 것은 IFileObject 인터페이스를 이용하는 클라이언트를 위해서 _AddRef 를 호출한다는 것이다.
마지막으로 Destroy 메소드를 다음과 같이 구현하면 된다.

```
destructor  TFileObjects.Destroy;
var
  i: Integer;
begin
  for i := 0 to FList.Count - 1 do
    IUnknown(FList[i])._Release;
  FList.Free;
  inherited Destroy;
end;
```

Destroy 메소드는 이와 같이 FList 에 저장된 IFileObject 인터페이스를 모두 _Release 메소드를 호출하여 해제하는 것이 중요하며, 동시에 생성한 FList 클래스를 해제하면 된다.
이것으로 TFileObjects 클래스의 구현이 모두 끝났다. 쉽지 않은 내용이지만, IEnumXXXX 인터페이스에 대해서는 연결점에 대해 설명하면서 다시 다루게 될 것이다.
그러면, 프로젝트를 컴파일하고 Run|Register ActiveX Server 메뉴를 선택하여 자동화 서버를 등록하도록 한다.

● 클라이언트 어플리케이션의 제작

그러면, 제작한 자동화 서버를 이용해서 디렉토리 브라우저 기능을 하는 클라이언트 어플리케이션을 하나 만들어 보자.
먼저 폼 위에 TDriveComboBox, TDirectoryListBox, TComboBox, TListView, TButton 컴포넌트를 하나씩 올려 놓자.
DriveComboBox1 의 DirList 프로퍼티는 DirectoryListBox1 으로 설정한다. 그리고 ComboBox1 의 Items 프로퍼티 에디터를 이용하여 *.*, *.txt, *.exe, *.dll 을 기본적으로 추가하여 이들을 이용하거나 직접 입력이 가능하도록 하자. 버튼 컴포넌트는 Caption 프로퍼티를 '찾 기'로 설정한다.
ListView1 컴포넌트는 먼저 ViewStyle 프로퍼티를 vsReport 로 설정한다. 그리고, Columns 프로퍼티 에디터를 이용하여 2 개의 새로운 컬럼을 추가하고 이들의 Caption 프로퍼티를 '파일 이름'과 '파일 크기'로 설정한다. 이들의 Alignment 프로퍼티를 각각 alLeft, alRight 로 설정하고 다음 그림과 같이 적절하게 Width 프로퍼티를 조절한다.



ExamSvr1 의 타입 라이브러리를 사용해야 하므로, uses 절의 ExamSvr1_TLB.pas 유닛을 추가하도록 한다.
그리고 전역 변수로 IFileFinder 인터페이스 변수를 다음과 같이 선언한다.

```
var
  Form1: TForm1;
  FileFinder: IFileFinder;
```

그리고, 폼의 OnCreate 이벤트 핸들러에서 FileFinder 변수에 early 바인딩을 이용하여 값을 대입한다.

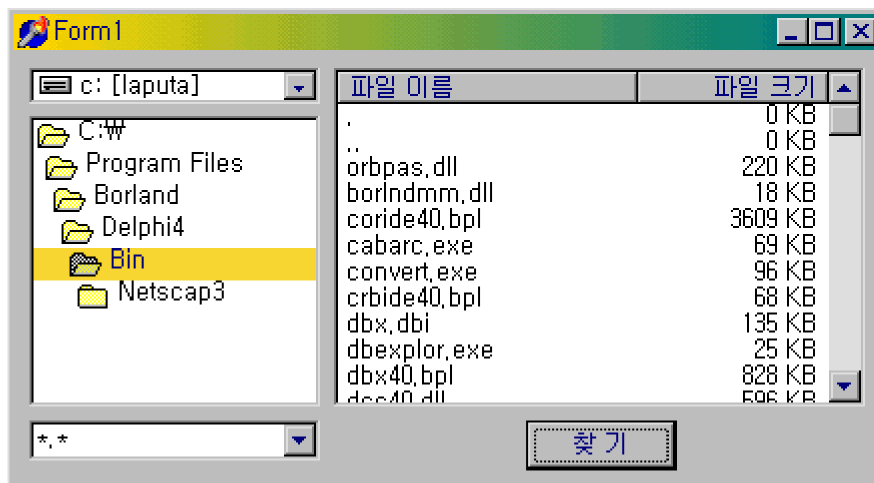
```
procedure TForm1.FormCreate(Sender: TObject);
begin
  FileFinder := CoFileFinder.Create;
end;
```

마지막으로 Button1 의 OnClick 이벤트 핸들러를 다음과 같이 구현하면 클라이언트 어플리케이션은 완성된다.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  FileObjects: IFileObjects;
  FileObject: IFileObject;
  i: Integer;
  FileItem: TListItem;
```

