Autodesk 官方最新的.NET 教程(C#)

第 1 章 Hello World: 访问 ObjectARX .NET 封装类

在这一章中，我们将使用 Visual Studio .NET 来创建一个新的类库工程。通过这个工程，你可以创建一个能被 AutoCAD 装载的.NET dll 文件。这个 dll 文件会向 AutoCAD 加入一个名为“HelloWorld”的新命令。当用户运行这个命令后，在 AutoCAD 命令行上将显示“Hello World”文本。

1) 启动 Visual Studio.NET，选择”文件>新建>工程”(File> New> Project)。在新建工程对话框中选择工程类型为”Visual C#工程”，然后选择”类库”模板，在工程名字框中输入”Lab1”，然后选择工程存放的位置。点击确定按钮来创建工程。

2) 在工程的 Class1.cs 文件中，一个公有类“Class1”已经被系统自动创建了。接下来向这个类加入命令。要加入命令，你必须使用 AutoCAD .NET 托管封装类。这些托管封装类包含在两个托管模块中。要加入对这两个托管模块的引用，请用鼠标右键单击”引用”然后选择”添加引用”。在弹出的”添加引用”对话框中选择”浏览”。在”选择组件”对话框中，选择 AutoCAD 2006 的安装目录（这里假定为 C:\Program Files\AutoCAD 2006\），在这个目录下找到“acdbmgd.dll”然后选择并打开它。再一次选择”浏览”，在 AutoCAD 2006 的安装目录下找到“acmgd.dll”并打开它。当这两个组件被加入后，请单击”添加引用” 对话框中的”确定”按钮。正如它们的名字所表示的，acdbmgd.dll 包含 ObjectDBX 托管类，而 acmgd.dll 包含 AutoCAD 托管类。

3) 使用对象浏览器（Visual Studio.NET 的”查看>其它窗口>对象浏览器”菜单项）来浏览加入的两个托管模块所提供的类。请展开“AutoCAD .NET Managed Wrapper”对象（在对象浏览器中显示为 acmgd），在整个教程中我们将使用这个对象中的类。在本章中，我们将使用“Autodesk.AutoCAD.EditorInput.Editor”类的一个实例来在 AutoCAD 命令行中显示文本。请再展开“ObjectDBX .NET Managed Wrapper” 对象（在对象浏览器中显示为 acdbmgd），这个对象中的类将被用来访问和编辑 AutoCAD 图形中的实体（这部分内容将在以后的章节中介绍）。

4) 引用了 ObjectARX .NET 封装类后，我们就可以导入它们。在 Class1 类的声明语句（位于 Class1.cs 文件的顶部的）之前，导入 ApplicationServices, EditorInput 和 Runtime 命名空间。

```
using Autodesk.AutoCAD.ApplicationServices;
```

```
using Autodesk.AutoCAD.EditorInput;
using Autodesk.AutoCAD.Runtime;
```

5) 接下来在类 Class1 中加入命令。要加入能在 AutoCAD 中调用的命令，你必须使用“CommandMethod”属性。这个属性由 Runtime 命名空间提供。在类 Class1 中加入下列属性和函数。

```
[CommandMethod("HelloWorld")]
   public void HelloWorld()
{ }
```

6) 当“HelloWorld”命令在 AutoCAD 中运行的时候，上面定义的 HelloWorld 函数就会被调用。在这个函数中，一个 Editor 类的实例将被创建。Editor 类拥有访问 AutoCAD 命令行的相关方法，它还包括选择对象和其它一些重要的功能。AutoCAD 当前活动文档的 Editor 对象可以使用 Application 类来访问。当 Editor 对象被创建后，你可以使用它的 WriteMessage 方法在命令行中显示“Hello World”文本。在 HelloWorld 函数中加入以下代码：

```
Editor ed = Application.DocumentManager.MdiActiveDocument.Editor;
ed.WriteMessage("Hello World");
```

7) 要在 AutoCAD 中调试这个程序，你可以让 Visual Studio.NET 启动一个 AutoCAD 进程。在解决方案管理器中右键单击“Lab1”，然后选择”属性”。在 Lab1 的属性页对话框中，选择”配置属性>调试”。在”启动”项中，选择”调试模式”为”程序”，在”启动程序”的右边单击省略号按钮然后选择 AutoCAD 2006 安装目录下的 acad.exe。设置好以后，按 F5 来启动一个 AutoCAD 进程。这样就会编译你的程序然后自动启动 AutoCAD，而当编译后有错误的时候就会停止。请修正你可能碰到的任何错误。

8) “NETLOAD”命令被用来装载托管程序。在 AutoCAD 命令行中输入 NETLOAD，会出现”选择.NET 组件”的对话框。选择上面生成的“lab1.dll”然后打开它。

9)在命令行中输入“HellowWorld”。如果一切顺利的话，命令行中将显示“Hello World”文本。切换到 Visual Studio.NET，在 ed.WriteMessage(“Hello World”);语句处加入一个断点。在 AutoCAD 中再次运行 HelloWorld 命令，你会注意到你可以跟踪代码的运行。Visul Studio.NET 的”调试”菜单有好几项可以用来跟踪程序的运行。

如果有时间的话，请浏览一下 CommandMethod 属性。你会发现它有七种不同的形式。在上面的例子中，我们使用了最简单的形式，它只有一个输入参数（命令的名字）。你可以使用其它的形式来控制命令的工作方式，例如你可以确定命令组的名字、全局和局部名字、命令标识（命令如何来运行）等。

第 2 章 .NET AutoCAD 向导及 Editor 类

在第一章中，我们使用的是类库模板，这样就不得不手工加入 acdbmdg. dll 和 acmgd.dll 这两个引用。在这一章中，我们将使用 AutoCAD 托管 C#应用程序向导来创建.NET 工程，它会自动加入以上两个引用。在开始本章之前，你首先得安装 ObjectARX 向导(ObjectARX2006 开发包的\utils\ObjARXWiz\ArxWizards.msi)。

1)启动 Visual Studio .NET，选择”文件>新建>工程”(File> New> Project)。在新建工程对话框中选择工程类型为”Visual C#工程”，然后选择“AutoCAD Managed CS Project Application”模板。在工程名字框中输入”Lab2”，然后选择工程存放的位置。点击确定按钮，“AutoCAD Managed CSharp Application Wizard”对话框将会出现。因为我们不需要使用非托管代码，所以不要选择“Enable Unmanaged Debugging”项。

“Registered Developer Symbol”将会使用你在安装 ObjectARX 向导时输入的值。单击”finish”按钮来创建工程。

2)下面来看一下向导生成的工程。在解决方案浏览器中，你会看到 acdbmgd 和 acmgd 已经被引用了。在 Class.cs 文件中，“Autodesk.AutoCAD.Runtime”命名空间已被导入，工程使用“Registered Developer Symbol”的名字来命名缺省的公有类。向导还为类加入了一个 CommandMethod 属性和一个函数，它们用于 AutoCAD 命令。

3)在前一章中，我们使用一个“Autodesk.AutoCAD.EditorInput.Editor”类的实例对象在 AutoCAD 命令行上输出文本。在这一章中，我们将使用这个类来提示用户在 AutoCAD 图形中选择一个点，然后将用户选择的点的 x,y,z 值显示出来。和前一章一样，请导入 Autodesk.AutoCAD.ApplicationServices 和 Autodesk.AutoCAD.EditorInput 命名空间。

4)把向导生成的 CommandMethod 属性的值改为有意义一些的名字如“selectPoint”（函数的名字可以不用修改）。PromptPointOptions 类用来设置提示字符串和其它的一些控制提示的选项。这个类的一个实例作为参数被传入到 Editor.GetPoint 方法。在函数的开始，实例化这个类，设置字符串参数为“Select a point”。因为 Editor.GetPoint 方法会返回一个 PromptPointResult 类的实例对象，所以我们也要把它实例化。

```
PromptPointOptions prPointOptions = new PromptPointOptions("Select a point");
PromptPointResult prPointRes;
```

5) 接下来实例化一个 Editor 类的对象并使用参数为 PromptPointOptions 对象的 GetPoint 方法。用 GetPoint 方法的返回值来给上面声明的 PromptPointResult 对象赋值。赋值好以后，我们可以测试 PromptPointResult 对象的状态，如果不是 OK 就返回。

```
prPointRes = ed.GetPoint(prPointOptions);
       if (prPointRes.Status != PromptStatus.OK)
       {
           ed.WriteMessage("Error");
}
```

6)如果 PromptPointResult 对象返回了一个有效的点,我们就可以使用 WriteMessage 方法把结果输出到命令行。PromptPointResult.Value 的 ToString 方法使输出非常容易：

```
ed.WriteMessage("You selected point "
prPointRes.Value.ToString)
```

7)按 F5 来运行一个调试 AutoCAD 的进程。（注意：向导已经设置好用 acad.exe 来调试）在 AutoCAD 命令行中输入 NETLOAD，选择 Lab2.dll 并打开。在命令行中输入你起的命令名字（selectPoint）。在选择点的提示下，单击图形中的任一点。如果一切正常的话，你可以在命令行中看到你所选的点的坐标值。在 Class.cs 文件的“ed.WriteMessage("Error");”行加入断点，然后再次运行 selectPoint 命令。这一次，在选择点的提示下按 ESC 键而不是选择一个点。PromptPointResult 对象的状态就不是 OK 了，所以上面代码中的 if 语句就会被执行，“ed.WriteMessage("Error")”;语句就会被调用。

8) 接下来我们将加入另外一个命令，它可以获取两个点之间的距离。向导没有添加命令的功能，所以我们必须手工添加。在 Class.cs 文件的选择点的函数（getPoint）下面添加一个名为 getDistance 的新命令。加入命令的方法请参考上一章的内容或本章的源代码，这里就不列出了。使用 CommandMethod 属性并使字符串参数为“getdistance”或其它类似的名字。在命令的函数中使用 PromptDistanceOptions 代替 PromptPointOptions。当然 GetDistance 方法的返回值是一个 PromptDoubleResult 类的实例对象，所以请用 PromptDoubleResult 来代替 PromptPointResult：

```
PromptDistanceOptions prDistOptions = new
     PromptDistanceOptions("Find distance, select first point:");
          PromptDoubleResult prDistRes;
prDistRes = ed.GetDistance(prDistOptions);
```

9) 和前面的命令一样，也可以测试 PromptDoubleResult 的状态，然后用 WriteMessage 方法在命令行中显示值。

```
 if (prDistRes.Status != PromptStatus.OK)
         {
             ed.WriteMessage("Error");
         }
       else
          {
ed.WriteMessage("The distance is: " + prDistRes.Value.ToString());

  }
```

