1 你好，ATL

欢迎来到活动模板库（Active Template Library，以后简称为 ATL）。在本章中，我将展示一些任务，希望您用 ATL 和集成的向导(integrated wizards)完成这些任务。这并不表示 ATL 只能完成这些工作，也不意味着这些工作完全覆盖了向导或者向导的输出。事实上本书其余部分的重点在于 ATL 如何实现 COM（Component Object Model，组件对象模型），以及通过 COM 粘合功能把这个例子粘在一起（以及其他一些例子）。本章实际上仅仅是我们熟悉“ Visual Studio 环境为 ATL 程序员提供的各种支持 ”的一次热身而已。

1.1 什么是 ATL？

ATL 的全称并没有很好地描述出什么是 ATL 或者为什么我们要使用它。事实上，其中的 *Active* 是 Microsoft 一切以市场为导向的产物，当时 ActiveX[1]就意味着 COM。当本书正在写作时，它的内涵又发生了变化，ActiveX 意味着控制（control，在其他一些中文书籍中，control 被译为控件——译者注)。ATL 确实为建立控制提供了大量的支持，但是它提供的内容远不止这些。

1.1.1 ATL 提供了什么

1．为维护代价很高的数据类型(例如接口指针、VARIANT、BSTR 和 HWND)提供包装类。

[1] 最初 ATL 的全称是 ActiveX Template Library。

2．提供一些类，它们实现了诸如 IUnknown、IClassFactory、IDispatch、IPersistXxx、IConnectionPointContainer 和 IEnumXxx 这些基本的 COM 接口。

3．管理 COM 服务器的类，它们用于暴露类对象、自注册和服务器生命周期管理。

4．节省手工录入的向导(Wizards)。

1.1.2 对读者的要求

ATL 从当今 C++类库领域中的模范成员 —— 标准模板库（STL，Standard Template Library）中得到启发。ATL 和 STL 一样，是由一些小巧、高效和灵活的类组成的集合。但是，伴随着权利而来的是职责。和 STL 一样，只有有经验的 C++程序员（最好有 STL 经验）才能有效地使用它。

当然，由于我们要进行 COM 编程，所以使用和实现 COM 对象和服务器的经验是绝对必需的。那些希望在对 COM 一无所知的情况下建立 COM 对象的人，ATL 不适合他们（在这种情况下 Visual Basic、Visual J++或者其他工具也不适合他们）。事实上，使用 ATL 意味着要非常熟悉 C++中的 COM 以及 ATL 本身的实现细节。

还有，ATL 集成了几种有助于生成初始代码的向导。在本章的剩余部分，我将展现 Visual C++ 6.0 中用于 ATL 编程的各种向导。让我们以轻松的心情继续吧！

1.2 创建 COM 服务器

1.2.1 运行 ATL COM AppWizard

使用 Visual C++进行开发的第一步总是建立工作区(workspace)和初始工程(project)。对于 ATL 程序员，这意味着从 New 对话框中的 Project 选项卡上选择 ATL COM AppWizard，如图 1.1 所示。注意 ATL COM AppWizard 是缺省的选择。这是因为它是使用 Visual Studio 创建的最重要的工程类型（按照字母顺序排在第一个不过是个巧合）。工程的名字（图中为 PiSvr）将成为最终生成的 DLL 或 EXE 服务器的名字。

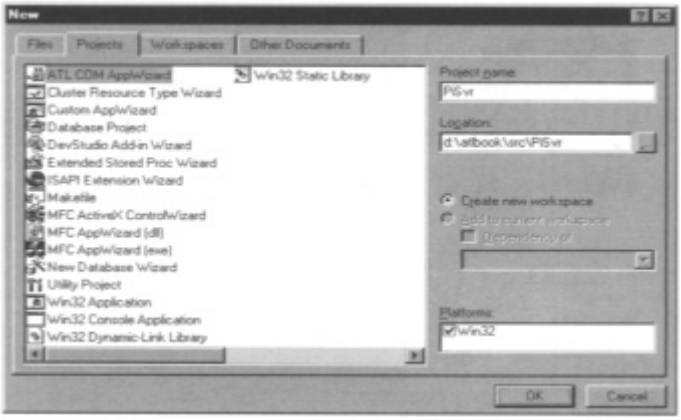

图 1.1　创建一个新的 Visual Studio 工程

ATL COM AppWizard **的工作是为我们的** COM **服务器建立一个工程。**COM **服务器可以是动态链接库（**DLL，dynamic link library**），也可以是可执行程序（**EXE，executable**）。进一步地，**EXE **既可以是独立的应用程序也可以是** NT **的系统服务。**ATL COM AppWizard **支持所有这三种服务器类型，如图** 1.2 **所示。**

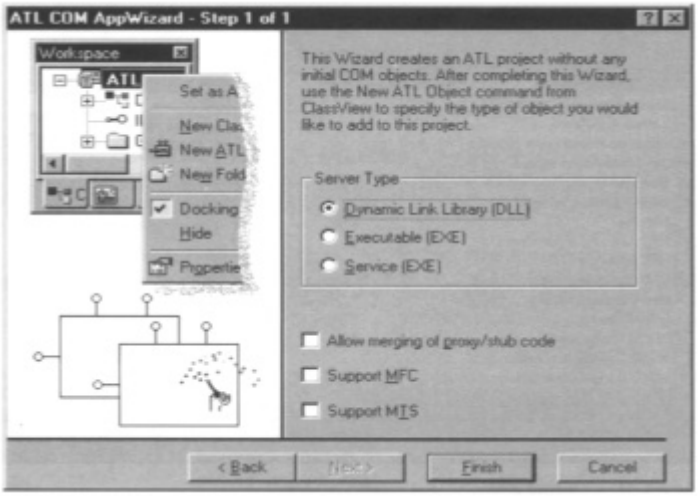

图 1.2　创建一个新的 ATL COM 服务器

ATL COM AppWizard **还提供了另外三个选项。第一个选项允许在** DLL **服务器中捆绑自定义的代理/存根（**proxy/stub**）代码。不管选择哪种服务器类型，当建立自定义的代理/存根时，**ATL **将为我们提供一个**<projectname>ps.mk **的编译文件。这个文件用于建立一个独立的代理/存根** DLL**，该** DLL **将被分发到所有需要列集（**marshal**）和散集（**unmarshal**）自定义接口的** COM **客户和服务器所在的机器上。如果我们希望把代理/存根** DLL **捆绑到** DLL **服务器中（因而在服务器所在的主机上可以节省一个独立的代理/存根** DLL**），那么请选择“**Allow merging of proxy/stub code**”选项。这样做之后，我们的服务器代码中将被放入一组条件编译声明，以便*允许*合并代理/存根代码。为了真正*合并*代理/存根代码，我们**

还必须完成以下操作步骤（在向导生成工程以后）。

1．在 Project Setting 对话框中为所有配置(all configurations)添加预处理器定义 _MERGE_PROXYSTUB。

2．在 dlldatax.c 文件的设置中不要选中“ Exclude file from build ”。dlldatax.c 实际上仅仅是把 dlldata.c 和<projectname>_Pi.c 带入工程的一个包装。

3．把 dlldatax.c 的预编译头文件的设置改为“ Not using precompiled headers ”。

第二个 ATL COM AppWizard 选项允许我们使用 MFC（Microsoft Foundation Classes，Microsoft 基础类库）。坦白地说，我们最好放弃这个选项。下面是我听到的关于“ 为什么开发人员认为他们应该在 ATL 服务器中使用 MFC ”的一些常见的意见。