```
begin
  ListView1.Items.Clear;
  FileObjects
    := FileFinder.FindFiles(DirectoryListBox1.Directory + '₩' + ComboBox1.Text);
  for i := 1 to FileObjects.Count do
  begin
    FileObject := FileObjects.Item[i];
    FileItem := ListView1.Items.Add;
    FileItem.Caption := FileObject.Name;
    FileItem.SubItems.Add(IntToStr(FileObject.Size div 1024) + ' KB');
  end;
end;
```

그러면 클라이언트 어플리케이션을 컴파일하고 실행해보자. 그리고, 디렉토리를 선택한 후 콤보 박스에 파일 찾기에서 입력할 수 있는 여러 가지 옵션을 주고 '찾 기' 버튼을 클릭하면 해당되는 파일 들을 리스트 뷰에서 볼 수 있을 것이다.



COM 에서의 콜백 함수 활용

COM 콜백 인터페이스를 활용하면 COM 서버 컴포넌트가 클라이언트 어플리케이션에 존재하는 객체의 메소드를 호출할 수 있게 된다. 이때 콜백은 이렇게 서버에 중요한 변화가 나타났을 때마다 클라이언트에 이를 알리는 역할을 하게 된다.

콜백에 대해서는 후킹 예제와 함께 제 7 부에서 더 자세하게 다루게 될 것이다.

이러한 콜백을 잘 활용하면 멀티-유저 환경의 클라이언트-서버 데이터베이스 어플리케이션에서 많은 수의 클라이언트에 데이터 변경 등에 대한 여러가지 조작을 할 수 있게 된다.

연결점(Connection point) 방법론

실제로 연결점에 대한 개념을 이해하는 것은 그다지 어려운 것이 아니다. 그렇지만 이를 어렵게 느끼게 하는 것은 몇 가지의 기술적인 용어가 많이 나오기 때문이다.
연결점의 기본적인 개념은 클라이언트 객체와 서버 객체가 서로 표준 프로토콜을 이용해서 쉽게 통신을 하게 만들자는 것이다. 이때 프로토콜에서 클라이언트 객체가 서버 객체에게 특정 콜백 인터페이스에 대해서 알고 있는지 묻게 되고, 서버 객체는 여기에 대해 긍정, 또는 부정의 반응을 할 수 있다. 서버 객체가 긍정의 답변을 하게 되면, 클라이언트는 자신의 콜백 인터페이스를 제공하고, 이를 이용해서 통신을 할 수 있도록 요청하게 된다. 이때부터 서버와 클라이언트는 서로의 메소드를 호출할 수 있게 된다. 이 개념을 이용하면 대단히 유연한 어플리케이션을 개발할 수 있게 된다. 클라이언트는 특정 콜백 인터페이스를 지원하는 특정 서버 객체에 대해 자세히 알 필요가 없으며, 단지 어느 서버 객체에나 자신이 구현할 수 있는 인터페이스를 지원하는지 여부 만을 알아보고 서버가 긍정의 답변을 할 경우에만 통신을 하면 된다.
서버의 관점에서 볼 때 이러한 콜백 인터페이스를 outgoing 인터페이스라고 한다. 이는 인터페이스가 클라이언트에서 구현되며, 서버에 의해 사용되기 때문이다. 반대로 서버에서 구현되고 클라이언트에 의해 사용되는 인터페이스는 incoming 인터페이스라고 한다. 최소한 하나 이상의 outgoing 인터페이스를 지원하는 서버 객체를 연결가능 객체(connectable object), 또는 소스(source)라고 하며, 콜백 인터페이스를 구현하는 클라이언트 객체를 싱크(sink)라고 한다. 클라이언트가 콜백 인터페이스를 이용해서 연결가능 객체에 연결하기 위해서, 연결가능 객체는 반드시 특정 인터페이스에 대한 연결점(connection point)을 구현해야 한다. 서버 객체가 둘 이상의 outgoing 인터페이스를 지원하는 경우도 많은데, 이런 경우에는 여러 종류의 클라이언트에 대한 여러 개의 연결점을 구현해야 한다.
COM 객체는 연결점을 구현하기 위해 IConnectionPointContainer 와 IConnectionPoint 인터페이스를 사용한다. 서버 객체에 연결하고자 하는 클라이언트 객체는 일단 서버 객체에게 IConnectionPointContainer 인터페이스를 질의한다. 서버가 유효한 인터페이스를 전달하면, 클라이언트는 이 인터페이스의 FindConnectionPoint 메소드를 호출한다. 이때 파라미터로 클라이언트가 구현하고 있는 outgoing 인터페이스의 인터페이스 ID(IID)를 넘기게 된다. 서버가 넘어온 인터페이스를 지원한다면 IConnectionPoint 인터페이스의 포인터를 cp 파라미터에 담아서 돌려주게 되며, 이 파라미터가 클라이언트가 사용하는 연결점이 된다. 마지막으로 클라이언트가 실제 인터페이스를 구현하는 부분의 포인터를 서버에 넘겨주게 되면 연결이 확립된다. 이를 위해, cp 파라미터를 이용해서 클라이언트는 IConnectionPoint 인터페이스의 Advise 메소드를 호출할 수 있는데, 여기에 지원하는 싱크 인터페이스의 포인터를 unkSink 파라미터에 담아서 보내게 된다. Advise 메소드의 dwCookie 파라미터는 서버가 클라이언트에게 반환하는 것으로, 일종의 ID 와 같은 것이다. 이를 이용해서, 클라

이언트가 연결점과의 연결을 해제할 수 있다. 실제로 연결을 해제할 때에는 IConnectionPoint 인터페이스의 UnAdvise 메소드를 사용한다.

쉽게 말해서, 연결점 컨테이너(IConnectionPointContainer)는 서버 객체가 지원하는 모든 연결점들에 대한 목록이다. 이는 서버 객체가 거의 무한대의 outgoing 인터페이스를 지원할 수 있다는 것으로, 이들 인터페이스는 각각 특정 연결점으로 정의된다. 또한, 하나의 연결점은 무한대의 클라이언트를 지원할 수 있다. 즉, 클라이언트가 IConnectionPoint 인터페이스의 Advise 메소드를 호출할 때, dwCookie 파라미터에 각 클라이언트에 대해서 유일한 ID 를 제공하기 때문에, 하나의 연결점에 연결되는 클라이언트 들이라도 서버는 각각을 dwCookie 를 이용해서 구별할 수 있게 된다.

이들 인터페이스는 다음과 같이 선언되어 있다.