第 3 章 数据库基础: 创建我们自己的 Employee 对象

打开 Lab3 文件夹下的 Lab3 工程文件，或或接着 Lab2 的代码。在这一章中，我们将创建一个‘Employee 对象’（包括一个圆，一个椭圆和一个多行文本对象），这个对象属于一个自定义的 EmployeeBlock’块（这个块驻留在‘EmployeeLayer’层，当在模型空间插入这个块的时候，‘EmployeeLayer’层就会拥有这个块的一个块索引）。本章的每一个步骤中的代码都可以运行，这样做的目的可以使你更清楚地知道每一部分代码完成的功能。第一步将简要说明一下如何在模型空间创建一个圆。

这一章的重点是在 AutoCAD 中访问数据库的基础。主要内容包括事务处理（Transaction）、对象 Id（ObjectId）、符号表（symbol tables，如块表 BlockTable 和层表 LayerTable）以及对象引用。使用的其它一些对象如颜色 Color、三维点 Point3d 和三维向量 Vector3d，都和各自的步骤有关，但重点应该放在数据库基础上。

1)创建一个名为‘CREATE’的命令，它调用函数 CreateEmployee()。这个函数用来在模型空间(MODELSPACE)的（10，10，0）点处创建一个半径为 2.0 的圆：

```
[CommandMethod("test")]
public void createCircle()
{ //首先声明我们要使用的对象
Circle circle; //这个是我们要加入到模型空间的圆
BlockTableRecord btr;//要加入圆，我们必须打开模型空间
BlockTable bt; //要打开模型空间，我们必须通过块表(BlockTable)来访问它

//我们使用一个名为‘Transaction’的对象，把函数中有关数据库的操作封装起来
Transaction trans;
 //使用 TransactionManager 的 StartTransaction()成员来开始事务处理
trans =
HostApplicationServices.WorkingDatabase.TransactionManager.StartTransaction();
```

```
 //现在创建圆……请仔细看这些参数——注意创建 Point3d 对象的‘New’和 Vector3d 的静态成员 ZAxis
circle = new Circle(new Point3d(10, 10, 0), Vector3d.ZAxis, 2);
bt =
(BlockTable)trans.GetObject(HostApplicationServices.WorkingDatabase.BlockTableId, OpenMode.ForRead);
 //使用当前的空间 Id 来获取块表记录——注意我们是打开它用来写入
btr =
(BlockTableRecord)trans.GetObject(HostApplicationServices.WorkingDatabase.CurrentSpaceId,OpenMode.ForWrite );
 //现在使用 btr 对象来加入圆
btr.AppendEntity(circle);
trans.AddNewlyCreatedDBObject(circle, true); //并确定事务处理知道要加入圆!
 //一旦完成以上操作，我们就提交事务处理，这样以上所做的改变就被保存了……
trans.Commit();
 //…然后销毁事务处理，因为我们已经完成了相关的操作（事务处理不是数据库驻留对象，可以销毁）
trans.Dispose();

}
```

请仔细阅读一下上面的代码块的结构，可以通过注释来了解相关的细节。

注意：要编译代码，你必须导入 Autodesk.AutoCAD.DatabaseServices 和 Autodesk.AutoCAD.Geometry 命名空间

运行这个函数来看看它是否可行。应该会在图形中创建一个在(10,10,0)处的半径为 2.0 的白色的圆。

2) 我们可以减少代码的输入量，这可以通过声明一个 Database 变量代替 HostApplicationServices.WorkingDatabase 来实现：

```
Database db = HostApplicationServices.WorkingDatabase;
```

使用这个变量来代替在代码中出现的 HostApplicationServices.WorkingDatabase。

3)在上面的代码中，我们没有使用任何异常处理，而异常处理对一个正确的.NET 应用程序来说是非常重要的。我们要养成使用异常处理的好习惯，所以让我们在这个函数中加入 try-catch-finally。

4)为了使代码紧凑，我们可以把许多变量的声明和初始化放在同一个语句中。现在，你的代码看起来应该是这样的：

```
 [CommandMethod("CREATE")]
public void CREATEEMPLOYEE()
{ Database db = HostApplicationServices.WorkingDatabase;
Transaction trans = db.TransactionManager.StartTransaction();
    try
   {
Circle circle = new Circle(new Point3d(10, 10, 0), Vector3d.ZAxis, 2);
BlockTable bt = (BlockTable)trans.GetObject(db.BlockTableId, OpenMode.ForRead);
```

```
BlockTableRecord btr =
(BlockTableRecord)trans.GetObject(HostApplicationServices.WorkingDatabase.Curre
ntSpaceId,OpenMode.ForWrite);
btr.AppendEntity(circle);
trans.AddNewlyCreatedDBObject(circle, true);
trans.Commit();
  }

  catch
  {
ed.WriteMessage("Error ");
 }
finally
{
trans.Dispose();
}
}
End Function
```

运行你的代码来进行测试……

上面的 catch 块只显示一个错误信息。实际的清理工作是在 finally 块中进行的。这样做的理由是如果在事务处理被提交（Commit()）之前，Dispose()被调用的话，事务处理会被 销毁。我们认为如果在 trans.Commit()之前出现任何错误的话，你应该销毁事务处理（因为 Commit 将永远不会被调用）。如果在 Dispose()之前调用了 Commit()，也就是说没有任何错误发生，那么事务处理将会被提交给数据库。

所以基于上面的分析，Catch 块其实并不是必须的，因为它只用来通知用户程序出现了一个错误。它将在下面的代码中被去掉。

5)现在让我们在 Employee 加入剩下的部分：椭圆和多行文本的实例。

多行文本实体：

中心点应该与圆心的创建一样：

（建议：创建一个名为‘center’而值为 10,10,0 的 Point3d 变量来表示中心点）

多行文本的内容可以是你的名字。

椭圆（提示：你可以先看一下 Ellipse 的构造函数）

法向量应该沿着 Z 轴（请查看 Vector3d 类型）

主轴设为 Vector3d(3,0,0)（提示：不要忘了用 new）

半径比例设为 0.5

椭圆还必须闭合（也就是说，开始和结束点必须相同）

运行你的代码来进行测试……应该可以生成一个圆、一个椭圆和一个中心点在 10,10,0 的多行文本。

注意：和事务处理对象有关的.NET API 中的 Try-Catch-Finally 块结构，应该是异常观察者。实际上我们是在 try 块中实例化对象的，但没有显式地销毁它们。当产生异常的时候可能会产生问题，特别是当观察者注意到我们实际上用的是封装的非托管对象！记住，当资源不再使用的时候，垃圾收集机制就会回收内存。垃圾收集机制会不时的调用封装类的 Dispose()方法，删除非托管对象。

这里还要注意的是 Dispose()作用于封装的非托管类对象的方式取决于对象是否是数据库驻留对象。由非数据库驻留对象调用的 Dispose()会删除非托管对象，而由数据库驻留对象调用的 Dispose()只是关闭它们。

<!--[if !supportLineBreakNewLine]-->
<!--[endif]-->

6) 接下来让我们来创建一个新的函数，它用来新建一个颜色为黄色，名字为“EmployeeLayer” 的 AutoCAD 层。

这个函数应该检查是否这个层已经存在，但不管这个层是否存在，函数都应该返回“EmployeeLayer”的 ObjectId。下面是这个函数的代码：

```
public ObjectId CreateLayer()
{
ObjectId layerId; //它返回函数的值
Database db = HostApplicationServices.WorkingDatabase;
Transaction trans = db.TransactionManager.StartTransaction();
//首先取得层表……
LayerTable lt = (LayerTable)trans.GetObject(db.LayerTableId, OpenMode.ForWrite);
//检查 EmployeeLayer 层是否存在……
if (lt.Has("EmployeeLayer"))
{
        layerId = lt["EmployeeLayer"];
}
else
{
//如果 EmployeeLayer 层不存在，就创建它
LayerTableRecord ltr = new LayerTableRecord();
ltr.Name = "EmployeeLayer"; //设置层的名字
ltr.Color = Color.FromColorIndex(ColorMethod.ByAci, 2);
layerId = lt.Add(ltr);
trans.AddNewlyCreatedDBObject(ltr, true);
}

trans.Commit();
trans.Dispose();
return layerId;
}
```

是不是觉得这个函数的基本结构与在模型空间加入实体的代码比较类似？访问数据库的方法都是这样的：使用事务处理来获取数据库对象，在符号表（模型空间所在的块表也是符号表之一）中加入实体，然后让事务处理知道。

7)在这个函数中加入异常处理，就像在 CreateEmployee 函数中的一样。

8)接下来，改变新建层的颜色。下面是实现的代码片断，请把它加入到你的代码中：

```
ltr.Color = Color.FromColorIndex(ColorMethod.ByAci, 2)
```

注意：ColorMethod.ByAci 可以让我们使用 AutoCAD ACI 颜色索引……这里为（表示黄色）。

<!--[if !supportLists]-->9) <!--[endif]-->回到 CreateEmployee()函数，加入把上面创建的几个实体设置到 EmployeeLayer 层的代码。声明

一个类型为 ObjectId 的变量，用 CreateLayer 函数的返回值给它赋值。使用每个实体（文本、圆和椭圆）的 LayerId 属性设置它们所在的层。

例如： text.LayerId = empId

运行代码来查看“EmployeeLayer”层是否已被创建,所有已创建的实体是否都在这一层上（应该显示为黄色）

10)现在为各个实体设置不同的颜色，可以使用 ColorIndex 属性（ColorIndex 属性表示 AutoCAD 的颜色）

圆为红色－1

椭圆为绿色－3

文本为黄色－2

运行代码，看看实体的颜色是否为设置的值，即使这些实体是在“EmployeeLayer”层上。

11) 接下来，我们要在 AutoCAD 数据库中创建一个独立的块，然后把它插入到块表而不是模型空间中。

首先把 CreateEmployee 函数的名字改为 CreateEmployeeDefinition()。

加入以下代码来创建一个独立的块：

```
BlockTableRecord newBtr = new BlockTableRecord();
newBtr.Name = "EmployeeBlock";
newBtrId = bt.Add(newBtr);
trans.AddNewlyCreatedDBObject(newBtr, true);
```

12)现在，请稍微改动一下加入实体到模型空间的代码（改为加入块到块表中，记得加入前要打开块表）。

现在运行代码，然后使用 INSERT 命令来检查是否可以正确插入这个块。

13)最后，我们要创建一个位于模型空间的块索引，它表示上面创建的块的一个实例。这一步留给大家练习。

下面是你要遵循的最基本的步骤：

<!--[if !supportLists]-->A) <!--[endif]-->创建一个名为 CreateEmployee 新的函数

<!--[if !supportLists]-->B) <!--[endif]-->把命令属性“CREATE”移动到 CreateEmployee()

<!--[if !supportLists]-->C) <!--[endif]-->修改 CreateEmployeeDefintion()来返回新创建的块“EmployeeBlock”的 ObjectId，操作的步骤请参考 CreateLayer()的作法。