1．“ 离开了 CString（或者 CMap、CList 等），我就无法工作。”在 C++标准委员会定义标准库之前，建立 MFC 实用类库只是权宜之计。既然标准库已经被建立起来了，那么我们就可以停止使用 MFC 了。STL 中提供的类（string、map、list 等等）比 MFC 中等价的类更为灵活和健壮。

2．“ 我不能没有向导。”本章全部是关于 Visual Studio 为 ATL 编程所提供向导的内容。到了 Visual C++ 6.0，ATL 向导几乎与生成 MFC 代码的向导一样强大。

3．“ 我已经了解了 MFC，我学不进任何新东西。”幸运的是，这些人不会读这本书，而且他们或者完全陶醉于正在使用的 973K 的 DLL，或者上帝保佑他们对此一无所知。[2]

第三个 ATL COM AppWizard 选项是“ 支持 MTS ”，它完成两件事情。第一，改变某些编译设置和链接设置使得 Microsoft 事务服务器（MTS ，Microsoft Transaction Server）能够获得它在代理/存根 DLL 中所需要的修改信息。这样可以避免一些单调乏味的手工操作。第二，向导加入了一个自定义的联编（build）选项，使得当建立服务器时 mtsrereg.exe 被自动执行。这使得 MTS 能够在每次联编之后重新获得 COM 服务器，因而我们无须手工更新 MTS 包。有些人在重编 MTS 服务器之后，突然发现它在 MTS 中不再运行了，这些人会很欣赏这个功能。

1.2.2 使用 ATL COM AppWizard 的结果

无论是否选中这三个选项，每个由 ATL COM AppWizard 生成的 COM 服务器都将具备普通 COM 服务器的三种功能，也就是自注册、服务器生命周期控制和暴露类对象的功能。作为一个附加的便利，AppWizard 添加了一个自定义的联编(build)步骤，用来在每次成功联编之后注册 COM 服务器。这个步骤根据服务器是 DLL 还是 EXE，分别运行 regsvr32.exe <project>.dll 或者运行<project>.exe /regserver。

关于每个 ATL COM 服务器支持的三种功能的更多信息，以及为了满足更高程度的并发性与生命周期的需要如何进行扩展，请参阅第 4 章。

[2] 公平地说，也可以把 MFC 静态链接，但是静态链接的最小开销仍然高达 250K 左右。

1.3　插入一个 COM 类

1.3.1　运行 ATL Object Wizard

一旦有了 ATL COM 服务器，我们可能想插入一个新的 COM 类。这可以通过 Insert 菜单中的 New ATL Object 菜单项来完成。必须指出的是，这里 ATL 开发组使用了错误的术语。实际上这里并不是要插入一个对象（即类型的实例），而是插入一个类（即类型的定义）。虽然术语是错误的，但是至少它们是一致的。在以后的章节中我们将要讨论的对象映射表，事实上全部是关于类的。[3]

当加入一个 ATL 类的时候，首先我们必须要选择 ATL 类的类型，如图 1.3 所示。如果我们正在学习和摸索，那么我们可能要花一些时间探究 ATLObject Wizard 所提供的各种类型的类。选择每一个类都会导致生成一组特定的代码，ATL Object Wizard 利用 ATL 基类提供大部分功能，然后为我们自定义的功能生成框架。ATL Object Wizard 实际上是我们决定 COM 类实现哪些接口的机会。当然，通过它我们并不能获得所有的 ATL 功能（甚至也不是绝大部分），但是当向导使我们完成初始工作之后，它所生成的代码被设计成“能够很容易地增加和减少功能”。多做试验是熟悉向导生成的各种类的类型和相应选项的最好方法。

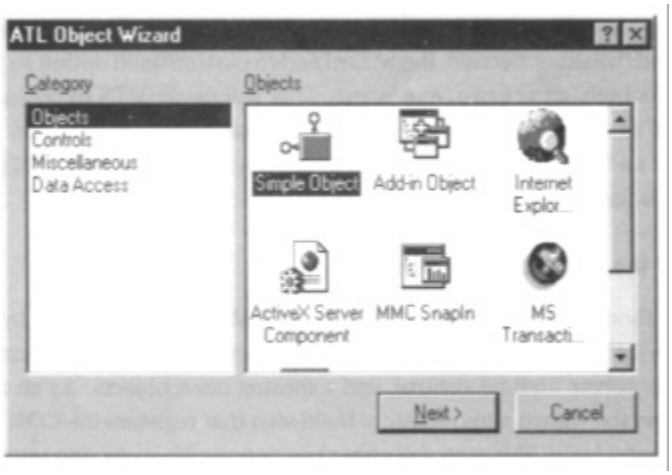

图 1.3　插入一个新的 ATL COM 类

[3] ATL 开发组的负责人 Christian Beaumont，向我保证他知道类和对象之间的区别，但是 Visual Studio 已经有了一个 ClassWizard（生成 MFC 代码），而且他不想把二者混淆。

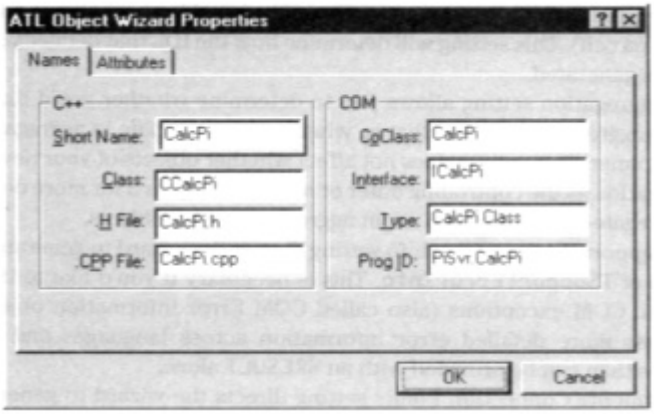

图 1.4　设置 COM 类的名字

一旦选定了某种类型（并且点击 OK**），向导一般会要求我们提供一些特定的信息。虽然有些类拥有比简单对象（**Simple Object**）更多的选项（如图** 1.3 **中的选择），但是绝大多数的** COM **类至少需要图** 1.4 **和图** 1.5 **所示的信息。**

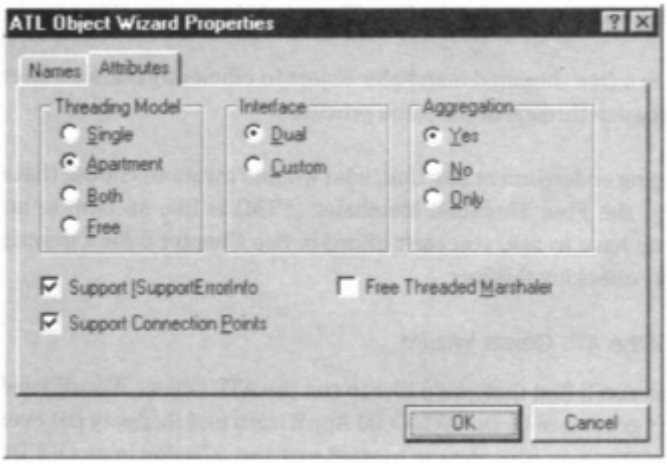

图 1.5　设置 COM 类的属性

ATL Object Wizard Properties **对话框中的** Name **选项卡实际上只需要输入短名字（**short name**），例如：**CalcPi**。这个短名字将用来生成对话框中的其余信息（如果愿意的话，也可以不用这些自动产生的信息）。这些信息被分为两类：必要的** C++**信息，包括** C++**类的名字、头文件和实现文件的名字；必要的** COM **信息，包括** CoClass **的名字（用于接口定义语言[**IDL**]）、缺省接口的名字（也用于** IDL**）、被称为** Type **的友好名字（用于** IDL **和注册设置），最后是版本无关的程序标识符（**programmatic identifier**，简称为** ProgID**，用于注册设置）。版本相关的** ProgID **由版本无关的** ProgID **加上后缀“** .1 **”组成。**