```
IConnectionPointContainer = interface
  function EnumConnectionPoints(out enum: IEnumConnectionPoints): HResult;
  function FindConnectionPoint(const iid: TIID; out cp: IConnectionPoint): HResult;
end;

IConnectionPoint = interface
  function GetConnectionInterface(out iid: TIID): HResult;
  function GetConnectionPointContainer(out cpc: IConnectionPointContainer): HResult;
  function Advise(const unkSink: IUnknown; out dwCookie: Longint): HResult;
  function Unadvise(dwCookie: Longint): HResult;
  function EnumConnections(out enum: IEnumConnections): HResult;
end;
```

그러면, 실제로 연결점(ConnectionPoint)을 이용하여 이벤트를 지원하는 COM 객체를 생성하고, 이를 활용하는 클라이언트를 간단하게 작성하도록 하자.

델파이 4 에서는 이벤트를 지원하는 COM 객체를 쉽게 만들 수 있도록 위저드의 기능이 확장 되었다. 그러므로, COM 객체에 이벤트를 지원하게 확장하는 것은 그리 어렵지 않게 구현할 수 있다.

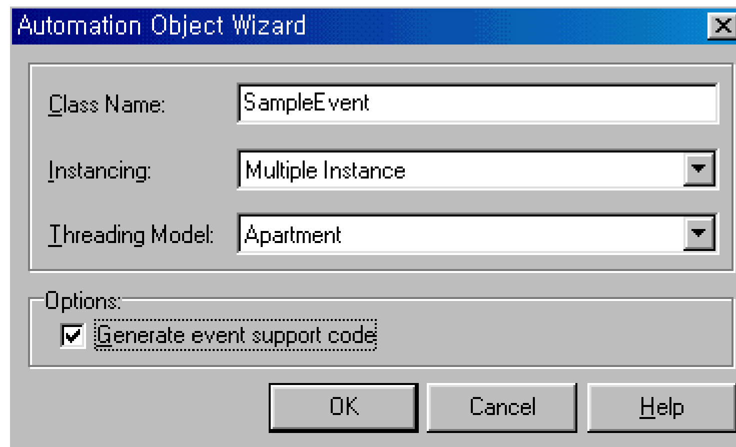
그런데, 이렇게 위저드의 형태로 확장한 이벤트 지원이 반쪽 밖에 없어서 이벤트를 지원하는 COM 객체는 쉽게 만들 수 있으나, 이를 이용하여 실제 이벤트를 지원하는 클라이언트를 제작하는 것은 꽤 어렵다. 보통은 해당 COM 객체에 대한 이벤트 wrapper 클래스를 오브젝트 파스칼에 맞도록 작성하여, 이를 이용하게 된다.

다행히, 이 문제를 쉽게 해결하기 위해 Binh Ly(bly@castle.net)가 이벤트 싱크를 쉽게 처리할 수 있는 유틸리티 유닛과 컴포넌트를 개발하여 프리웨어로 배포하고 있어서 비교적 쉽게 이벤트를 처리할 수 있게 되었다. 그러나, 이런 컴포넌트와 클래스의 이용 방법을 소개

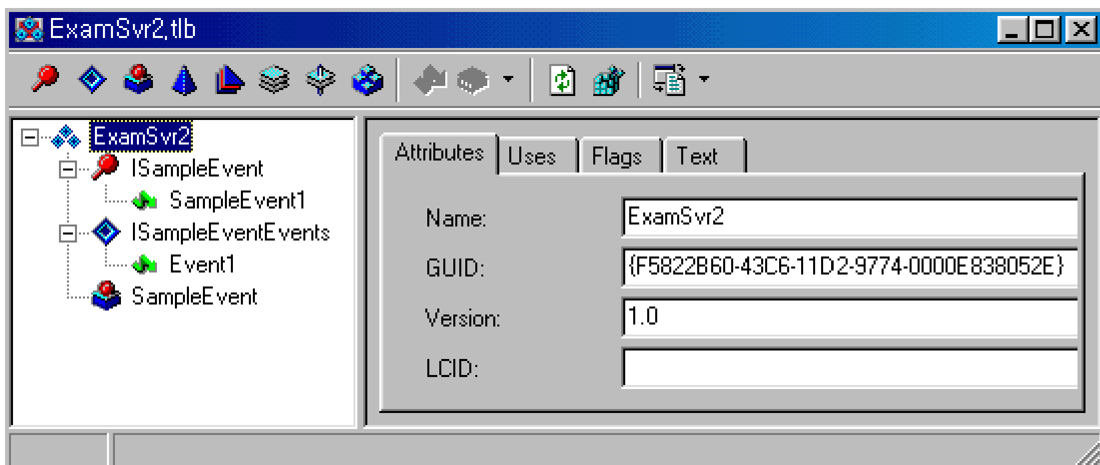
하는 것으로는 이벤트에 대한 명확한 이해가 어렵기 때문에, 다소 복잡하더라도 먼저 간단한 이벤트를 지원하는 COM 객체와 클라이언트를 직접 작성한 뒤에 이들에 대해 따로 설명하도록 하겠다.

이벤트를 지원하는 COM 객체

COM 객체에 이벤트를 지원하게 하는 것은 그리 어렵지 않게 구현할 수 있다.
먼저 File|New 메뉴를 선택한 뒤 ActiveX 탭에서 ActiveX Library 아이콘을 더블 클릭하여 프로젝트 파일을 생성한다. 그리고, File|New 메뉴의 ActiveX 탭에서 Automation Object 아이콘을 더블 클릭하여 자동화 객체를 생성할 대화 상자를 띄우도록 한다. 이 대화 상자에 생성할 클래스 이름을 지정하고, 이벤트를 지원하기 위해서는 다음과 같이 Generate event support code 체크 박스를 선택한다.



OK 버튼을 클릭하면, 타입 라이브러리 에디터가 실행되는데 여기에서 인터페이스의 메소드들을 다음과 같이 설정하도록 한다.



즉, 사용할 메소드를 ISampleEvent 메소드에 추가하고 이벤트는 DispInterface 인 ISampleEventEvents 인터페이스에 Event1 메소드를 추가한다. 이렇게 하고, OK 버튼을 클릭하면 다음과 같은 코드가 자동으로 생성될 것이다.