<!--[if !supportLists]-->D) <!--[endif]-->你需要修改 CreateEmployeeDefintion()来查看块表中是否已包含“EmployeeBlock”块,如果包含这个块,则返回它的 ObjectId(做法与 CreateLayer()一样）。

提示：把‘bt’的声明语句移动到 try 块的顶部，使用 BlockTable.Has()方法，把其它的代码移动到 else 语句：

```
try
   {
      //获取 BlockTable 对象
BlockTable bt = (BlockTable)trans.GetObject(db.BlockTableId, OpenMode.ForWrite);
      if ((bt.Has("EmployeeBlock")))
          {
            newBtrId =bt["EmployeeBlock"];
          }
         else
```

{ …

<!--[if !supportLists]-->E) <!--[endif]-->在新创建的 CreateEmployee()函数中创建一个新的 BlockReference 对象，并把它加入到模型空间。提示：我们可以使用 CreateEmployeeDefinition()中引用模型空间的代码，这些代码在这里不需要了

<!--[if !supportLists]-->F) <!--[endif]-->在 CreateEmployee 中调用 CreateEmployeeDefinition()函数，使上面生成的 BlockReference 对象的 BlockTableRecord()指向 CreateEmployeeDefinition()函数。提示：请参考 BlockReference 的构造函数。

附加的问题：

让我们来看一下代码的运行情况，执行命令会生成一个 EmployeeBlock 的块索引，你会看到它被插入到 20,20,0 而不是 10,10,0。为什么？

如果你知道原因，那么怎样才能使块索引插入到正确的点？

当你用 List 命令查看块索引时，它会告诉你它位于 0 层（或者当命令运行时位于当前层）。为什么？

怎样才能让块索引总是位于 EmployeeLayer 层？

第 4 章 数据库基础 2: 添加自定义数据

在这一章中，我们将创建一个新的字典对象，它用来表示我们雇员就职的 ‘Acme 公司‘（呵呵，当然是虚构的一家公司）的部门。这个“部门”字典对象将包含一个表示部门经理的记录。我们还会加入代码到雇员创建过程，这个过程会加入一个索引到雇员工作的部门。

我们要说明的是如何在 DWG 文件中创建自定义数据，包括“每个图形”的自定义数据和“每个实体”的自定义数据。“每个图形”的自定义数据是指只在整个图形中加入一次的数据，它表示对象可以引用的单一类型或特性。“每个实体”的自定义数据是指是为特定的对象或数据库中的实体加入的数据。

在下面的示例中，我们将加入“每个图形”的自定义数据到命名对象字典(简称 NOD)。NOD 存在于每一个 DWG 文件中。“每个实体”的自定义数据加入到一个名为“扩展字典”的字典（可选）中，它表示每一个雇员。每一个由 DBObject 派生的对象都拥有存储自定义数据的扩展字典。而在我们的示例中将包含这种自定义数据如名字、薪水和部门。

因此这一章的重点是字典对象和扩展记录（XRecord），它们是我们用来表示自定义数据的容器。

首先让我们来创建表示公司的条目。在本章的前几个步骤中，我们将创建如下所示的部门层次结构：

NOD－命名对象字典

ACME_DIVISION－自定义公司字典

销售(Sales) －部门字典

部门经理－部门条目

请打开 Lab4 文件夹下的 Lab4 工程，或接着 Lab3 的代码。

<!--[if !supportLists]-->1) <!--[endif]-->我们首先要做的是定义一个新的函数，它用来在命名对象字典(NOD)中创建公司字典对象。为这个函数取名为 CreateDivision(),, 并使用命令属性来定义 CREATEDIVISION 命令。

下面是这个函数的代码，它的形式非常简单，只是用来在 NOD 中创建一个 ACME_DIVISION(用来表示公司）

```
[CommandMethod("CreateDivision")]
public void CreateDivision()
{
      Database db = HostApplicationServices.WorkingDatabase;
      Transaction trans = db.TransactionManager.StartTransaction();

      try
      {
            //首先，获取 NOD……
            DBDictionary NOD =
(DBDictionary)trans.GetObject(db.NamedObjectsDictionaryId, OpenMode.ForWrite);
            //定义一个公司级别的字典
            DBDictionary acmeDict;
            try
            {
                  //如果 ACME_DIVISION 不存在，则转到 catch 块，这里什么也不做
                  acmeDict =
(DBDictionary)trans.GetObject(NOD.GetAt("ACME_DIVISION"), OpenMode.ForRead);
            }
            catch
            {
                  //如果 ACME_DIVISION 不存在，则创建它并把它加入到 NOD 中……
                  acmeDict = new DBDictionary();
                  NOD.SetAt("ACME_DIVISION", acmeDict);
                  trans.AddNewlyCreatedDBObject(acmeDict, true);
            }
            trans.Commit();
      }
      finally
      {
            trans.Dispose();
      }
}
```

请仔细阅读一下上面的代码块的结构，可以通过注释来了解相关的细节。特别要注意的是我们是如何用一个 try-catch 块来处理 ACME_DIVISION 是否存在？如果 ACME_DIVISION 字典不存在，GetObject()将会抛出异常，catch 块被执行，它会创建一个新的字典。
运行这个函数来看它是否可行。可以使用数据库查看工具来检查字典是否已被加入（建议使用 ARX SDK 的 ArxDbg 工具）
<!--[if !supportLists]-->2) <!--[endif]-->接下来，我们要在 ACME_DIVISION 字典中加入销售(Sales)条目。销售(Sales)条目同样也是一个字典。由于销售(Sales)字典与 ACME_DIVISION 字典的关系如同 ACME_DIVISION 字典与 NOD，所以代码是类似的。定义下面的代码部分在 ACME_DIVISION 字典中创建一个名为’Sales’的字典。
代码提示：

```
                DBDictionary divDict;
                try
        {
                divDict = (DBDictionary)trans.GetObject(acmeDict.GetAt("Sales"),
OpenMode.ForWrite);
        }
        catch
           …
```

运行函数来看‘Sales’条目是否已加入到 ACME_DIVISION 字典。

<!--[if !supportLists]-->3) <!--[endif]-->现在我们要在这个字典中加入一个特殊的记录，它可以包含任意的自定义数据。我们要加入的数据类型为扩展记录（XRecord），它可以包含任何东西，因此我们可以让它包含 ResultBuffer 类的对象（就是有些人可能非常熟悉的‘resbuf’）。ResultBuffer 可以存储不同类型的预定义数据。扩展记录存储任意数目的 ResultBuffer 关系列表，因此可能会很大。下表是可以包含在 ResultBuffer 中一些数据类型（位于 Database 类的 DxfCode 枚举中）：

Start	0	
Text	1	
XRefPath	1	
ShapeName	2	
BlockName	2	
AttributeTag	2	
SymbolTableName	2	
MstyleName	2	
SymTableRecName	2	
AttributePrompt	3	
DimStyleName	3	
LinetypeProse	3	
TextFontFile	3	

在下面的代码部分，我们将创建只包含一个 ResultBuffer 的扩展记录。这个 ResultBuffer 包含一个单一的的字符串值，它表示’Sales’部门的部门经理的名字。我们使用和加入字典一样的方法加入扩展记录。唯一的区别是扩展记录与字典的不同：

```
            mgrXRec = new Xrecord();
 mgrXRec.Data = new ResultBuffer(new TypedValue((int)DxfCode.Text, "Randolph P.
 Brokwell"));
```

请看一下我们是怎样使用 new 来创建一个新的扩展记录。但我们也使用了 new 来创建一个 ResultBuffer，传入的参数是一个名为‘TypedValue’的对象。‘TypedValue’对象和 C++中 resbuf 的成员‘restype’是类似的。这个对象一般表示一个特定类型的 DXF 值，我们使用它来组装诸如扩展数据或扩展记录之类的通用数据容器。在这里，我们简单地使用 DxfCode.Text 键值和“Randolph P. Brokwell”数据值来定义一个 TypedValue，然后把它作为单一的参数传入 ResultBuffer 构造函数(由 new 来调用)中。

XRecord 的’Data’属性实际上正是扩展记录链的第一个 ResultBuffer，我们使用它来表示扩展记录链是从什么地方开始的。

所以接下来的代码块看起来和前面两个非常相似：

```
Xrecord mgrXRec;
            try
            {
                  mgrXRec =
(Xrecord)trans.GetObject(divDict.GetAt("Department Manager"),
OpenMode.ForWrite);
            }
            catch
            {
                  mgrXRec = new Xrecord();
                  mgrXRec.Data = new ResultBuffer(new
TypedValue((int)DxfCode.Text, "Randolph P. Brokwell"));
                  divDict.SetAt("Department Manager", mgrXRec);
                  trans.AddNewlyCreatedDBObject(mgrXRec, true);
            }
```

运行函数并使用数据库查看工具来确定部门经理已被加入到’Sales’字典。

4) 我们已经定义了公司字典，现在我们要把每个雇员的数据加入到前一章定义的块索引中。我们要加入的数据是：名字、薪水和雇员所属的部门。要加入这些数据，我们要同前几个步骤一样使用扩展记录。因为我们要加入三个条目，所以我们要使扩展记录可以把这些数据联系在一起。

一般来说，扩展记录只能存在于字典中。而我们要为每个雇员加入这些数据（就是本章开头所讲的“每个图形”的自定义数据和“每个实体”的自定义数据），那应该怎么做呢？答案就是：每一个对象或 AutoCAD 中的实体实际上都有一个名为’扩展字典（Extension Dictionary）’的可选字典。我们可以把扩展记录直接加入到这个字典中。

请回到我们在上一章创建的 CreateEmployee()函数。这个函数是我们创建块索引的地方。让我们像前面的步骤一样来创建一个新的扩展记录。因为我们要加入 3 个条目，因此我们既可以使用 ResultBuffer 的 Add 方法（它会在扩展记录链中加入一个链接），也可以利用 ResultBuffer 的构造函数（它的一种构造函数可以输入可变数量的参数）。

无论用哪一种方法，请在 CreateEmployee()函数中使用 ResultBuffer 来创建一个新的 XRecord，ResultBuffer 包括以下的类型和值：

```
Text  -  “Earnest Shackleton”          (或是你选择的其它雇员的名字)
Real  -  72000                  或者更多的薪水 J
Text  -  “Sales”                   雇员所在的部门
```

5) 要把上面的扩展记录加入到块索引，我们必须把它加入到扩展字典。通常这个字典是不存在的，除非它被明确地创建，块索引就是这种情况。要给一个对象创建扩展字典，你要调用它的成员‘CreateExtensionDictionary()’。这个函数不返回任何值，所以要访问它创建的扩展字典，你还得使用对象的‘ExtensionDictionary’属性。你可以使用类似于以下的代码来创建并访问扩展字典：

```
br.CreateExtensionDictionary();
```

```
        DBDictionary brExtDict  =
(DBDictionary)trans.GetObject(br.ExtensionDictionary, OpenMode.ForWrite, false);
```