Attributes **选项卡使我们有机会对** COM **底层特征作出决定。线程模型设置（**Threading Model Setting**）描述了我们希望新类的实例将在何种套间中运行：单线程套间（**STA**，**

SingleThreaded Apartment）——也被称为套间线程模型（Apartment Model）或者多线程套间（MTA，MultiThreaded Apartment）——也被称为自由线程模型（Free-Threaded Model）。Single 模型用于少数“无论客户端的线程模型如何，都要求所有对象共享同一套间”的类。Both 模型用于那些和客户运行在同一套间中的对象，这样可以避免代理/存根对的开销。我们选择的线程模型将决定进程内服务器自注册设置中的 ThreadingModel 值，同时也决定了对象 AddRef 和 Release 方法的实现需要怎样的线程安全性。

接口设置（Interface Setting）允许我们为该类指定缺省接口的类型：自定义（需要自定义的代理/存根并且不从 IDispatch 派生）或者双接口（使用 typelib 列集器并且从 IDispatch 接口派生）。这个设置将决定如何生成定义缺省接口的 IDL。

聚合设置（Aggregation Setting）允许我们决定对象是否能够被聚合，也就是说是否能够作为受控的内部对象参与聚合。这个设置并不影响类的对象是否能够作为外部控制对象使用聚合。关于对象被聚合的更多信息请参阅第 3 章，关于聚合其他对象的更多信息请参阅第 5 章。

支持 ISupportErrorInfo 设置（Support ISupportErrorInfo）表明向导将生成 Isupport ErrorInfo 的一个实现。如果您希望在对象发生错误时抛出 COM 异常，那么这是必须的。COM 异常（也被称为 COM 错误信息对象）使得我们（跨过编程语言和套间的边界）传递的细节错误信息比单独一个 HRESULT 所能提供的信息更多。

支持连接点设置（Support Connection Point）将指示向导生成 IConnectionPoint 接口的一个实现，这个实现允许我们向脚本环境激发事件，例如 Internet Explorer 的脚本环境。连接点也被控制(control)用于向它的容器激发事件。

自由线程列集器（Free Threaded Marshaler）设置的帮助信息是这样描述的：

创建一个自由线程列集器对象，能够有效地在同一进程的不同线程之间列集接口指针。

带着如此冠冕堂皇的标签，谁会不愿意选择这个选项呢？不幸的是，自由线程列集器（Free Threaded Marshaler，FTM）就像高消费商店中的物品：无论是否需要，我们根本负担不起。在选择这个选项之前，请先参阅第 5 章关于 FTM 的描述。

1.3.2　骗出 ATL 对象向导

有时我们需要在一个不是由 ATL COM AppWizard 生成的工程中运行 ATL Object Wizard，甚至这个工程根本就不是一个 COM 服务器。例如，在一个独立的 Win32 应用中加入对话框或者 OLEDB Data Consumer 是件很普通的事。如果 Object Wizard 在工程中找不到生成代码的位置，它就会弹出一个如图 1.6 的对话框。如果您象我一样，把这条消息当做是对个人的一种挑战而不是真正的障碍，那么您会想知道怎么解决这个问题。最后，这里是 Visual C++ 6.0 中使用 ATL Object Wizard 的最低要求：[4]

[4] 感谢 ATL 邮件列表中的成员共同努力发现了这些要求，天晓得，这些内容竟然没有任何文档。

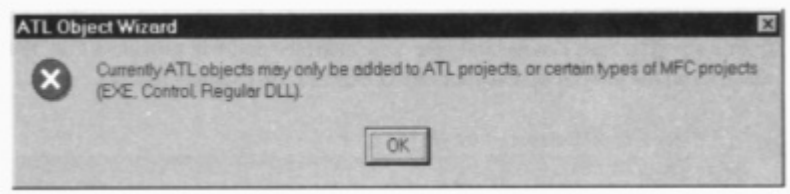

图 1.6　ATL Object Wizard 错误消息框

- **必须是一个** DLL **或者** Win32 **应用，不能是控制台应用程序。**
- **以下的代码必须出现在**<project>.cpp **文件中：**

```
BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

- **工程中必须有<project>.idl 文件（尽管在设置中它可以被标记为“Exclude file from build”）并且下面最小的 library 块必须出现（注释不是必须的）。**

```
// Library does not need a fuuid] attribute
library LIB // Library can be named anything, but it does need a name
{
}
```

- **如果集成开发环境（**intergrated development environment，IDE**）仍然给我们找麻烦，那么关闭工作区，删除**.ncb **文件，重新装载工作区，再试一次。在这样做没有任何效果的情况下，再试试**...**。**

警告：必须认识到即使骗出了 ATL Object Wizard**，我们仍将被那些本来** ATL Object Wizard **已经解决的问题所困扰，也就是说，我们的工程并没有完全正确的支持，以便能够真正编译和链接向导为我们生成的代码。所以如果想使用这项技术，我们最好能熟悉** ATL **源代码，以便手工把**.h **文件和**.cpp **文件添加到** stdafx.h **和** stdafx.cpp **中。附录** B **列出了** ATL **类所对应的头文件。**

1.3.3　ATL Object Wizard 的结果

完成必要的选项选择工作之后，ATL Object Wizard **将生成一个新的** C++**类，该类从适当的接口和** ATL **的接口实现派生，并且放到一个新的头文件中。同时也会生成一个** C++ **的实现文件供我们加入自己的代码。这个** C++**类从某些** ATL **基类和我们的新接口派生，并且提供一个** COM_MAP **列出对象暴露的所有接口。下面是一个例子。**

```
// Shaded code generated by ATL wizard
```

```
class ATL_NO_VTABLE CCalcPi :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CcomClClass<CCalcPi, &CLSID_CalcPi>,
  public IsupportErrorInfo,
  public IconnectionPointContainerImpl<CCalcPi>,
  public IDisatchImpl<ICalcPi, &IID_ICalcPi, &LIBID_PISVRL ib>
{
public:
  CCalcPi()
  {
  }

DECLARE_REGISTRY_RESOURCEID(IDR_CALCPI)

BEGIN_COM_MAP(CCalcPi)\
  COM_INTERFACE_ENTRY(ICalcPi)
  COM_INTERFACE_ENTRY(IDispatch)
  COM_INTERFACE_ENTRY(ISupportErrorInfo)
  COM_INTERFACE_ENTRY(IConectionPointContainer)
END_COM_MAP()

BEGIN_CONNECTION_POINT_MAP(CCalcPi)
END_CONNECTION_POINT_MAP()

// IsupportsErrorInfo
STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid);