```
unit U_ExamSvr2;

interface

uses
  ComObj, ActiveX, AxCtrls, ExamSvr2_TLB;

type
  TSampleEvent = class(TAutoObject, IConnectionPointContainer, ISampleEvent)
  private
    { Private declarations }
    FConnectionPoints: TConnectionPoints;
    FEvents: ISampleEventEvents;
  public
    procedure Initialize; override;
  protected
    { Protected declarations }
    property ConnectionPoints: TConnectionPoints read FConnectionPoints
      implements IConnectionPointContainer;
    procedure EventSinkChanged(const EventSink: IUnknown); override;
    procedure SampleEvent1; safecall;
  end;

implementation

uses ComServ;

procedure TSampleEvent.EventSinkChanged(const EventSink: IUnknown);
begin
  FEvents := EventSink as ISampleEventEvents;
end;
```

```
procedure TSampleEvent.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoints.CreateConnectionPoint(AutoFactory.EventIID,
      ckSingle, EventConnect);
end;

procedure TSampleEvent.SampleEvent1;
begin

end;

initialization
  TAutoObjectFactory.Create(ComServer, TSampleEvent, Class_SampleEvent,
    ciMultiInstance, tmApartment);
end.
```

이 코드를 설명하면, 앞에서도 설명한대로 연결점 컨테이너 인터페이스를 이용하여 이벤트를 구현하게 된다. 클라이언트가 IConnectionPoint 인터페이스의 Advise 메소드를 호출할 때, dwCookie 파라미터에 각 클라이언트에 대해서 유일한 ID 를 제공하기 때문에, 하나의 연결점에 연결되는 클라이언트 들이라도 서버는 각각을 dwCookie 를 이용해서 구별할 수 있게 된다. 그런데, 델파이에서는 TConnectionPoints 클래스에서 이 인터페이스를 구현하기 때문에 이와 같이 간단히 선언하는 것으로 충분하다.

TAutoObject 클래스의 EventSinkChanged 메소드는 다른 인터페이스를 이벤트로 처리할 수 있도록 제공되는 가상 메소드로, 이를 오버라이드하여 파라미터인 EventSink 를 이벤트를 지원하는 Dispinterface 로 타입 캐스팅하여 이를 이용하여 이벤트의 지원이 가능하다.

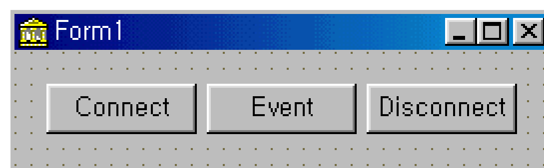
TAutoObject 클래스의 Initialize 메소드는 객체의 초기화를 할 수 있는 가상 메소드로, 여기에서 TConnectionPoints 클래스의 객체를 생성하고, 이 객체의 CreateConnectionPoint 메소드를 호출하여 파라미터로 지정된 연결점 객체를 생성하여 컨테이너에 포함한다.

개발자가 할 일은 어떤 메소드에서 이벤트를 발생시킬 것인지를 결정하면 된다. 앞에서 델파이가 자동으로 생성한 코드에 의해 FEvents 필드에 ISampleEventEvents 인터페이스에서 정의한 메소드들(이번 예제의 경우 Event)이 포함되므로, 특정 메소드에서 이들 FEvents 의 메소드를 호출하면 특정 메소드를 호출할 때 이벤트가 발생한다.

이번 예제에서는 ISampleEvent 인터페이스의 SampleMethod1 메소드를 호출하면 다른 작업은 하지 않고 Event1 을 호출하여 이벤트만 발생시키도록 다음과 같이 입력한다.