由于扩展字典也是字典，我们可以和第 3 步一样在扩展字典中加入扩展记录。请完成有关的代码来创建和访问块索引的扩展字典，加入你在第 4 步中创建的扩展记录，然后把扩展记录加入到事务处理。

6) 返回到 NOD……因为在 NOD 中创建公司字典只需要一次(就像创建 Employee 块一样)，因此我们应该把 CreateDivision 函数的命令属性去掉，而在 CreateEmployeeDefinition()中调用这个函数。请自己完成这些改变。当所有这些都做完后，当 CREATE 命令第一次运行的时候，所有的函数都会被调用。

7) 下面的步骤和上面的无关。我们将创建一个函数来遍历模型空间，以用来查找加入的 Employee 对象（这里其实是块索引）的数目。在 VB.NET 或 C#中，我们可以把模型空间块表记录（ModelSpace BlockTableRecord）当作一个集合，这样就可以使用 For Each(C#是 foreach)来遍历它。请仔细研究一下下面的代码片断：

VB.NET:

```
            Dim id As ObjectId  ‘首先，定义一个 For 循环要使用的 ObjectId 变量。
            For Each id In btr
                Dim ent As Entity = trans.GetObject(id, OpenMode.ForRead, False)
’打开当前的对象!
```

C#:

```
        foreach (ObjectId id in btr)
        {
            Entity ent = (Entity)trans.GetObject(id, OpenMode.ForRead, false);
  //打开当前的对象!
```

一旦我们获得模型空间对象，你们就可以定义一个 ObjectId 变量，然后把它用于 For Each 循环(C#是 foreach)。

现在，我们需要使用一些方法来筛选雇员。我们知道模型空间中的对象都是实体，但不全是雇员。我们需要使用一些方法来加以区分。在这里，我们可以使用 VB.NET 的 TypeOf 关键字并用 CType 进行类型转换（C#是 GetType 函数和 typeof）：

VB.NET:

```
                If TypeOf ent Is BlockReference Then
            Dim br As BlockReference = CType(ent, BlockReference)
            …
```

C#:

```
        If(ent.GetType() == typeof(BlockReference))
            BlockReference br = (BlockReference)ent;
```

上面讲的概念对于 AutoCAD 编程是很重要的，因为容器对象经常包含不同类型的对象。你会在 AutoCAD 程序的开发中经常碰到这种类型转化。

请定义一个名为 EmployeeCount()的函数，函数的结构如上所示，它用来统计模型空间中的块索引的数目。这个函数不会输出任何东西，但你可以使用逐步调试程序来查看整数变量的增加（每发现一个块索引对象）。

8) 接下来，为了把结果输出到命令行，我们需要使用 Application.DocumentManager.MdiActiveDocument.Editor对象的服务。要使用它，请加入下面的代码：

```
Imports Autodesk.AutoCAD.EditorInput
Imports Autodesk.AutoCAD.ApplicationServices
```

在函数的内部：

```
Editor ed = Application.DocumentManager.MdiActiveDocument.Editor;
```

最后，在循环的后面确定找到了多少个块索引：

```
ed.WriteMessage("Employees Found: " + nEmployeeCount.ToString());
```

第四章结束

下面的代码片断演示了怎样获取 Employee 对象的所有内容，包括 ACME_DIVISION 字典中的部门经理的名字。这部分要在后面的章节中使用，但因为它和本章有关，因此我们把它放在本章作介绍。如果有时间的话，请阅读一下其中的代码来看看它是怎么使用的。它可以被直接放到你的类中并可以运行。命令的名字是 PRINTOUTEMPLOYEE。ListEmployee()函数接收一个 ObjectId 参数，它通过一个 ref 类型的字符串数组返回值（包含相应的雇员数据）。调用它的 PrintoutEmployee()函数只是用来在命令行中输出这些数据。

我们需要一个遍历并显示所有雇员数据的命令。

```
public static void ListEmployee(ObjectId employeeId, ref string[]
saEmployeeList)
{
       int nEmployeeDataCount = 0;
       Database db = HostApplicationServices.WorkingDatabase;
       Transaction trans = db.TransactionManager.StartTransaction(); //开始事务
处理。
       try
       {
              Entity ent = (Entity)trans.GetObject(employeeId, OpenMode.ForRead,
 false); //打开当前对象!
              if (ent.GetType() == typeof(BlockReference))
              {
//不是所有的块索引都有雇员数据，所以我们要处理错误
                     bool bHasOurDict = true;
                     Xrecord EmployeeXRec = null;
                     try
                     {
                            BlockReference br = (BlockReference)ent;
                            DBDictionary extDict =
(DBDictionary)trans.GetObject(br.ExtensionDictionary, OpenMode.ForRead, false);
```

```
                        EmployeeXRec =
(Xrecord)trans.GetObject(extDict.GetAt("EmployeeData"), OpenMode.ForRead,
false);
                    }
                    catch
                { bHasOurDict = false; //出现了错误……字典或扩展记录不能访问
                }
if (bHasOurDict) //如果获得扩展字典，而又有扩展记录……
                    {
                        // 为雇员列表分配内存
                        saEmployeeList = new String[4];
                        //加入雇员的名字
                        TypedValue resBuf = EmployeeXRec.Data.AsArray()[0];
                        saEmployeeList.SetValue(string.Format("{0}\n",
resBuf.Value), nEmployeeDataCount);
                        nEmployeeDataCount += 1;
                        //加入雇员的薪水
                        resBuf = EmployeeXRec.Data.AsArray()[1];
                        saEmployeeList.SetValue(string.Format("{0}\n",
resBuf.Value), nEmployeeDataCount);
                        nEmployeeDataCount += 1;
                        //加入雇员所在的部门
                        resBuf = EmployeeXRec.Data.AsArray()[2];
                        string str = (string)resBuf.Value;
                        saEmployeeList.SetValue(string.Format("{0}\n",
resBuf.Value), nEmployeeDataCount);
                        nEmployeeDataCount += 1;
                        //现在，让我们从公司字典中获取老板的名字
                        //在 NOD 中找到.
                        DBDictionary NOD =
(DBDictionary)trans.GetObject(db.NamedObjectsDictionaryId, OpenMode.ForRead,
false);
                                DBDictionary acmeDict =
(DBDictionary)trans.GetObject(NOD.GetAt("ACME_DIVISION"), OpenMode.ForRead);
                        //注意我们直接使用扩展数据...
                                DBDictionary salesDict =
(DBDictionary)trans.GetObject(acmeDict.GetAt((string)EmployeeXRec.Data.AsArray()
[2].Value), OpenMode.ForRead);
                        Xrecord salesXRec =
(Xrecord)trans.GetObject(salesDict.GetAt("Department Manager"),
OpenMode.ForRead);
                        //最后，把雇员的数据输出到命令行
                        resBuf = salesXRec.Data.AsArray()[0];
```

```
                        saEmployeeList.SetValue(string.Format("{0}\n",
resBuf.Value), nEmployeeDataCount);
                        nEmployeeDataCount += 1;
                    }
                }
                trans.Commit();
        }
        finally
        {
                trans.Dispose();
        }
}


[CommandMethod("PRINTOUTEMPLOYEE")]
public static  void PrintoutEmployee()
{
        Editor ed = Application.DocumentManager.MdiActiveDocument.Editor;
        //声明我们将在下面使用的工具...
        Database db = HostApplicationServices.WorkingDatabase;
        Transaction trans = db.TransactionManager.StartTransaction();
        try
        {
                //首先，获取块表和模型空间块表记录
BlockTable bt =
(BlockTable)trans.GetObject(HostApplicationServices.WorkingDatabase.BlockTableI
d, OpenMode.ForRead);
                BlockTableRecord btr =
(BlockTableRecord)trans.GetObject(bt[BlockTableRecord.ModelSpace],
OpenMode.ForRead);
                //现在，我们需要把内容输出到命令行。这里可以有一个对象帮助我们：
                        //下面的部分，我们将遍历模型空间:

                foreach (ObjectId id in btr)
                {
                        Entity ent = (Entity)trans.GetObject(id, OpenMode.ForRead,
false); //打开当前对象!
                        if (ent is BlockReference)
                        {
                                string[] saEmployeeList = null;// 这是正确的...定义
新的列表。

                                ListEmployee(id, ref saEmployeeList);
                                if ((saEmployeeList.Length == 4))
```

```
        {
              ed.WriteMessage("Employee Name: {0}", saEmployeeList[0]);
              ed.WriteMessage("Employee Salary: {0}", saEmployeeList[1]);
              ed.WriteMessage("Employee Division: {0}", saEmployeeList[2]);
              ed.WriteMessage("Division Manager: {0}", saEmployeeList[3]);
                          }                       }               }       }
       finally
       {
       }
}
```

第 5 章 用户互操作：提示和选择

背景

提示通常包含一个描述性信息，伴随一个停止以让用户理解所给的信息并输入数据。数据可以通过多种方式被输入，如通过命令行、对话框或 AutoCAD 编辑窗口。给出的提示要遵循一定的格式，格式要与一般的 AutoCAD 提示相一致，这一点是非常重要的。例如，关键字要用“/”号分隔并放在方括号“[]”中，缺省值要放在“<>”内。对于一个 AutoCAD 用户来说，坚持统一的格式将会减少信息理解错误的产生。

当用户在 AutoCAD 命令行中选择一个实体时，实体是使用选择机制被选择的。这种机制包括一个提示，用来让用户知道选择什么并怎样选择（如，窗口或单一实体），然后是一个停顿。

试一下诸如 PINE 这种命令来看一下提示的显示，PEDIT 来看一下使用单一实体或多线来进行选择。

练习

Prompts:

提示：

在本章中，我们将提示输入雇员名字、职位、薪水和部门来创建一个雇员块索引对象。如果输入的部门不存在，我们将提示输入部门经理的名字来创建一个新的部门。在我们继续之前，让我们试着重用以前的代码。

为了进行选择，我们将提示用户在一个窗口中进行选择或选择一个实体，而我们只显示选择集中的雇员对象。

在前面的章节中，我们创建了一个名叫“Earnest Shackleton”的雇员，名字被存储为“EmployeeBlock”块定义（块表记录）中的 MText。如果我们多次插入这个块，那么我们看到的都是同一个雇员的名字。我们怎样才能自定义这个块以使每次插入这个块的时候显示不同雇员的名字？这就要使用块属性的功能了。属性是存储在每一个块索引实例中的文本，并被作为实例的一部分来被显示。属性从存储在块表记录中的属性定义中继承相关的属性。