// ICalcPi
public:
};
```

关于 ATL 用来实现基本 COM 功能的基础类的更多内容，以及怎样利用这些实现建立对象层和正确地同步多线程对象的内容，请参阅第 3 章。关于怎样充分利用 COM_MAP 的内容请看第 5 章。

1.4 加入属性和方法

类的声明（通常在.h 文件中）和类的定义（通常在.cpp 文件中）的分离是造成程序员的工作更加艰苦的因素之一。使之成为痛苦的原因是必需在两个文件之间进行维护工作。任何时候其中一个加入了一个成员函数，那么它必须被复制到另外一个文件中。如果用手工来完成，这是非常令人厌烦的工作，并且如果我们使用 C++进行 COM 编程，那么必须从.idl 开始工作，这更加令人厌烦。不要误解我。我很喜欢 IDL。一个独立于语言的接口和类的表示手段是我们继续前进的唯一途径。但是在向接口添加属性和方法时，我们希望 C++开发环境能够替我们把 IDL 翻译成 C++并且放到.h 和.cpp 文件中，为我们留下合适的空间来提供我们自己的实现。这正是 Visual C++提供的功能。

通过在 ClassView 中右键点击一个 COM 接口，我们可以选择添加一个新的属性或者方法。图 1.7 所示的对话框允许我们向 COM 接口添加一个属性。注意图 1.7 中的 Implementation 框。它是当前对话框中的选项设置以 IDL 形式表达的结果。当我们点击 OK 按钮之后，IDL 文件将被更新，并且将生成适当的 C++代码。下面带阴影的代码是向导生成的实现框架。我们仅须提供正确的行为（如没有阴影的代码所示）。

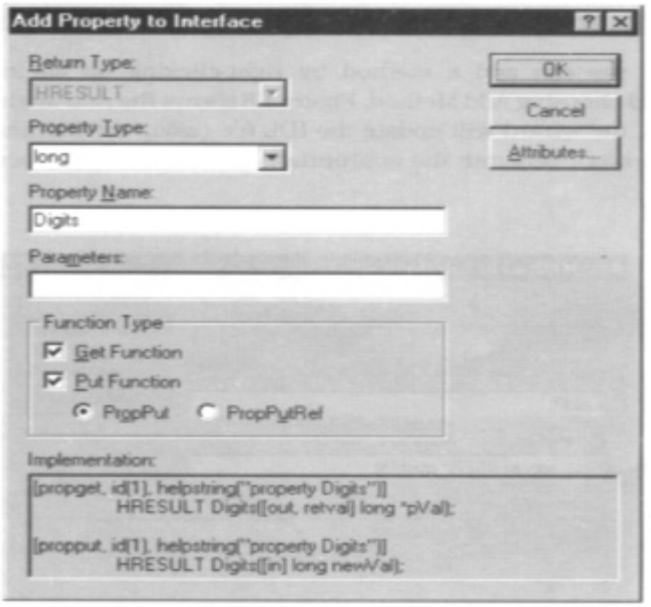

图 1.7 加入一个属性

```
STDMETHODIMP CCalcPi::get_Digits(long *pVal)
{
    *pVal = m_nDigits;
```

```
    return S_OK;
}

STDMETHODIMP CCalcPi::put_Digits(long newVal)
{
   if( newVal < 0)
    return Error(L"Can't calculate negative digits of PI");
    m_nDigits = newVal;

    return S_OK;
}
```

类似地，通过在 ClassView **中右键点击一个** COM **接口并且选择** Add Method**，我们可以增加一个方法。图** 1.8 **是向接口添加方法的对话框。向导将再次更新** IDL **文件（使用** Implementation **框中显示的** IDL**），生成适当的** C++**代码，并且把我们带到函数实现的框架部分以便于我们开始工作。下面带阴影的代码是向导生成的代码，其他则是我们为方法实现而加入的代码。**

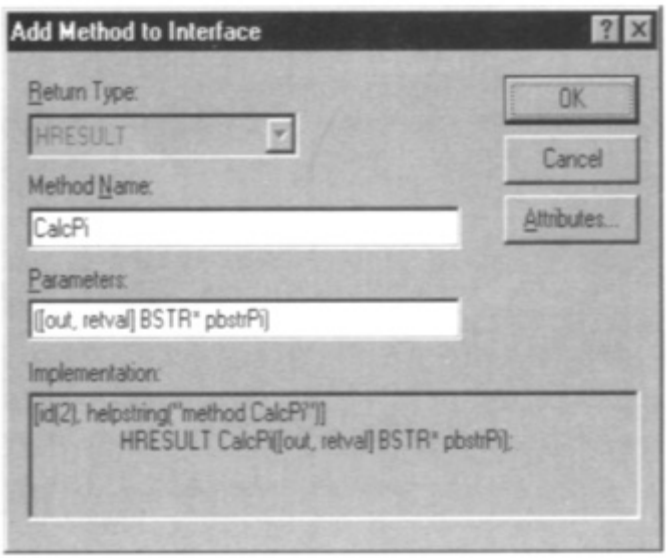

图 1.8　加入一个方法

```
STDMETHODIMP CCalcPi::CalcPi(BSTR *pbstrPi)
{
  _ASSERTE(m_nDigits >= 0);
```

```
if( m_nDigits )
{
   //create buffer for "3." and up to 8 more digits than asked for
   *pbstrPi = SysAllocStringLen  (OLESTR ("3."), m_nDigits + 10);
   if( *pbstrPi )
  {
      for( int i = O; i < m nDigits; i += 9 )
     {
         long nNineDigits = NineDigitsOfPiStartingAt(i+l);
         swprintf(*pbstrPi + i+2, L"%O9d", nNineDigits);
      // Truncate to number of digits
      (*pbstrPi) [m_nDigits+2] = O;
}
else
{
   *pbstrPi = SysAllocString(L"3");
return *pbstrPi ? S_OK : E_OUTOFMEMORY;
}
```

对 COM **异常和** ATL Error **函数的描述（成员函数** put_Digits **中用到的），请看第** 4 **章。**

1.5 实现其他接口

接口是 COM **的核心，大多数** COM **对象不只实现一个接口。甚至前面讲到的由向导生成的简单对象也实现了四个接口（一个自定义接口和三个标准接口）。如果希望基于** ATL **的** COM **类实现另一个接口，我们必须首先拥有它的定义。例如：**

```
interface IAdvertiseMyself : Iunknown
{
 [helpstring("method ShowAd")] HRESULT ShowAd(BSTR bstrClient);
};
```

为了实现这个接口，我们需要一个描述该接口的 C++**头文件，并且必须作少量的录入工作来为类添加接口和它的方法：**

```
class ATL_NO_VTABLE CCalcPi :
 publ i c CComObjectRootEx<CComSi ngl eThreadModel>,
 publ i c CComCoC1 ass<CCal cpi, &CLSID_Cal cpi >,
 ...
 publ ic IAdvertiseMysel f
{
...
BEGIN_COM_MAP (CCal c Pi )
    ...
   COM_INTERFACE_ENTRY(IAdverti seMysel f)
END_COM_MAP ()
...
// IAdvertiseMyself
      STDMETHOD(ShowAd) (BSTR bstrC1 i ent);
};
```

当然，添加了这几行代码后，我们还必须实现这些方法。

```
STDMETHODIMP CCal cpi:: ShowAd (BSTR bstrC1 ient)
{
 CComBSTR  bstrCaption = OLESTR("CalcPi hosted by ");
 bstrCaption += (bstrClient && *bstrClient ? bstrClient : OLESTR
     ("no one");
 CComBSTR  bstrText = OLESTR("These digits of pi brought to you by
    Cal cpi !");
USES_CONVERSION;
MessageBox(O, OLE2CT(bstrText), OLE2CT(bstrCaption),
   MB_SETFOREGROUND);
return S_OK;
}
```

如果我们想实现的接口碰巧在类型库（TypeLib**）中已经有定义，那么可以使用** Implement Interface wizard**（实现接口向导）为我们加入框架代码。通过在类上右键点击并选择“** Implement Interface **”进入实现接口向导。**Implement Interface **对话框（图** 1.9**）显示出当前选择的类型库中所有当前类尚未实现的自定义接口和双接口。缺省情况下，向导显**

示与当前工程关联的类型库，[5] 但是我们可以通过 Add TypeLib 按钮来选择任何类型库。不幸的是，截止到写作本书时，Implement Interface 对话框不支持类型库中不存在的接口，这漏掉了大部分标准的 COM 接口，例如：IPersist、IMarshal 和 IOleItemContainer。希望将来的版本能够弥补这个缺陷。

在我们选择实现一个接口之后，向导使当前类从该接口派生，并把接口加入到 COM_MAP 中，把方法原型加入类中（以及一些有用的框架代码）。

关于 ATL 允许我们的 COM 类实现接口的各种方法的更多内容请看第 5 章。关于 ShowAd 方法中用到的 CComBSTR 和字符串转换例程的内容请看第 2 章。

图 1.9　Implement Interface 对话框

1.6　支持脚本

当我们运行 ATL Object Wizard 并且选择接口类型为双接口时，ATL 会为从 IDispatch 派生的缺省接口生成 IDL，并且标记为 dual 属性，如下：