```
procedure TSampleEvent.SampleEvent1;
begin
  FEvents.Event1;
end;
```

이것으로 이벤트를 지원하는 COM 객체 서버의 제작되었다. 실제로 개발자가 한 일은 이벤트 코드를 생성하도록 하는 체크 박스를 선택한 것과 타입 라이브러리에서 인터페이스의 이름 뒤에 Events 가 붙은 Dispinterface 에 이벤트로 사용할 메소드를 추가하는 것, 그리고 인터페이스의 메소드 중에서 이벤트를 발생시킬 곳에서 FEvents 필드에 저장된 이벤트 메소드를 호출한 것 밖에 없다. 나머지는 모두 델파이가 알아서 한다.
이제 이를 컴파일하고, Run|Register ActiveX Server 명령을 선택하여 액티브 X 서버를 등록하도록 한다.
실제로 이렇게 이벤트를 지원하는 COM 객체 서버를 작성하는 것보다, 이벤트를 지원하는 서버를 사용하는 것이 훨씬 더 어렵다. 그러면, 앞에서 작성한 COM 객체 서버를 이용하는 클라이언트 어플리케이션을 만들어 보자.
먼저 폼을 버튼 3 개를 올려 놓고 다음과 같이 디자인한다.



이벤트를 구현한 서버를 이용하기 위해서는 먼저 interface 섹션의 uses 절에 ActiveX.pas 유닛과 ExamSvr2_TLB.pas 유닛을 추가한다. 또한 implementation 섹션의 uses 절에는 ComObj.pas 유닛을 추가하여 여러 유틸리티 함수를 이용할 수 있도록 한다.
그리고, 이벤트 싱크를 지원하는 클래스와 이를 객체로 사용하여 실제 사용할 수 있도록 하는 클래스를 선언하고 이를 구현해야 한다. 먼저 이벤트 싱크를 지원하는 클래스를 다음과 같이 선언한다.

```
  TSampleEventSink = class(TInterfacedObject, IUnknown, IDispatch)
  private
    FOwner : TObject;
    FDispatch: IDispatch;
    FDispIntfIID: TGUID;
```

```
    FConnection: Integer;
    FOnEvent: TNotifyEvent;
  protected
    //IUnknown
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    //IDispatch
    function GetIDsOfNames(const IID: TGUID; Names: Pointer;
      NameCount, LocaleID: Integer; DispIDs: Pointer): HResult; stdcall;
    function GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HResult; stdcall;
    function GetTypeInfoCount(out Count: Integer): HResult; virtual; stdcall;
    function Invoke(dispID: Integer; const IID: TGUID; LocaleID: Integer;
      Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HResult; stdcall;
  public
    constructor Create(AOwner: TObject; ADispatch: IDispatch; const DispIntfIID: TGUID);
    destructor Destroy; override;
    property OnEvent: TNotifyEvent read FOnEvent write FOnEvent;
  end;
```

여기서 TSampleEventSink 클래스는 TInterfacedObject 를 상속하되, 이벤트를 지원하기 위해서 IUnknown 과 IDispatch 인터페이스를 다시 구현하는 클래스이다. private 섹션에 선언된 FOwner 필드는 이벤트를 발생시킬 객체를 지정하게 되고, FDispatch 는 사용할 IDispatch 인터페이스를 그리고, FDispIntfIID 는 이벤트를 지원하는 Dispinterface 의 IID 를 저장한다. FConnection 필드는 이벤트와 연결을 하기 위해 호출하는 함수에서 필요로 하는 필드이다. 그리고, 실제 이벤트로 사용할 필드인 FOnEvent 를 선언하는데 이 필드는 TNotifyEvent 형으로 선언한다.
protected 섹션에는 IUnknown 과 IDispatch 인터페이스를 구현하기 위해 이들 인터페이스의 메소드 들을 선언한다. 그리고, public 섹션에 클래스의 constructor 인 Create 메소드를 새로 정의하고 Destroy 메소드는 오버라이드한다. 마지막으로 OnEvent 라는 이벤트를 프로퍼티로 정의한다.
그러면, 이 클래스를 구현해 보도록 하자. 먼저, constructor 와 destructor 를 다음과 같이 구현한다.

```
constructor TSampleEventSink.Create(AOwner: TObject; ADispatch: IDispatch;
  const DispIntfIID: TGUID);
```

```
begin
  inherited Create;
  FOwner := AOwner;
  FDispIntfIID := DispIntfIID;
  FDispatch := ADispatch;
  InterfaceConnect(FDispatch, FDispIntfIID, Self, FConnection);
end;

destructor TSampleEventSink.Destroy;
begin
  InterfaceDisconnect(FDispatch, FDispIntfIID, FConnection);
  inherited Destroy;
end;
```

즉, InterfaceConnect 와 InterfaceDisconnect 함수를 호출하기 위해서 필드에 constructor 에 넘어온 파라미터를 저장하고, 이를 이용하여 이벤트와 연결을 한다.
InterfaceConnect 프로시저는 IConnectionPoint 인터페이스를 이용하여 COM 서버에서 이벤트를 지원할 수 있도록 한다. 이 프로시저는 다음과 같이 선언되어 있다. 파라미터로 이벤트의 Source 로 사용될 인터페이스를 처음에, 이벤트 dispInterface 의 IID 를 두번째, 이벤트 Sink 로 사용할 인터페이스를 세번째 파라미터로 사용하며, 마지막에는 연결된 인터페이스의 핸들에 해당되는 값을 넘겨 받게 된다.