属性：

让我们来把 MText 实体类型改变为属性定义。在 CreateEmployeeDefinition()函数中，把下面的代码替换

```
    //文本:
  MText text = new MText();
  text.Contents = "Earnest Shackleton";
  text.Location = center;
```

为

```
//属性定义
AttributeDefinition text = new AttributeDefinition(center, "NoName", "Name:",
"Enter Name", db.Textstyle);
text.ColorIndex = 2;
```

试着使用 TEST 命令来测试一下 CreateEmployeeDefinition()函数：

```
[CommandMethod("Test")]
public void Test()
{
CreateEmployeeDefinition();
}
```

你现在应该可以使用 INSERT 命令来插入 EmployeeBlock 块并对每一个实例确定一个雇员名。

当你插入 Employee 块时，请注意一下块插入的位置。它是正好被放置在所选点还是有些偏移？试试怎样修复它。（提示：检查块定义中的圆心）

修改 CreateEmployee ()以重用

1)让我们来修改 CreateEmployee()函数，以让它可以接收名字、薪水、部门和职位并返回创建的雇员块索引的 ObjectId。函数的形式如下（你可以改变参数顺序）

```
public ObjectId CreateEmployee(string name, string division, double salary,
Point3d pos)
```

2) 移除上面函数中的 CommandMethod 属性”CREATE”，这样它就不再是用来创建雇员的命令。

3) 修改函数的代码，这样就可以正确地设置块索引的名字、职位、部门和薪水和它的扩展字典。

- 替换

```
BlockReference br = new BlockReference(new Point3d(10, 10, 0),
CreateEmployeeDefinition());
```

为

```
BlockReference br = new BlockReference(pos, CreateEmployeeDefinition());
```

- 替换

```
xRec.Data = new ResultBuffer(
new TypedValue((int)DxfCode.Text, "Earnest Shackleton"),
new TypedValue((int)DxfCode.Real, 72000),
new TypedValue((int)DxfCode.Text, "Sales"));
```

为

```
xRec.Data = new ResultBuffer(
new TypedValue((int)DxfCode.Text, name),
```

```
new TypedValue((int)DxfCode.Real, salary),
new TypedValue((int)DxfCode.Text, division));
```

4) 因为我们把雇员的名字从 MText 替换成块的属性定义，因此我们要创建一个相应的属性索引来显示雇员的名字。属性索引将使用属性定义的属性。

- 替换:

```
btr.AppendEntity(br); //加入索引到模型空间
trans.AddNewlyCreatedDBObject(br, true); //让事务处理知道
```

为

```
AttributeReference attRef = new AttributeReference();
//遍历雇员块来查找属性定义
    BlockTableRecord empBtr =
(BlockTableRecord)trans.GetObject(bt["EmployeeBlock"], OpenMode.ForRead);
foreach (ObjectId id in empBtr)
{
Entity ent = (Entity)trans.GetObject(id, OpenMode.ForRead, false);
//打开当前的对象!
if (ent is AttributeDefinition)
{
//设置属性为属性索引中的属性定义
AttributeDefinition attDef = ((AttributeDefinition)(ent));
attRef.SetPropertiesFrom(attDef);
attRef.Position = new Point3d(attDef.Position.X + br.Position.X,
attDef.Position.Y + br.Position.Y, attDef.Position.Z + br.Position.Z);
attRef.Height = attDef.Height;
attRef.Rotation = attDef.Rotation;
attRef.Tag = attDef.Tag;
attRef.TextString = name;
}   }
//把索引加入模型空间
btr.AppendEntity(br);
//把属性索引加入到块索引
br.AttributeCollection.AppendAttribute(attRef);
//让事务处理知道
trans.AddNewlyCreatedDBObject(attRef, true);
trans.AddNewlyCreatedDBObject(br, true);
```

研究一下上面的代码，看看是怎样把属性定义中除显示用的文本字符串外的属性复制到属性索引的。属性被加入到块索引的属性集合中。这就是你怎样来为每一个实例自定义雇员名字。

5)不要忘记返回雇员块索引的 ObjectId，但要在提交事务处理之后才能返回：

```
trans.Commit();
return br.ObjectId;
```

6) 测试 CreateEmployee。

加入一个 Test 命令来测试 CreateEmployee：

```
[CommandMethod("Test")]
```

```
public void Test()
{
CreateEmployee("Earnest Shackleton", "Sales", 10000, new Point3d(10, 10, 0));

}
```

修改 CreateDivision()以重用：

让我们来修改 CreateDivision ()函数，以让它可以接收部门名字、经理名字并返回创建的部门经理扩展记录的 ObjectId。如果部门经理已经存在，则不改变经理的名字。

1) 如果你先前在 CreateEmployeeDefinition()中调用了 CreateDivision(),请把它注释掉，因为我们在这里不需要创建一个部门

2) 改变 CreateDivision()的形式让它接收部门和经理的名字并返回一个 ObjectId。

```
public ObjectId CreateDivision(string division, string manager)
```

3) 修改上面函数的代码创建部门的名字和经理：

- 替换:

```
divDict = (DBDictionary)trans.GetObject(acmeDict.GetAt("Sales"),
OpenMode.ForWrite);
```

为:

```
divDict = (DBDictionary)trans.GetObject(acmeDict.GetAt(division),
OpenMode.ForWrite);
```

- 替换:

```
acmeDict.SetAt("Sales", divDict);
```

为:

```
acmeDict.SetAt(division, divDict);
```

- 替换:

```
mgrXRec.Data = new ResultBuffer(new TypedValue((int)DxfCode.Text,
"Randolph P. Brokwell"));
```

为:

```
mgrXRec.Data = new ResultBuffer(new TypedValue((int)DxfCode.Text,
manager));
```

不要忘了返回部门经理这个扩展记录的 ObjectId，但要在提交事务处理后才返回。

```
trans.Commit();
//返回部门经理这个扩展记录的 ObjectId
return mgrXRec.ObjectId;
```

现在把在中 CreateEmployeeDefinition 调用的 CreateDivision 函数给注释掉。

4) 现在通过使用 TEST 命令来测试调用 CreateDivision 函数。使用 ArxDbg 工具来检查条目是否已被加入到“ACME_DIVISION”下的命名对象字典。

```
CreateDivision("Sales", "Randolph P. Brokwell")
```

使用 CREATE 命令来创建雇员：

我们将加入一个名为 CREATE 的新命令,此命令用来提示输入雇员的详细资料来创建雇员块索引。让我们来看一下这个命令是怎样使用的。

1) 让我们加入一个名为 CREATE 的新命令，并声明几个常用的变量和一个 try-finally 块。

```
[CommandMethod("CREATE")]
public void CreateEmployee()
```

```
{
 Database db = HostApplicationServices.WorkingDatabase;
 Editor ed = Application.DocumentManager.MdiActiveDocument.Editor;
 Transaction trans = db.TransactionManager.StartTransaction();
 try
 {    trans.Commit();    }
 finally
 {
 trans.Dispose();
 }   }
```

2) 让我们来为雇员定义可以用作为提示缺省值的常数。注意，布尔值 gotPosition 是用来判断用户是否已输入职位。

. 雇员名 - 类型 ：String -缺省值 “Earnest Shackleton”
. 雇员所在部门名　　- 类型：String　　-缺省值“Sales”
. 薪水 -类型：Double (non-negative and not zero) -缺省值 10000
. 职位 -类型：Point3d -缺省值(0,0,0)

把这些常数加入到 try 语句后面：

```
string empName = "Earnest Shackleton";
string divName = "Sales";
double salary = new double();
salary = 10000;
Point3d position = new Point3d(0, 0, 0);
bool gotPosition = new bool();
//布尔值用来判断用户是否已输入职位
gotPosition = false;
```

3) 现在让我们提示用户输入值。我们先使用 PromptXXXOptions 类来初始化要显示的提示字符串。

```
//提示输入每个雇员的详细资料
PromptStringOptions prName = new PromptStringOptions("Enter Employee Name <"
+ empName + ">");
PromptStringOptions prDiv = new PromptStringOptions("Enter Employee Division
<" + divName + ">");
PromptDoubleOptions prSal = new PromptDoubleOptions("Enter Employee Salary <"
+ salary + ">");
PromptPointOptions prPos = new PromptPointOptions("Enter Employee Position
or");
```

注意,提示字符串用尖括号来显示变量的值。这是 AutoCAD 用来提示用户这个值为缺省值 。

4) 当提示用户输入职位时，我们也提供了一个关键字列表选项，如名字、部门和薪水。如果用户想要在选择一个点的时候改变为其它值，他可以选择那个关键字。

一个命令提示的例子如下：

```
Command: CREATE
```

```
Enter Employee Position or [Name/Division/Salary]:
```

要创建一个雇员，用户会选择一个点而其它的值被设置为缺省值。如果用户要改变其它的值，如名字，他可以输入”N”或全名”Name”，然后输入名字：

```
Command: CREATE
Enter Employee Position or [Name/Division/Salary]:N
Enter Employee Name <Earnest Shackleton>:
```

如果用户想要再次选择缺省的名字，他可以按回车键。

让我们创建用于职位提示的关键字列表：

```
//加入用于职位提示的关键字
prPos.Keywords.Add("Name");
prPos.Keywords.Add("Division");
prPos.Keywords.Add("Salary");
//设置提示的限制条件
prPos.AllowNone = false; //不允许没有值
```

5) 现在让我们声明 PromptXXXResult 变量来获取提示的结果：

```
//prompt results
PromptResult prNameRes;
PromptResult prDivRes;
PromptDoubleResult prSalRes;
PromptPointResult prPosRes;
```

6) 直到用户成功输入一个点后，循环才结束。如果输入错误的话，我们会提示用户并退出函数：

判断用户是否输入了关键字，我们通过检查 promptresult 的状态来进行：

```
//循环用来获取雇员的详细资料。当职位被输入后，循环终止。
while (!gotPosition)
{    //提示输入职位
 prPosRes = ed.GetPoint(prPos);
 //取得一个点
 if (prPosRes.Status == PromptStatus.OK)
 {    gotPosition = true;
 position = prPosRes.Value;
 }
 else if (prPosRes.Status == PromptStatus.Keyword) //获取一个关键字
 {
 //输入了 Name 关键字
 if (prPosRes.StringResult == "Name")
 {
 //获取雇员名字
 prName.AllowSpaces = true;
 prNameRes = ed.GetString(prName);
 if (prNameRes.Status != PromptStatus.OK)
```

```
{   return;    }
//如果获取雇员名字成功
if (prNameRes.StringResult != "")
{
empName = prNameRes.StringResult;
}
}
}
else
{
// 获取职位时发生错误
ed.WriteMessage("***Error in getting a point, exiting!!***" + "\r\n");
return;
} // 如果获取一个点
  }
```