```
[
   object,
   uuid(DEC22F36-DD78-11D1-97FD-006008243C8C),
   dual,
   helpstring(IcalcPi Interface),
   pointer_default(unique)
```

[5] 如果我们想实现工程的 IDL 文件中新声明的接口，那么在使用 Implement Interface 向导之前必须要重建类型库。

```
]
interface IcalcPi : Idispatch
{
...
};
```

因为我们的双接口从 IDispatch 派生，所以它可以被脚本环境中的客户使用，例如：Active Server Pages（ASP）和 Internet Explorer（IE）。ATL 在 IDispathImpl 基类中提供了四个 IDispatch 方法的实现：

```
template <class T,
          const IID* piid,
          const GUID* plibid =&CcomModule::m_libid,
          WORD wMajor = 1,
          WORD wMinor = 0,
          Class tihclass = CcomTypeInfoHolder>
Class ATL_NO_VTABLE IDispatchImpl : public T
{...};
```

为了支持脚本环境，我们的 ATL COM 类从 IDispathImp1 派生并且把 IDispath 添加到 COM_MAP 中，如下所示：

```
class ATL_NO_VTABLE Ccalcpi : ...
   public IDispatchImpl<icALCPI, &IID_Icalcpi, &LIBID_PISVRL ib>
{
public:
BEGIN_COM_MAP(CcalcPi)
   COM_INTERFACE_ENTRY(IcalcPi
   COM_INTERFACE_ENTRY(IDispatch)
   COM_INTERFACE_ENTRY(ISupportErrorInfo)
   COM_INTERFACE_ENTRY(ICONNECTIONPointContainer
END_COM. MAPO
...
};
```

毫无疑问您已经注意到，通过在 ATL Object Wizard 中简单地选择双接口，向导就会生成前面展示的所有实现代码以及接口定义。一旦我们的 COM 类支持 IDispatch，我们就可

以在脚本环境中使用该类的对象。下面是一个使用 CalcPi 对象实例的 HTML 页面的例子：

```
<object c] assid="cl si d: DEC22F37-DD78-IIDI-97FD-O06008243CSC"
          i d=obj PiCa1 culator>
</object>
<script 1 anguage=vbscri pt>
  'Set the digits property
 objPiCalculator.digits = 5

  ' Calculate pi
 dim pi
 pi = objPiCalculator.CalcPi

  ' Tell the world!
 document.write "Pi to "& objPiCalculator.digits &" digits is "& pi
</script>
```

更多关于如何处理脚本相关的数据类型，也就是 BSTR 和 VARIANT 的内容请看第 2 章。

1.7 添加永久性

ATL 为那些希望能够被永久化的对象提供了一些基类，也就是说，这些对象可以被保存到某种永久介质上（如磁盘），并且以后可以恢复。COM 对象通过实现一个 COM 永久接口，例如：IPersistStreamInit、IPersistStorage、IPersistPropertyBag 来暴露永久性支持。ATL 提供了这三个永久接口的实现，也就是 IPersistStreamInitImpl、IPersistStorageImpl 和 IPersistPropertyBagImpl。只要我们的 COM 类从这些基类中的任何一个派生，并且增加一个名为 m_bRequiresSave(三个基类都需要)的数据成员，再把接口名字添加到 COM_MAP 中，那么我们的 COM 对象就支持永久性。

```
class ATL_NO_VTABLE CCalcPi :
 publ i c CComObj ectRootEx<CComSi ngl eTh readModel >,
 public CComCoClass<CCalcPi, &CLSID_CalcPi>,
 public IDispatchlmpl<ICalcPi, &IID_ICalcPi, &LIBID_PISVRLib>,
 publ ic IPersi stPropertyBagImpl <CCal cpi>
```

```
{
public:
...

BEGIN_COM_MAP (CCal c Pi )
   COM_INTERFACE_ENTRY (ICal cpi )
   COM_INTERFACE_ENTRY (IDi spatch)
   COM_INTERFACE_ENTRY (IPe fsi stPropertyBag)
END_COM_MAP O

 // ICal cpi
public:
 STDMETHOD(CalcPi) (/*[out, retval]*/ BSTR* pbstrPi);
 STDMETHOD(get_Digits) (/*[out, retval]*/ long *pVal);
 STDMETHOD(put_Digits) (/*[in]*/ long newVal);

publ i c:
 BOOL m_bRequiresSave; // Used by persistence base classes

private:
 long m_nDigits;
};
```

然而，所有这些工作并没有完全实现永久性。ATL **对永久性的实现需要知道对象的那些部分需要被保存和恢复。**ATL **对永久接口的实现依赖于一个对象属性表，其中包括我们希望在两次会话之间保持不变的对象属性。这个表被称作** PROP_MAP**，它包含了属性名字和分发标识符（**DISPID**，定义在** IDL **中）之间的映射。因此，假设下面的接口，**

```
interface ICalcPi:IDispatch
{
   propget, id (1)] HRESULT Digits([out, etval] long *pVal);
  [propput, id(l)] HRESULT Digits([in] long newVal);
  [id(2)] HRESULT CalcPi([out, retval] BSTR* pbstrPi);
};
```

PROP_MAP **必须像下面那样包含在** ICalcPi **实现的内部：**

```
class ATL_NO_VTABLE CCalcPi : ...
{
...
public:
BEGIN_PROP_MAP (CCal cpi)
 PROP_ENTRY("Digits", 1, CLSID_NULL)
END_PROP_MAP O
};
```

给出了 IPersistPropertyBag 的一个实现之后，我们的 IE 例子代码可以扩充为通过永久性、并且使用<param>标记来支持对象属性的初始化，如下：

```
<object cl assi d:"cl si d: DEC22F37-DD78-11D1-97FD-006008243C8C"
           i d=obj Pi Cal cul ator>
           <param name:digits value:5>
</object>
<script language=vbscript>
  ' Calculate pi
 dim pi
 pi = objPiCalculator.CalcPi

  ' Tell the world!
 document.write "Pi to "& objPiCalculator.digits &" digits is "& pi
</scri pt>
```

更多关于 ATL 永久性实现的内容请参见第 6 章。

1.8 添加和激发事件

当 COM 对象内部发生了有趣的事情，我们希望它能够自发地通知客户，而不用客户去轮询对象。通过连接点结构，COM 提供了向客户发送这些通知（通常称之为*激发一个事件*）的标准机制。

事实上连接点事件仅仅是一个接口的方法。为了支持最广泛的客户，事件接口常常被定义为 dispinterface。我们在 ATL Object Wizard 中选择支持连接点（Support Connection Points）之后，向导会生成一个事件接口，并且把它当作类中缺省的源接口（default source

interface）进行发布。下面的例子显示了向导生成的代码，以及后增加的一个事件方法（如粗体字所示）：

```
dispinterface _IcalcPiEvents
{
 properties:
 methods:
      [id(l)] void OnDigit([in] short nlndex, [in] short nDigit);
};

coclass CalcPi
{
      [default] interface ICalcPi;
      [default, source] dispinterface _ICalcPiEvents;
};
```

一旦定义了事件接口，并且服务器的<project>.tlb 已经被建立起来，那么右键点击 ClassView 中的类并且选择“ Implement Connection Pont ”将会弹出如图 1.10 所示的对话框。在对话框中选择一个事件接口（并按 OK 按钮），向导将生成一个包装类（被称为连接点代理）用于向感兴趣的客户激发方法：