```
procedure InterfaceConnect(const Source: IUnknown; const IID: TIID;
  const Sink: IUnknown; var Connection: Longint);
```

마찬가지로, InterfaceDisconnect 프로시저는 지정된 인터페이스의 이벤트 연결을 해제하게 된다.
이렇게 이벤트를 지원하는 인터페이스가 있으면, IUnknown 인터페이스의 QueryInterface 메소드를 다시 구현해서 이벤트 인터페이스에 접근할 수 있도록 해야 한다. 그러므로, QueryInterface 를 다음과 같이 구현한다.

```
function TSampleEventSink.QueryInterface(const IID: TGUID; out Obj): HRESULT;
begin
  Result := E_NOINTERFACE;
  if GetInterface(IID,Obj) then
    Result := S_OK;
```

```
  if IsEqualGUID(IID,FDispIntfIID) and GetInterface(IDispatch, Obj) then
    Result := S_OK;
end;
```

즉, Obj 로 넘어온 파라미터만 검사하는 것이 아니라 FDIspIntfIID 필드에 저장된 이벤트 인터페이스의 IID 와도 동일한지 검사하여 이를 모두 허용하도록 하는 것이다.
IUnknown 인터페이스의 다른 메소드인 _AddRef, _Release 는 다음과 같이 구현한다. 이 내용은 인터페이스가 자동으로 해제되지 않도록 참조 계수를 지정하는 것이다.

```
function TSampleEventSink._AddRef: Integer;
begin
  Result := 2;
end;

function TSampleEventSink._Release: Integer;
begin
  Result := 1;
end;
```

이제는 IDispatch 인터페이스의 GetTypeInfo, GetTypeInfoCount, GetIDsOfNames, Invoke 메소드를 구현할 차례인데 이들 중 Invoke 를 제외하고는 그다지 중요하지 않으므로 다음과 같이 간단하게 구현한다.

```
function TSampleEventSink.GetTypeInfoCount(out Count: Integer): HRESULT;
begin
  Count  := 0;
  Result := S_OK;
end;

function TSampleEventSink.GetTypeInfo(Index, LocaleID: Integer; out TypeInfo): HRESULT;
begin
  Result := E_NOTIMPL;
end;

function TSampleEventSink.GetIDsOfNames(const IID: TGUID; Names: Pointer;
  NameCount, LocaleID: Integer; DispIDs: Pointer): HRESULT;
```

```
begin
  Result := E_NOTIMPL;
end;
```

Invoke 메소드에서는 이벤트를 지원하는 인터페이스에서 DispID 파라미터를 이용하여 여러 개의 이벤트 메소드를 매핑하는 역할을 하는데, 이 예제에서는 DispID 가 1 인 메소드 하나만 존재하므로 다음과 같이 간단하게 구현이 가능하다.

```
function TSampleEventSink.Invoke(DispID: Integer; const IID: TGUID; LocaleID: Integer;
  Flags: Word; var Params; VarResult, ExcepInfo, ArgErr: Pointer): HRESULT;
begin
  case DispID of
    1: if Assigned(FOnEvent) then FOnEvent(FOwner);
  end;
  Result := S_OK;
end;
```

이것으로 이벤트 싱크를 지원하는 클래스는 모두 구현하였다. 이번에는 우리가 작성한 자동화 객체 서버와 그 이벤트를 사용할 수 있도록 wrapping 한 클래스를 선언하고 구현할 차례이다. Wrapper 클래스를 다음과 같이 선언한다.

```
  TSampleEventObject = class
  private
    FSampleEvent: ISampleEvent;
    FEventSink : TSampleEventSink;
    function GetOnEvent: TNotifyEvent;
    procedure SetOnEvent(Value: TNotifyEvent);
  public
    constructor Create;
    destructor Destroy; override;
    property SampleEvent: ISampleEvent read FSampleEvent;
    property OnEvent: TNotifyEvent read GetOnEvent write SetOnEvent;
  end;
```

private 섹션에 사용할 인터페이스인 ISampleEvent 를 담을 필드 변수인 FSampleEvent 와 이벤트 싱크 객체를 담을 필드 변수인 FSampleEventSink, 그리고 이벤트를 실제로 구현할

접근 메소드(access method)인 GetOnEvent, SetOnEvent 메소드를 선언한다. 그리고, public 섹션에 constructor 와 destructor, 그리고 자동화 객체 서버의 인터페이스를 프로퍼티와 이벤트를 프로퍼티로 제공한다.
이 클래스의 구현은 간단하다. 먼저 constructor 를 다음과 같이 구현한다.