7) 上面的代码只提示输入名字，请加入提示输入薪水和部门的代码。

8) 完成提示输入后，我们将使用获得的值来创建雇员。

```
//创建雇员
CreateEmployee(empName, divName, salary, position);
 //www.knowsky.com
```

9) 现在来检查部门经理是否已存在。我们通过检查 NOD 中部门的扩展记录中的经理名字来进行。如果检查到的是一个空字符串，那么我们会提示用户输入经理的名字。注意，通过修改 CreateDivision()函数，获取经理的名字变得简单了。

```
string manager = "";
//创建部门
//给经理传入一个空字符串来检查它是否已存在
Xrecord depMgrXRec;
ObjectId xRecId;
xRecId = CreateDivision(divName, manager);
//打开部门经理扩展记录
depMgrXRec = (Xrecord)trans.GetObject(xRecId, OpenMode.ForRead);
TypedValue[] typedVal = depMgrXRec.Data.AsArray();
foreach (TypedValue val in typedVal)
{
 string str;
 str = (string)val.Value;
 if (str == "")
 {
 //经理没有被设置，现在设置它
 // 先提示输入经理的名字
 ed.WriteMessage("\r\n");
 PromptStringOptions prManagerName = new PromptStringOptions("No manager set
for the division! Enter Manager Name");
 prManagerName.AllowSpaces = true;
```

```
PromptResult prManagerNameRes = ed.GetString(prManagerName);
if (prManagerNameRes.Status != PromptStatus.OK)
{
return;
}
//设置经理的名字
depMgrXRec.Data = new ResultBuffer(new TypedValue((int)DxfCode.Text,
prManagerNameRes.StringResult));
}
}
```

10) 测试 CREATE 命令

选择集：

现在让我们来创建一个命令，当用户在图形中选择一个雇员对象时，它会显示雇员的详细资料。

我们会使用上一章中创建的 ListEmployee()函数在命令行中输出雇员的详细资料。

下面是你必须遵循的步骤：

调用“LISTEMPLOYEES”命令

调用 Editor 的 GetSelection()函数来选择实体

```
PromptSelectionResult res = ed.GetSelection(Opts, filter);
```

上面的 filter 用来过滤选择集中的块索引。你可以创建如下的过滤列表：

```
TypedValue[] filList = new TypedValue[1];
filList[0] = new TypedValue((int)DxfCode.Start, "INSERT");
SelectionFilter filter = new SelectionFilter(filList);
```

从选择集中获取 ObjectId 数组：

```
//如果选择失败则什么也不做
if (res.Status != PromptStatus.OK)
return;
Autodesk.AutoCAD.EditorInput.SelectionSet SS = res.Value;
ObjectId[] idArray;
idArray = SS.GetObjectIds();
```

5. 最后，把选择集中的每个 ObjectId 输入到 ListEmployee()函数来获取一个雇员详细资料的字符串数组。把雇员的详细资料输出到命令行。例如：

```
//获取 saEmployeeList 数组中的所有雇员
foreach (ObjectId employeeId in idArray)
{
ListEmployee(employeeId, ref saEmployeeList);
//把雇员的详细资料输出到命令行
foreach (string employeeDetail in saEmployeeList)
{
ed.WriteMessage(employeeDetail);
}
ed.WriteMessage("----------------------" + "\r\n");
}
```

第 6 章　　更多的用户界面：添加自定义数据

在本章中，我们将介绍.NET API 的用户界面部分能做些什么。我们首先将介绍一个自定义上下文菜单（快捷菜单）。接下来我们将实现一个无模式可停靠的面板（一个真正的 AutoCAD 增强辅助窗口）来支持拖放操作。接着我们将介绍通过模式窗体选取实体。最后，我们将介绍使用 AutoCAD 的选项对话框来设置雇员的缺省值。
本章还会介绍和上面内容有关的 API。

第一部分　自定义上下文菜单

到目前为止，我们所写的代码只与 CommandMethod 属性定义的命令行进行相互操作。一个 AutoCAD .NET 程序能通过一个特殊的类来实现装载时的初始化工作。这个类只要实现 IExtensionApplication .NET 接口并暴露一个组件级别的属性（此属性把类定义为 ExtensionApplication），就可以响应一次性的装载和卸载事件。例子：

```
[assembly: ExtensionApplication(typeof(Lab6_CS.AsdkClass1))]
  public class AsdkClass1 : IExtensionApplication
                                          {
```

1)　现在修改 AsdkClass1 类来实现上面的接口。要实现这个接口，你必须实现 Initialize()和 Terminate()函数。因为我们要实现的是一个接口，而接口中的函数总是定义为纯虚拟的 。

```
    public void Initialize()
    {
       AddContextMenu();
       EmployeeOptions.AddTabDialog();

    }
    public void Terminate()
    {
    }
```

为了加入自定义上下文菜单，我们必须定义一个‘ContextMenuExtension’类的成员。这个类位于 Autodesk.AutoCAD.Windows 命名空间中。

要使用 ContextMenuExtension，我们必须使用 new 关键字来进行初始化，给必要的属性赋值，并调用 Application.AddDefaultContextMenuExtension()。上下文菜单的工作方式是:对于每个菜单条目，我们定义一个成员函数来处理菜单单击事件。我们可能通过.NET 的代理来实现。我们使用 C#关键字+=和-=确定让哪个函数来处理事件。请尽快熟悉这种设计模式，因为在 C#中会使用很多。

2)　添加一个‘ContextMenuExtension’成员变量和下面两个用来添加和移除自定义菜单的函数。请好好研究一下代码来看看究竟发生了什么。

```
    void AddContextMenu()
    {
     try
       {  m_ContextMenu = new ContextMenuExtension();
          m_ContextMenu.Title = "Acme Employee Menu";
          Autodesk.AutoCAD.Windows.MenuItem mi;
```

```
            mi = new Autodesk.AutoCAD.Windows.MenuItem("Create Employee");
            mi.Click +=  new EventHandler(CallbackOnClick);
            m_ContextMenu.MenuItems.Add(mi);
            Autodesk.AutoCAD.ApplicationServices.Application.AddDefaultContextM
enuExtension(m_ContextMenu);
         }
         catch
         {         }
      }

      void RemoveContextMenu()
      {
       try
         { if( m_ContextMenu != null )
            { Autodesk.AutoCAD.ApplicationServices.Application.RemoveDefaultCon
textMenuExtension(m_ContextMenu);
              m_ContextMenu = null;
            }
         }
         catch
         {         }
      }
```

注意我们在代码中使用了‘CallbackOnClick’函数。我们希望这个函数（我们现在还没有定义）响应菜单项选择事件。在我们的例子中，我们想要做的是调用我们的成员函数‘Create()’。请加入下面的代码。

```
      void CallbackOnClick(object Sender, EventArgs e)
      {   Create();      }
```

现在，从 Initialize()中调用 AddContextMenu()函数，同样地，请在 Terminate()中调用 RemoveContextMenu()。

请运行代码。使用 NETLOAD 来装载编译好的组件，然后在 AutoCAD 的空白处右击……你应该可以看到’Acme‘快捷菜单了。如果失败了，明明你做的都是正确的……为什么呢？

通常，AutoCAD 的数据是存储在文档中的，而访问实体的命令有权修改文档。当我们运行代码来响应上下文菜单单击事件，我们是从命令结构的外部来访问文档。当我们调用的代码尝试通过添加一个雇员来修改文档时，我们就碰到了错误。正确的做法是必须锁住文档，这可以通过使用 Document.LockDocument()命令来实现。

3) 修改 CallbackOnClick 来锁住文档：

```
      void CallbackOnClick(object Sender, EventArgs e)
      {   DocumentLock docLock =
Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDocum
ent.LockDocument();
         Create();
         docLock.Dispose();
      }
```

注意，我们保留了一个‘DocumentLock’对象的拷贝。要把文档解锁，我们只要销毁这个 DocumentLock 对象。

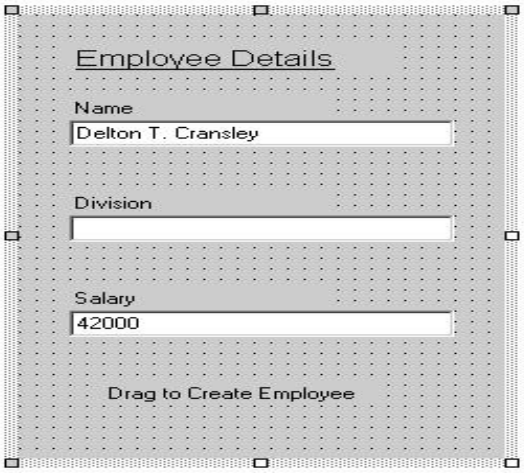
再次运行代码。现在应该可以看到快捷菜单了。

第 2 部分　无模式对话框、可进行拖放的可停靠面板

为了使我们的用户界面和 AutoCAD 实现无缝链接，我们要尽可能在所有的地方使用同样的用户界面结构。这会使应用程序看起来与 AutoCAD 结合的很好，并有效地减少代码的重复。一个很好的例子是 AutoCAD 中的可停靠面板。

使用.NET API，我们可以创建一个简单的窗体并把它放到面板中。我们可以实例化一个自定义的‘PaletteSet’对象来包含窗体，并可以把这个 PaletteSet 定义成我们喜欢的样式。

4)　在解决方案浏览器中通过右击工程来添加一个用户控件。给它命名为 ModelessForm。使用控件工具箱，加入如下所示的编辑框和标签控件。



使用属性窗口设置三个编辑框的属性。设置如下：

〈 首先是最上面的编辑框〉

(Name)　　= tb_Name

Text =〈请输入一个名字〉

〈第二个编辑框〉

(Name) = tb_Division

Text = Sales

〈第三个编辑框〉

(Name) = tb_Salary

Text =〈请输入薪水〉

要使用.NET API 实例化一个面板对象，你必须要实例化用户控件对象（无模式窗体）和‘PaletteSet’对象。调用 PaletteSet 的成员函数 Add 来传递用户控件对象。

5)　接下来，我们要加入一个命令来创建面板。在类中加入一个名为 CreatePalette 的函数和 CommandMethod 属性来定义名为“PALETTE”的命令。

请看一下下面的代码块。这是实例化面板的代码。

```
ps = new Autodesk.AutoCAD.Windows.PaletteSet("Test Palette Set");
ps.MinimumSize = new System.Drawing.Size(300, 300);
System.Windows.Forms.UserControl myCtrl = new ModelessForm();
ps.Add("test", myCtrl);
```

```
ps.Visible = true;
```

6) 把上面的代码加入到 CreatePalette()函数。‘ps’需要在函数的外部声明：

```
private Autodesk.AutoCAD.Windows.PaletteSet ps;
```

在函数的实例化面板代码之前加入检查 ps 是否为 null 的代码。

编译并运行工程。在 AutoCAD 中装载组件，运行‘PALETTE’命令来检查面板是否被装载。

使用 PaletteSet.Style 来看看 PaletteSetStyles 对象。例如：

```
ps.Style = PaletteSetStyles.ShowTabForSingle;
```