```
template<class T>
class Cproxy_ICalcPiEvents:
 public IConnectionPointImpl<T,&DIID_ICalcPiEvents,
                          CComDynamicUnkArray>
{
//Warning this class may be recreated by the wizard.
public:
  VOID Fire_OnDinit(SHORT nTndex,SHORT nDigit);
};
```

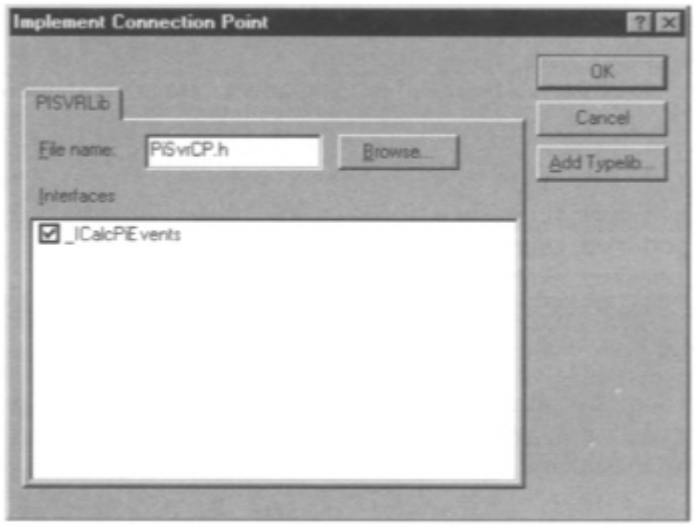

图 1.10　Implement Connection Point 对话框

由于向导同样会为激发这些事件的对象的类生成代码，使它们从代理类派生，所以当任何感兴趣的事件发生时，这个类可以自由地使用 Fire **方法。例如：**

```
STDMETHODIMP CCalcPi: :CalcPi (BSTR *pbstrPi) {
 // (code to calculate pi removed for clarity)
...

 // Fire each digit
 for( short j = O; j != m_nDigits; ++j ) {
    Fire_OnDigit(j, (*pbstrPi) [j+2]-L' 0');
 }


...
}
```

为了使脚本客户接收事件，对象必须支持 IProvideClassInfo2 **接口。这个接口允许客户查询对象的缺省接口标识符（**default interface identifier**），然后用它通过** IconnectPoint Container **接口建立联系。**ATL **提供了** IProvideClassInfo2 **接口的一个实现，如下：**

```
class ATL_NO_VTABLE CCalcPi :
  public CComObjectRootEx<CComSingleThreadModel>,
```

```
  public CcomCoClass<CCalcPi, &CLSID_CalcPi>,
  public IconnectionPointContainerImpl<CCalcPi>,
  public Cproxy_ICalcPiEvents<CCalcPi>,
  public IprovideClassInfo2Imp<&CLSID_CalcPi, &DIID_IcalcPiEvents>,
... {
public:
BEGIN_COM_MAP(CCalcPi)
   COM_INTERFACE_ENTRY(IconnectionPointContainer)
   COM_INTERFACE_ENTRY(IProvideClassInfo)
   COM_INTERFACE_ENTRY(IProvideClassInfo2)
   ...
END_COM_MAP()

BEGIN_CONNECTION_POINT_MAP(CCalcPi)
  CONNECTION_PDINT_ENTRY(DIID__IcalcPiEvents)
END_CONNECTION_POINT_MAP()
...
};
```

对象现在可以发送被 HTML 页面处理的事件，如下：

```
<object cl assi d="cl si d: DEC22F37-DD78-11D1-97FD-O06008243C8C"
           i d=obj Pi Cal cul ator>
           <param name=digits value=50>
</object>

<input type=button name=cmdCalcPi value="Pi to 50 Digits:">
<span id=spanPi>unknown</span>

<p>Distribution of first 50 digits in pi:
<table border cellpadding=4>
... <!- table code removed for clarity-->
</tabl e>

<script language=vbscript>
  ' Handle button click event
```

```
sub cmdCalcPi_onClick
   spanPi .innerText = objPiCalculator.CalcPi
end sub

 ' Handle calculator digit event
sub objPiCalculator_onDigit(index, digit)
   select case digit
   case 0: span0.innerText = span0.innerText + 1
   case 1: span1.innerText = span1.innerText + 1
    ... <!- etc -->
   end select
   spanTotal.innerText = spanTotal.innerText + 1
  end sub
</script>
```

HTML 例子页面处理这些事件以便提供 pi 的前 50 位和它们的分布，如图 1.11 所示。

图 1.11　50 位 pi 的 HTML 例子页面

关于 ATL 对连接点支持的更多内容请看第 8 章。

1.9　使用窗口

由于我们的开发平台是 Microsoft Windows，所以有时可以很容易地放置一个窗口或者对话框。例如我们前面对 MessageBox 的调用产生一个有点令人生厌的招牌，如图 1.12。一般来说，建立一个自定义的对话框是一种痛苦。对于普通的 C++程序员，要么陷入很多

我们不喜欢的过程代码，要么就是陷入建立那些“把 Windows 消息映射到成员函数”的传递代码（毕竟对话框是一个对象）。和 MFC 一样，ATL 有许多功能可用来建立窗口和对话框。为了添加新的对话框，ATL Object Wizard 在杂项类别（Miscellaneous category）中提供了 Dialog 对象，如图 1.13 所示。ATL Object Wizard 中有关对话框的内容（图 1.14）比其他部分要简单得多，仅允许我们输入 C++名字信息，因为对话框只是一个 Win32 对象，不是 COM 对象。

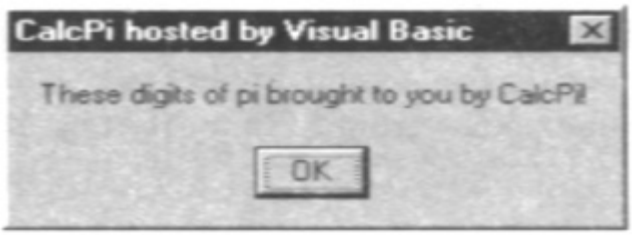

图 1.12　令人生厌的消息框

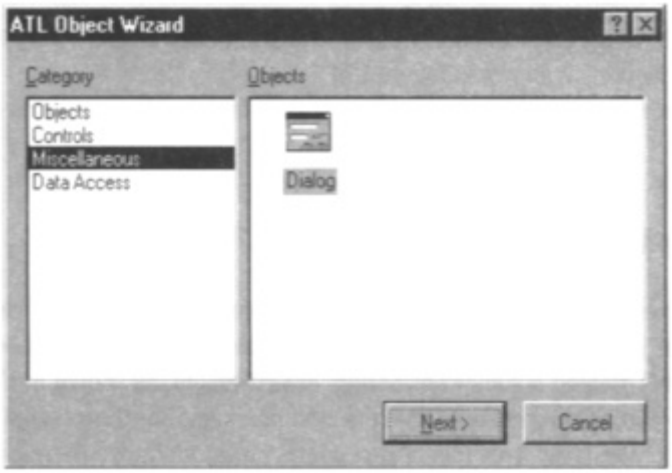

图 1.13　插入对话框类

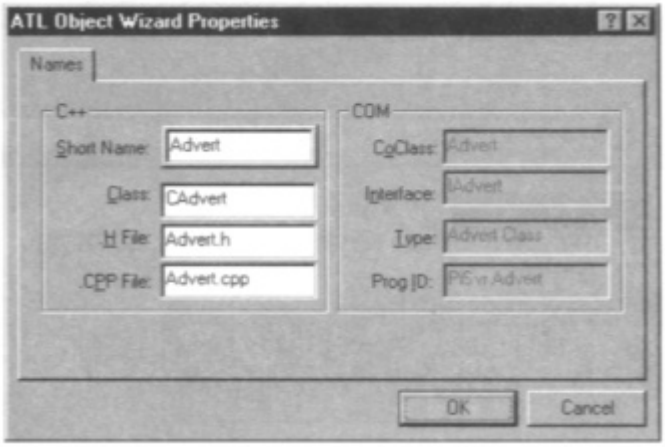

图 1.14　与对话框有关的 ATL Object Wizard 属性

产生的代码创建了一个从 CAxDialogImpl **派生的类，并且使用一个同样由向导提供的新的对话框模板。派生类使用** MSG_MAP **宏把消息传递给处理函数，如下所示：**