```
constructor TSampleEventObject.Create;
begin
  FSampleEvent := CoSampleEvent.Create;
  FEventSink := TSampleEventSink.Create(Self, FSampleEvent,
    DIID_ISampleEventEvents);
end;
```

즉, ISampleEvent 를 프로퍼티로 접근하여 사용할 수 있도록 필드 변수에 CoClass 를 생성하여 대입하고, 이벤트를 사용할 수 있도록 앞서 작성한 이벤트 싱크 클래스를 인스턴스화하면 된다. 이때 이벤트를 발생시키는 객체를 첫번째 파라미터로 사용하게 된다. 주의할 것은 세번째 파라미터인 DIID_ISampleEventEvents 인데 이 값을 이용하여 이벤트를 지원하는 인터페이스와 연결하게 된다. 이 값은 서버의 타입 라이브러리의 파스칼 버전인 ExamSvr2_TLB.pas 유닛에서 인터페이스의 이벤트를 지원하기 위해 선언된 Dispinterface 의 IID 상수를 사용해야 한다.
destructor 와 GetOnEvent, SetOnEvent 메소드는 다음과 같이 간단히 구현할 수 있다.

```
destructor TSampleEventObject.Destroy;
begin
  FEventSink := nil;
  inherited Destroy;
end;

function TSampleEventObject.GetOnEvent: TNotifyEvent;
begin
  Result := FEventSink.OnEvent;
end;

procedure TSampleEventObject.SetOnEvent(Value: TNotifyEvent);
begin
  FEventSink.OnEvent := Value;
end;
```

이것으로 wrapper 클래스의 구현이 끝났다. 이제 이를 이용하여 실제로 이벤트가 동작하는지 알아보도록 하자. 우리가 앞에서 작성한 자동화 서버는 서버의 인터페이스인 ISampleEvent 의 SampleMethod1 메소드를 호출하면 이벤트가 발생하도록 구현하였다. 이를 위해 이벤트인 이벤트 싱크 클래스를 구현하여 델파이의 이벤트 구조와 호환되도록 하였으므로 다음과 같이 TNotifyEvent 형의 메소드를 하나 구현하고, 이 값을 이벤트에 대입하여 이벤트가 발생시 이 메소드가 실행되도록 하면 된다.
먼저, wrapper 클래스를 담을 수 있는 전역 변수를 다음과 같이 선언한다.

```
var
  Form1: TForm1;
  SampleEventObject: TSampleEventObject;
```

그리고, 폼의 private 섹션에 OnEvent 이벤트 핸들러로 사용할 수 있는 메소드를 다음과 같이 추가하고 구현한다.

```
  private
    procedure Event(Sender: TObject);

... (중략)

procedure TForm1.Event(Sender: TObject);
begin
  ShowMessage('Event Fired !');
end;
```

Button1 을 클릭하면 이벤트 싱크 객체를 생성하고, 이벤트 핸들러로 앞서 선언하고 구현한 Event 메소드를 사용하도록 대입하도록 OnClick 이벤트 핸들러를 다음과 같이 작성한다.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not Assigned(SampleEventObject) then
  begin
    SampleEventObject := TSampleEventObject.Create;
    SampleEventObject.OnEvent := Event;
  end;
```

```
end;
```

그리고, Button3 를 클릭하면 생성된 wrapper 객체가 있으면 이를 해제하도록 다음과 같이 OnClick 이벤트 핸들러를 작성한다.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  if Assigned(SampleEventObject) then
  begin
    SampleEventObject.Free;
    SampleEventObject := nil;
  end;
end;
```

마지막으로 이벤트를 발생시키는 Button2 의 OnClick 이벤트 핸들러는 다음과 같이 작성한다. 단순히 ISampleEvent 인터페이스의 SampleEvent1 메소드를 호출하는 것으로 이벤트가 발생할 것이다.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if Assigned(SampleEventObject) then
  begin
    SampleEventObject.SampleEvent.SampleEvent1;
  end;
end;
```

프로젝트를 컴파일하고 실행한 뒤에, 'Connect' 버튼과 'Event' 버튼을 차례로 클릭하면 다음과 같이 이벤트가 발생하여 이벤트 핸들러가 실행되는 화면을 볼 수 있을 것이다.