我们也可以试试诸如透明性的属性，例如：

```
ps.Opacity = 90;
```

注意：要使用 PaletteSet 和 PaletteSetStyles 对象，你必须加入两个命名空间 Autodesk.AutoCAD.Windows 和 Autodesk.AutoCAD.Windows.Palette

在我们继续之前，让我们执行一个快速的维护更新：请在 AsdkClass1 类中加入下列成员：

```
public static string sDivisionDefault = "Sales";
public static string sDivisionManager = "Fiona Q. Farnsby";
```

这些值将被用作为部门和部门经理的缺省值。由于它们被声明为’static’，它们在每个程序中只实例化一次，并在组件装载的时候实例化。

第 2a 部分　在无模式窗体中加入拖放支持

在这部分，我们将加入允许我们使用面板窗体中编辑框的值来创建一个雇员。当用户从面板中拖动到 AutoCAD 中，将会提示输入职位，一个新的雇员实体将使用这些值来进行创建。

7) 为了支持拖放，我们首先需要一个对象来进行拖动。在编辑框的下面，另外加入一个名为 Label4 的标签控件，设置标签的文本为一些提示性的东西（‘Drag to Create Employee’）。通过这个标签，我们可以在 AutoCAD 中处理拖放。

要捕捉到什么时候拖动事件发生，我们必须要知道什么时候鼠标开始操作。

首先，我们要在类的构造函数中注册事件，代码如下：

```
Label4.MouseMove += new
System.Windows.Forms.MouseEventHandler(Label4_MouseMove);
```

8) 在 ModelessForm 类中加入下面的函数声明：

```
private void Label4_MouseMove( object sender,
System.Windows.Forms.MouseEventArgs e)
    {
        if (System.Windows.Forms.Control.MouseButtons ==
System.Windows.Forms.MouseButtons.Left)
        {
            // start dragDrop operation, MyDropTarget will be called when the
cursor enters the AutoCAD view area.
            Autodesk.AutoCAD.ApplicationServices.Application.DoDragDrop(this,
this, System.Windows.Forms.DragDropEffects.All, new MyDropTarget());
        }
    }
```

通常事件处理器有 2 个输入参数，一个 object 类的 sender 和与事件有关的参数。对于 MouseMove，我们也要做同样的事情。

运行这个工程，检查一下当鼠标经过文本的时候，函数是否被调用的。

我们还可以进一步知道是不是按了鼠标左键：

```
        if (System.Windows.Forms.Control.MouseButtons ==
System.Windows.Forms.MouseButtons.Left)
        {        }
```

我们需要一个方法来检测什么时候对象被拖入到 AutoCAD。我们可以使用.NET 的基类 DropTarget 来实现。要使用它，你只要创建从这个基类派生的类并实现你想要的函数。在我们这个例子中，我们需要的是 OnDrop()。

9) 在工程中加入一个从 Autodesk.AutoCAD.Windows.DropTarget 派生的类‘MyDropTarget’。如果你把这个类加入到 ModelessForm.cs 文件中，请把这个类加入到 ModelessForm 类之后。

```
 override public void OnDrop(System.Windows.Forms.DragEventArgs e)
{}
```

在这个函数中，我们最后会调用 AsdkClass1 的成员 CreateDivision() 和 CreateEmployee，传入 ModelessForm 类中的编辑框的值。要实现这个功能，我们需要一个方法来连接 ModelessForm 实例。最佳的方法是通过 DragEventArgs。但首先我们要把鼠标事件连接到 MyDropTarget 类。

10) 加入下面的代码到鼠标左键（MouseButtons.Left）处理函数中：

```
 Autodesk.AutoCAD.ApplicationServices.Application.DoDragDrop(this,   this,
System.Windows.Forms.DragDropEffects.All, new MyDropTarget());
```

注意我们传入’this’两次。第一次是用于 Control 参数，第二次是用于传入用户自定义数据。因为我们传入的是 ModelessForm 类的实例，所以我们可以在放下的时候使用它来获取编辑框的值。

11) 回到 OnDrop 处理函数，让我们使用参数来调用创建雇员的函数。首先，添加职位提示的代码。在 AsdkClass1.Create()中已经有相关的代码了，位于‘Get Employees Coordinates…’.注释下面。添加此代码来提示输入职位。

12) 接下来，获取传入到 DragEventArgs 参数的 ModelessForm 对象：

```
        ModelessForm ctrl = (ModelessForm)e.Data.GetData(typeof(ModelessForm));
```

请注意一下怎样通过 typeof 关键字把参数强制转化为 ModelessForm 的实例。

13) 使用上面的实例来调用 AsdkClass1 成员：

```
AsdkClass1.CreateDivision(ctrl.tb_Division.Text,
AsdkClass1.sDivisionManager);
AsdkClass1.CreateEmployee(ctrl.tb_Name.Text, ctrl.tb_Division.Text,
Convert.ToDouble(ctrl.tb_Salary.Text), prPosRes.Value);
```

注意：AsdkClass1 的方法要不通过 AsdkClass1 的实例来调用，那么方法必须被声明为’public static’。因为 public static 方法只能调用其它的 public static 方法，你需要修改几个 AsdkClass1 类中的方法为’ public static’。请你进行相关的修改（应该至少有 4 项要修改）。

14) 最后，因为我们处理的事件位于 AutoCAD 命令之外，我们必须再次在会修改数据库的代码处锁住文档。请加入锁住文档的代码，加入的方法与前面的上下文菜单是一样的。

编译、装载并运行组件，使用 PALETTE 命令，你应该可以使用拖放操作来创建一个雇员了。

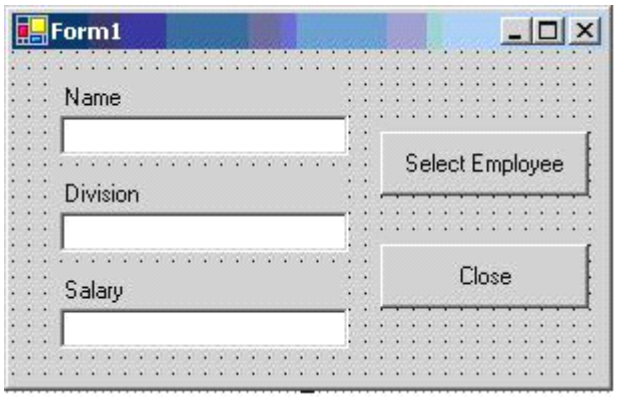
第三部分　　从有模式窗体中选择实体

本章的以下部分将演示获取一个用户在屏幕上选择的雇员实例的详细信息，并把信息显示在一个有模式窗体的编辑框中。这部分的重点是创建一个有模式窗体，并在执行选择操作

而窗体要失去焦点时隐藏它。为了获取雇员的详细信息，我们将使用第 4 章结束时给出的 ListEmployee 帮助函数。

首先，我们需要创建一个窗体类。这个类是一个真实的窗体而不是我们在 ModelessForm 中创建的用户控件。

15) 在工程中创建一个 Windows 窗体类。调用‘ModalForm’类。在窗体中加入以下所示的三个编辑框控件和标签控件以及两个按钮。



使用属性窗口来设置三个编辑框的属性。设置如下：

〈首先是最上面的编辑框〉

(Name) = tb_Name

Text = 〈空白〉

〈第二个编辑框〉

(Name) = tb_Division

Text = 〈空白〉

〈第三个编辑框〉

(Name) = tb_Salary

Text = 〈空白〉

〈上部的按钮〉

(Name) = SelectEmployeeButton

Text = Select Employee

〈下部的按钮〉

(Name) = Close

Text = Close

接下来创建按钮的事件处理函数。‘Close’按钮可以只简单地调用：

```
this.Close();
```

要显示对话框，让我们在类中创建一个把窗体实例化为有模式对话框的命令函数。下面的实现的代码：

```
[CommandMethod("MODALFORM")]
public void ShowModalForm()
{   ModalForm modalForm = new ModalForm();
Autodesk.AutoCAD.ApplicationServices.Application.ShowModalDialog(modalForm);
}
```

编译、装载并在 AutoCAD 中运行 MODALFORM 命令来看看对话框是否可以工作。试试在对话框的右下角调整对话框的大小，然后关闭它。注意，重新使用 MODALFORM 命令时，对话框

会出现在你上次离开的地方！这是 ShowModalDialog 方法的一个特征。大小和位置值被 AutoCAD 保存了。

‘Select Employee’按钮首先将执行一个简单的实体选择。这我们可以通过使用 Editor.GetEntity()方法来实现，选择单一的实体比使用选择集来得方便的多。下面是怎样使用这个方法的代码：

```
PromptEntityOptions prEnt = new PromptEntityOptions("Select an Employee");
PromptEntityResult prEntRes = ed.GetEntity(prEnt);
```

16)把上面的代码加入到 SelectEmployeeButton_Click 处理函数中，还要加入必需的数据库、命令行、事务处理设置变量和一个 try catch 块。不要忘了在 finally 块中销毁它们。

使用 PromptStatus.OK 来测试 GetEntity 的返回值，如果返回不等于，就调用 this.Show 并退出处理函数。

一旦我们获得的返回值是 OK,那么我们就可以使用 PromptEntityResult.ObjectId()方法来获取所选实体的 object Id。这个 id 可以和一个固定的字符串数组被传入到 AsdkClass1.ListEmployee 函数中来获取雇员的详细信息。可以通过以下的代码说明：

```
ArrayList saEmployeeList = new ArrayList();
AsdkClass1.ListEmployee(prEntRes.ObjectId, saEmployeeList);
if (saEmployeeList.Count == 4)
 {   tb_Name.Text = saEmployeeList[0].ToString();
     tb_Salary.Text = saEmployeeList[1].ToString();
     tb_Division.Text = saEmployeeList[2].ToString();
 }
```

17) 加入上面的代码，它会在窗体的编辑框中显示雇员的详细信息。

在开始测试代码之前，我们还要记住的是代码是在有模式对话框中运行的，也就意味着当对话框可见的时候用户与 AutoCAD 的互操作是被禁止的。在用户能够进行选择雇员对象之前，我们必须隐藏窗体。当选择结束后，我们可以再次站窗体显示（例如，可以在 finally 块的函数中）

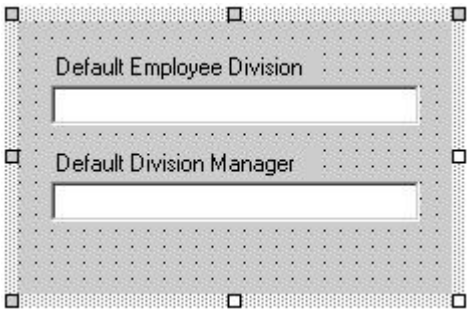
18) 在选择之前加入隐藏窗体的代码（例如在 try 块之前） ‘this.Hide’ 和选择结束后显示窗体的代码（例如，可以在 finally 块中）‘this.Show’。