```
class Cadvert : public CaxDialogImp<CAdvert>
{
public:
 CAdvert() {}
 ~CAdvert() {}
 enum { IDD = IDD_ADVERT };

BEGIN_MSG_MAS(CAdvert)
 MESSAGE_HANDLER(WM_INITDIDIALOG, onInitDialog)
 COMMAND_ID_HANDLER(IDOK, OnOK)
 COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
END_MSG_MAP()

 LRESULT OnInitDialog(UINT uMsg, wAPAM wPARAM wParam, LPARAM 1Param,
                      BOOL& bHandled) {
    if( m_bstrClient. Length()) {
    CComBSTR  bstrCaption = OLESTR("CalcPi sponsored by ");
    bstrCaption += m_bstrClient;

    USES_CONVERSION;
    SetWindowText(OLE2CT(bstrCaption));
```

```
  }

  return 1; // Let the system set the focus
}

LRESULT OnOK(WORD wNotifyCode, WORD wID, HWND hWndCtl,
        BOOL& bHandled) {
   EndDialog(wID);
  Return 0;
}
  LRESULT OnCancel(WORD wNotifyCode, WORD wID, HWND hWndCtl,
       BOOL& bHandled) {
   EndDialog(wID);
   Return 0;
}

CComBSTR m_bstrClient;
};
```

如果想处理另外一个消息，我们可以在消息映射中加入适当的入口，并且手工添加消息处理成员函数。如果愿意，我们也可以运行 Message Handler **向导，做法是在** ClassView **中右键点击以** CWindowImpl **或者** CDialogImpl **为基类的类的名字，并且选择** Add Window Message Handler**。**Message Handler **对话框如图** 1.15 **所示。**

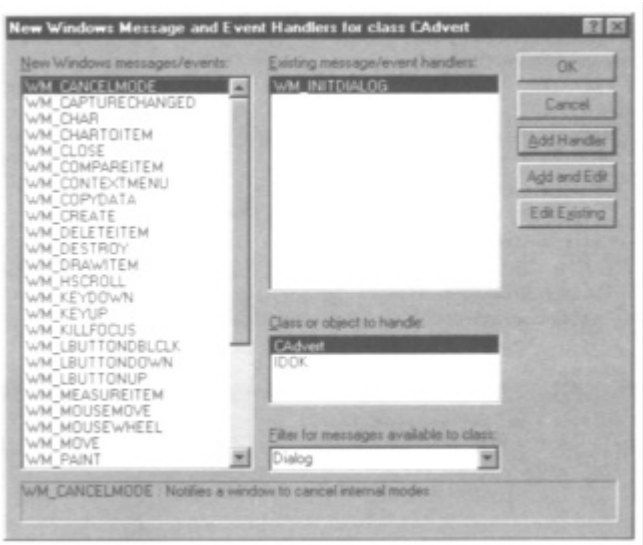

图 1.15　加入一个窗口消息处理函数

有关 ATL 对窗口的各种支持（包括创建独立的 Windows 应用）的更多内容请参阅第 9 章。

1.10　实现组件类别

如果您一直跟随本书学习，或者您以前曾经在 IE 中运行过 COM 对象，那么您可能已经注意到如图 1.16 所示的令人厌烦的对话框。这是 IE 用它的方式告诉我们，CalcPi 对象在 Internet 这个疯狂的世界中没有承诺它一定是安全的。因为实际上计算 pi 并不造成任何危害，所以我们希望 IE 停止使用这种不友好的对话框来骚扰我们的用户（或者至少是我们自己）。为了让 IE 知道我们承诺这种行为方式，我们必须在注册表中为我们的类列出它实现了哪些组件类别。组件类别是“指出一个类具有特定行为”的途径。例如，Embedded 组件类别意味着一个类的对象能够使用 OLE 协议被嵌入到其他程序中。

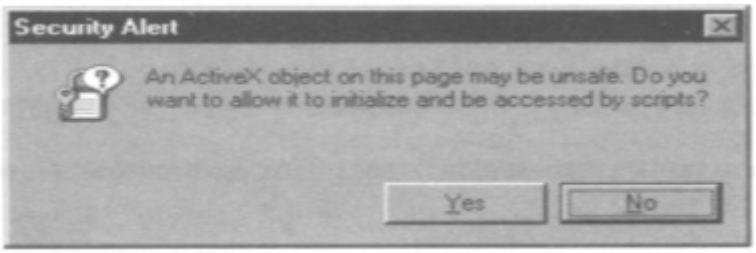

图 1.16　Internet Explorer Security Alert 对话框

为了承诺无论是初始化数据还是在脚本中运行代码都没有任何危害（我们能做的最坏的事情是使一个进程崩溃），我们的 CalcPi 类需要实现“Safe For Scripting”和“Safe For Initialization”组件类别。我们使用 ATL 的 CATEGORY_MAP 和适当的组件类别唯一标识符（component category unique IDs，CATID）来做到这一点：

```
class ATL_NO_VTABLE CCalcPi : ... {
...

public:
BEGIN_CATEGORY_MAP(CCalcPi)
 IMPLEMENTED_CATEGORY(CATID_SafeForScripting)
 IMPLEMENTED_CATEGORY(CATID_SafeForlnitializing)
END_CATEGORY_MAPO
```

```
...
};
```

组件类别不仅有利于向 IE 作出承诺，也有利于向任意的客户发布类，同时还有利于在客户创建实例之前对客户的功能提出要求。更多有关 ATL 对组件类别的支持信息，请参阅第 4 章。

1.11　添加用户界面

COM **控制**(controls)**是那些能够自己提供用户界面**（user interface，UI）**的对象，它们的 UI 与客户紧密集成。**ATL **通过** CComControl **基类和其他各种** IXxxImp1 **基类来提供对** COM **控制的广泛支持。这些基类解决了基本控制的绝大部分细节问题（虽然对高级特性而言还有很大的发挥空间，详见第** 10 **章）。当生成** CalcPi **类时，我们在** ATL Object Wizard **中的控制（**Controls**）选择区中选择完全控制（**Full Control**）或者轻量控制（**Lite Control**），此时仅仅实现** OnDraw **函数就可以提供** UI**：**

```
HRESULT CCalcPi::OnDraw(ATL_DRAWINFO& di) {
 CComBSTR  bstrPi;
 if(SUCCEEDED(this->CalcPi(&bstrPi)) ) {
    USES_CONVERSION;
    DrawText(di.hdcDraw, OLE2CT(bstrPi), -1, (RECT*)di.prcBounds,
                 DT_SINGLELINE I DT_CENTER I DT_VCENTER);
 }

 return S_OK;
}
```

向导同时生成一个 HTML **页面的示例，在例子中我把它扩大到占据整个浏览器窗口，并且设置位数的初始值为** 50**。**