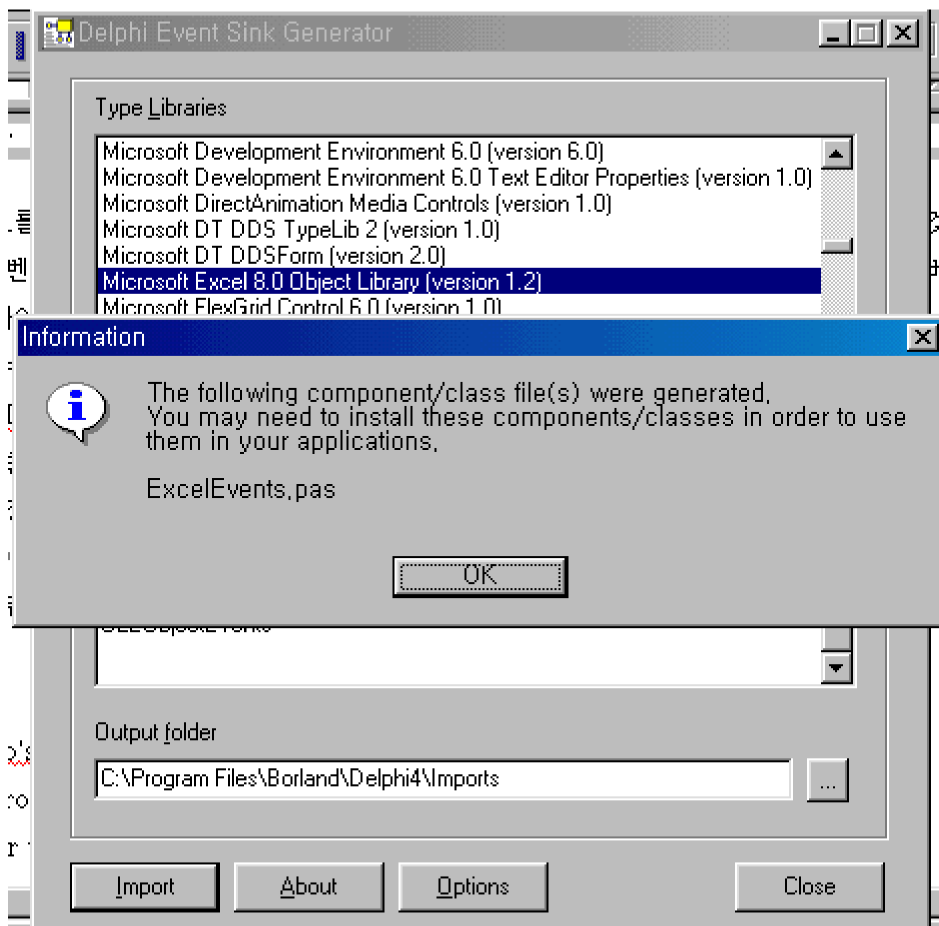
이벤트 지원 컴포넌트의 활용

간단한 이벤트를 지원하도록 했지만, 생각보다 작업이 만만치 않다는 것을 알 수 있을 것이다. 특히, 이벤트를 지원하도록 서버를 작성하는 것도 문제지만 이벤트를 지원하는 서버를 사용하는 클라이언트의 제작이 대단히 까다로운 것을 알 수 있다. 그렇다면, 이를 보다 편하게 해결할 수 있는 방법은 없을까 ?

다행히 Binh Ly(bly@castle.net)가 공개한 컴포넌트와 유틸리티를 이용하면 까다로운 자동화 객체의 이벤트를 쉽게 이용할 수 있다. 이 컴포넌트와 유틸리티의 소스와 데모 어플리케이션이 이 장에 해당되는 디렉토리의 EventSink 서브 디렉토리에 제공되므로 이를 참고하기 바란다.

이해를 돕기 위해 사용 방법을 간단히 소개하면 다음과 같다.

먼저 EventSinkImp 유틸리티를 실행하면, 컴퓨터에 설치된 자동화 서버의 타입 라이브러리에 대한 정보가 나열될 것이다. 이 중에서 사용할 타입 라이브러리를 선택하고 Import 버튼을 클릭하거나 더블 클릭하면 이벤트를 쉽게 사용할 수 있는 파스칼 유닛 파일이 다음과 같이 자동으로 생성된다.



이렇게 생성된 소스 코드는 Output folder 로 지정된 디렉토리에 생성되는데, 이 소스 코드는 컴포넌트 소스 코드이므로 Component|Install Component 메뉴를 이용하여 설치가 가능하다.
컴포넌트를 설치한 뒤에는 이 컴포넌트를 폼에 올려 놓고, 필요한 싱크 메소드를 후킹하여 사용하면 된다.
이렇게 설치된 컴포넌트에는 공통적인 서버 객체에 이벤트 싱크를 연결하고 해제하는 Connect, Disconnect 메소드가 제공되는데 이들은 다음과 같이 early 바인딩과 late 바인딩을 모두 사용하여 이용할 수 있다.
Early 바인딩의 경우에는 다음과 같이 한다.

```
var
  pObject1: IObject1;
begin
  pObject1 := CoObject1.Create;
  SinkComponent.Connect (pObject1 as IUnknown);
end;
```

그리고, late 바인딩의 경우 사용하는 방법은 다음과 같다.

```
var
  vObject1 : OleVariant;
begin
  vObject1 := CreateOleObject ('Server.Object1');
  SinkComponent.Connect (IUnknown (vObject1));
end;
```

이렇게 연결한 서버 객체와의 연결을 해제하려면 Disconnect 메소드를 호출하면 되는데, 싱크 컴포넌트가 파괴되는 경우에는 자동으로 호출되므로 따로 호출할 필요는 없다.
디렉토리에 포함된 데모는 IE 4.0 의 이벤트를 활용하는 예제이다. 이를 분석하면 구체적인 사용방법을 익힐 수 있을 것이다.

정 리 (Summary)

이번 장에서는 컬렉션과 이벤트를 구현하는 고급스러운 COM 기술의 구현 방법에 대해서 알아보았다. 간단한 자동화 객체를 만들기 위해서는 이런 어려운 내용을 특별히 익힐 필요가 없겠지만, 실제로 쓸모가 있는 제대로된 객체를 만들어 사용하려면 컬렉션과 이벤트의

구현은 필수적이다.

아무쪼록, 이 장에서 설명한 내용을 바탕으로 국내에서도 훌륭한 자동화 서버 객체나 COM 객체를 작성하여 강력한 어플리케이션을 개발하고, 가능하면 많은 개발자 들이 공유할 수 있도록 범용성을 갖춘 멋진 객체를 개발하고, 이를 공개하여 여러 개발자들에게 도움을 줄 수 있는 개발자들이 많았으면 하는 바람이다.