```
<HTML>
<HEAD>
<TITLE>ATL 3.0 test page for object CalcPi</TITLE>
```

```
</HEAD>
<BODY>
<OBJECT CLASSID="CLSID:DEC22F37-DD78-11D1-97FD-O06008243CSC"
          ID="CalcPi"
          height=lO0% width=lO0%>
    <param name=digits value=50>
</OB3ECT>
</BODY>
</HTML>
```

在 IE 中显示这个页面将产生控制的一个视图（如图 1.17）。

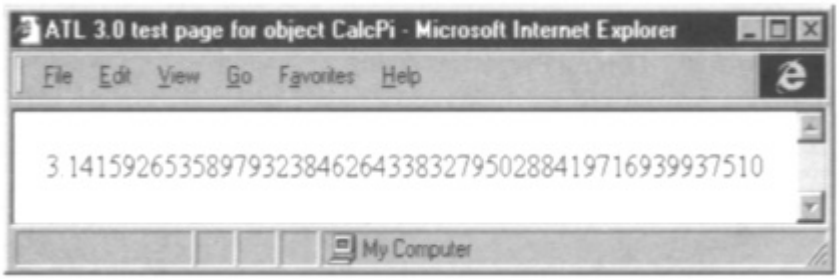

图 1.17　Internet Explorer 中的 CalcPi 控制

关于 ATL 建立控制的更多内容请看第 10 章。

1.12　容纳控制

如果想容纳一个控制，我们可以使用 ATL 对容纳控制的支持来实现。例如，CAxDialogImpl 中的 Ax 表示 ActiveX 控制，并且表示这个对话框能够容纳控制。为了在对话框中容纳控制，右键点击对话框资源并且选择插入 ActiveX 控制（Insert ActiveX Control）。这样会弹出一个对话框，它列出了当前系统中所有安装的控制，如图 1.18 所示。一旦插入了一个控制，我们就可以右键点击它并且设置它的属性，如图 1.19 所示。同样通过点击右键，我们可以选择处理一个控制的事件，如图 1.20 所示。

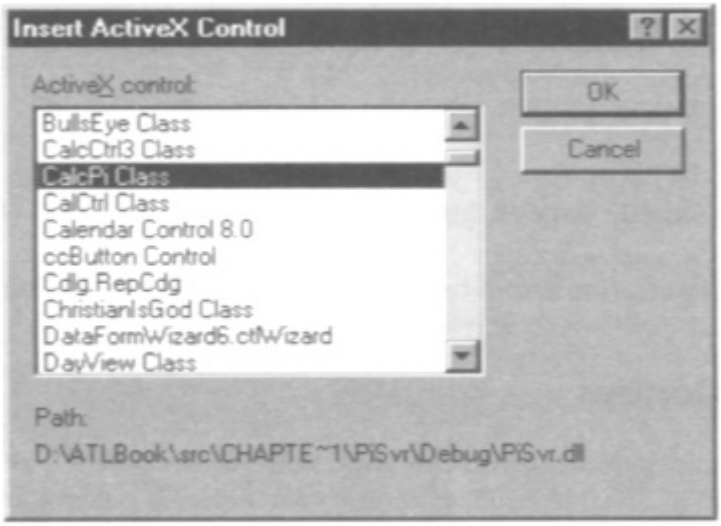

图 1.18　Insert ActiveX Control 对话框

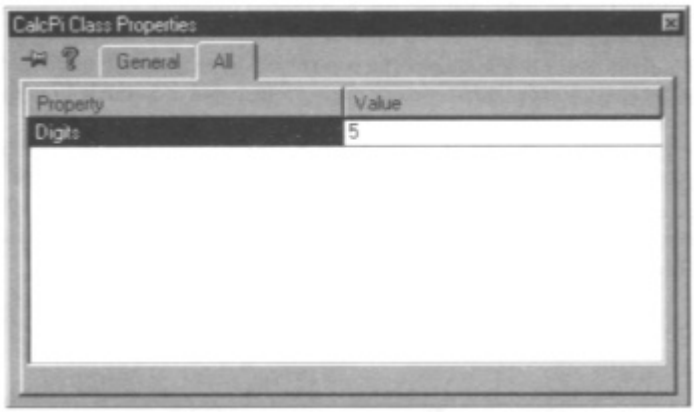

图 1.19　Control Properties 对话框

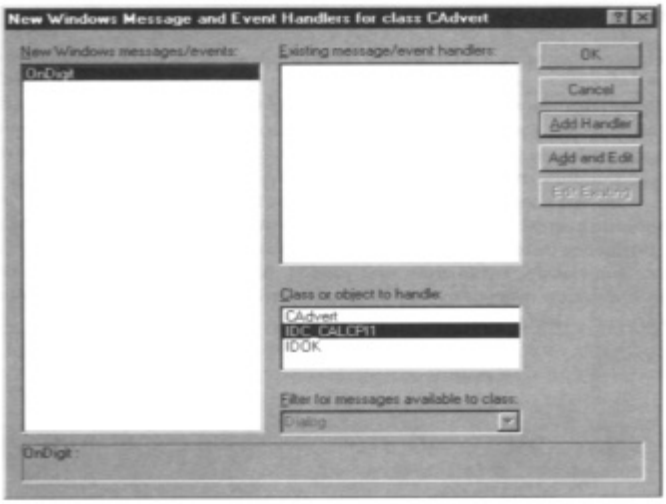

图 1.20　选择处理哪个控制事件

在显示对话框时，控制将被创建，并且按照开发时设置好的属性进行初始化。图 1.21 **展示了一个容纳一个控制的对话框的例子。**

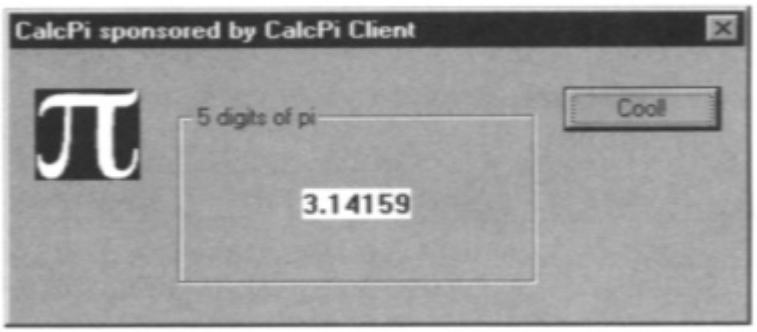

图 1.21　容纳 COM 控制的对话框

ATL **提供的“对容纳** ATL **控制的支持”不仅在于对话框中，也可以在其他窗口中和那些“包含以对话框资源作为** UI **”的控制中（被称为复合控制，**composite control**），还可以在那些“包含以** HTML **作为资源的** UI **”的控制中（被称为** HTML **控制）。有关包含控制的更多内容请参见第** 11 **章。**

1.13　总结

本章我们旋风般地浏览了向导提供的某些 ATL **功能，同时还有一些** ATL **基本接口的实现。我们必须清楚，即使有向导的帮助，**ATL **仍不能替代坚实的** COM **知识。我们仍然**

必须知道如何设计和实现自己的接口。在阅读本书剩余部分的时候，您会明白，我们仍然需要了解接口指针、引用计数、运行时刻类型发现、线程、永久性和其他方面的内容。ATL 能够帮助我们，但我们还是必须了解 COM。

同样应该明白，向导也不能替代 ATL 知识。在本章中介绍的每个 ATL 专题，有十个以上显著的细节、扩展和缺陷。虽然向导为我们节省了手工录入工作，但是实际上除此之外它什么事情也做不了。它无法确保我们的设计和实现能够达到目标，这是我们自己的责任。