编译、装载并在 AutoCAD 中运行 MODALFORM 命令来看看对话框是否工作。试试选择一个实体并填充窗体中编辑框的值。

第四部分 在 AutoCAD 选项对话框中加入页面

本章的最后部分将向你介绍如何定义一个用户控件，这个控件可以被作为一个页面显示在 AutoCAD 的选项对话框中。我们可以使用这个页面来设置程序运行期间的缺省值。在 Employee 例子中，我们只是在 AsdkClass1 类中简单地设置了 sDivisionDefault 和 sDivisionManager 字符串。

19) 在工程中加入另外一个名为‘EmployeeOptions’的用户控件。在控件中加入两个编辑框和标签控件，如下图所示：



使用属性窗口来设置编辑框的属性，设置如下：

〈上面的编辑框〉

(Name) = tb_EmployeeDivision

Text = 〈空白〉

〈下面的编辑框〉

(Name) = tb_DivisionManager

Text = 〈空白〉

使用.NET API 来显示自定义多页对话框，需要两个步骤。首先，通过传入要调用的成员函数的地址，来知道什么时候选项对话框出现。其次是实现回调函数。传入到回调函数中的第二个参数是一个‘TabbedDialogEventArgs’对象，我们必须使用它来调用‘AddTab’函数。AddTab 使用一个标题字符串和一个‘TabbedDialogExtension’对象的实例，此实例封装了我们的窗体（其实是用户控件）。在 TabbedDialogExtension 的构造函数中，我们输入窗体的实例和回调函数（OnOK, OnCancel 或 OnHelp）的地址。

20) 在 EmployeeOptions 类中，加入一个名为 AddTabDialog 的 public static 函数，它会添加一个可供系统调用的事件处理：

```
public static void AddTabDialog()
{Autodesk.AutoCAD.ApplicationServices.Application.DisplayingOptionDialog    +=
new TabbedDialogEventHandler(TabHandler);
 }
```

在 AsdkClass1 的 Initialize 函数中加入调用此函数的代码。因为这个函数是在程序启动的时候调用的（因为类已经实现了 IExtensionApplication 接口），所以多页对话框就被自动的加载。

20a) 实现一个相同的函数来移除事件处理，使用 C#的-=关键字。

在这里，你可以看到我们为 AutoCAD 中的 Application 对象的 DisplayingOptionDialog 事件加入了一个处理函数，此函数会调用‘TabHandler’函数。所以接下来我们要实现这个函数。

21) 加入下面的代码来实现处理函数：

```
private static void TabHandler(object sender,
Autodesk.AutoCAD.ApplicationServices.TabbedDialogEventArgs e)
{
   EmployeeOptions EmployeeOptionsPage = new EmployeeOptions();
   e.AddTab("Acme Employee Options",
      new TabbedDialogExtension(
      EmployeeOptionsPage,
      new TabbedDialogAction(EmployeeOptionsPage.OnOk)));
}
```

我们首先实例化了一个 EmployeeOptions 对象。然后调用 e.AddTab()，在这个函数中传入了一个 TabbedDialogExtension 的实例。TabbedDialogExtension 的构造函数使用了 EmployeeOptionsPage 实例和一个 TabbedDialogAction 对象。TabbedDialogAction 对象的参数可以是 Ok, Cancel 或 Help 回调函数。在这个函数中，我们使用的是 OK。

22) 现在剩下的就是确定回调函数的内容，也就是 ONOK 的内容。前面已经说过了，我们只要设置 AsdkClass1 的 static 成员，也就是设置 tb_DivisionManager 和 tb_EmployeeDivision 编辑框中的值。下面是代码：

```
public void OnOk()
```

```
    {
        AsdkClass1.sDivisionDefault = tb_EmployeeDivision.Text;
        AsdkClass1.sDivisionManager = tb_DivisionManager.Text;
    }
```

编译、装载并选择 AutoCAD 的选项菜单项来看一下我们的自定义对话框。试试设置对话框中的值并实例化一个雇员。你可以使用 PRINTOUTEMPLOYEE 命令来查看详细信息。

附加的问题：怎样让对话框的编辑框能自动显示为 AsdkClass1 中的 Manager 和 Division 字符串的内容？

第 7 章　事件

本章将讨论 AutoCAD 中的事件。我们将介绍事件处理函数的使用，特别是监视 AutoCAD 命令的事件处理函数和监视被 AutoCAD 命令修改的对象的事件处理函数。在解释怎样在 C# 中实现 AutoCAD 的事件处理之前，我们将首先简要地讨论一下.NET 中的事件。

第一部分　　C#中的事件

事件只是用来通知一个行为已经发生的信息。在 ObjectARX 中，我们使用反应器(reactor)来处理 AutoCAD 的事件。而在 AutoCAD .NET API 中，ObjectARX 反应器被换成了事件。事件处理函数（或者叫回调函数）是用来监视和反馈程序中出现的事件。事件可以以不同的形式出现。

在介绍 AutoCAD .NET API 中的事件之前，让我们先来简单地了解一下代理。

第 1a 部分　　代理

代理是一个存储方法索引的类（概念与函数指针类似）。代理对方法是类型安全的（与 C 中的函数指针类似）。代理有特定的形式和返回类型。代理可以封装符合这种特定形式的任何方法。

代理的一个用途就是作为产生事件的类的分发器。事件是.NET 环境中第一级别的对象。虽然 C#把事件处理的许多细节给隐藏掉了，但事件总是由代理来实现的。事件代理可以多次调用（就是它们可以存储多于 1 个的事件处理方法的索引）。它们保存了用于事件的一个注册事件处理的列表。一个典型的代理有以下的形式：

```
public delegate Event (Object sender, EventArgs e)
```

第一个参数 sender 表示引发事件的对象。第二个参数 e 是一个 EventArgs 参数（或者是一个派生的类），这个对象通常包含用于事件处理函数的数据。

第 1 b 部分　+=和-=语句

要使用事件处理函数，我们必须把它与事件联系起来。这要通过使用+=语句。+=和-=允许你在运行时连接、断开或修改与事件联系的处理函数。

当我们使用+=语句时，我们要确定事件引发者的名字，并要使用 new 语句来确定事件处理函数，例如：

```
MyClass1.AnEvent += new  HandlerDelegate(EHandler)
```

前面我们说过要使用-=语句从事件处理函数中断开事件（移除联系）。语法如下所示：

```
MyClass1.AnEvent -= new  HandlerDelegate(EHandler)
```

第 2 部分　　处理.NET 中的 AutoCAD 事件

在 ObjectARX 中，我们使用反应器来封装 AutoCAD 事件。在 AutoCAD .NET API 中，我们可以使用事件来代替 ObjectARX 反应器。

通常，处理 AutoCAD 事件的步骤如下：

1. 创建事件处理函数

当一个事件发生时，事件处理函数（或称为回调函数）被调用。任何我们想要处理的回应 AutoCAD 事件的动作都在事件处理函数中进行。

例如，假定我们只想通知用户一个 AutoCAD 对象已被加入。我们可以使用 AutoCAD 数据库事件”ObjectAppended”来完成。我们可以编写回调函数（事件处理函数）如下：

```
public void objAppended(object o, ObjectEventArgs e)
{
   // 在这里加入处理代码
}
```

函数中的第一个参数代表 AutoCAD 数据库。第二个参数代表 ObjectEventArgs 类，它可能包含对处理函数有用的数据。

2.把事件处理函数与事件联系起来

为了开始监视动作，我们必须把事件处理函数与事件联系起来。在这里，当一个对象加入到数据库时，ObjectAppended 事件将会发生。但是，事件处理函数不会响应这个事件，除非我们把它与这个事件联系起来，例如：

```
Database db;
db = HostApplicationServices.WorkingDatabase;
db. ObjectAppended += new ObjectEventHandler(objAppended);
```

3. 断开事件处理函数

要终止监视一个动作，我们必须断开事件处理函数与事件的联系。当对象被加入时，我们想要停止通知用户这个事件，我们要断开事件处理函数与事件 ObjectAppended 的联系。

```
db. ObjectAppended -= new ObjectEventHandler(objAppended);
```

第 3 部分　　使用事件处理函数来控制 AutoCAD 的行为

本章的目的是解释 AutoCAD 事件怎样才能被用于控制 AutoCAD 图形中的行为。现在，让我们使用前一章（第六章）的内容在 AutoCAD 图形中创建几个 EMPLOYEE 块索引。我们不想让用户能改变 EMPLOYEE 块索引的位置，而对于其它的非 EMPLOYEE 块索引的位置则没有这个限制。我们将混合使用数据库与文档事件来做到这一点。

首先，我们想要监视将要被执行的 AutoCAD 命令（使用 CommandWillStart 事件）。特别地，我们要监视 MOVE 命令。另外，当一个对象要被修改时，我们应该被通知（使用 ObjectOpenedForModify 事件），这样我们可以确定它是否为一个 EMPLOYEE 块索引。如果这时就修改对象可能是无效的，因为我们的修改可能会再次触发事件，从而引起不稳定的行为。所以，我们要等待 Move 命令的执行结束（使用 CommandEnded 事件），这时就可以安全地修改对象了。当然，任何对块索引的修改将会触发 ObjectOpenedForModify 事件。我们还需要设置一些全局变量来表明一个 MOVE 命令在运行和被修改的对象是一个 EMPLOYEE 块索引。

注意：因为本章需要比较多的代码来获得想要的结果，所以我们不会解释任何与事件处理无关的代码，而只是将它们粘贴到事件处理函数中。这里的重点是成功创建和注册事件处理函数。

第一步：创建新工程

我们以第六章的工程开始。请新加入一个类 AsdkClass2。我们还要加入四个全局变量。前两个是 Boolean 型的：一个用来表示我们监视的命令是否是活动的，另外一个用来表示 ObjectOpenedForModify 事件处理函数是否该被忽略。

```
//全局变量
```

```
bool bEditCommand;
bool bDoRepositioning;
```

接下来，我们要声明一个全局变量来表示一个 ObjectIdCollection，它用来存储我们所选择的要修改的对象的 ObjectID。

```
ObjectIdCollection changedObjects = new ObjectIdCollection();
```

最后，我们要声明一个全局变量来表示一个 Point3dCollection，它用来包含我们所选对象的位置（三维点）。

```
Point3dCollection employeePositions = new Point3dCollection();
```

第 2 步：创建第一个文档事件处理函数（回调函数）

现在我们要创建一个事件处理函数。当 AutoCAD 命令开始执行的时候它会通知我们。我们要检查 GlobalCommandName 的值是否为 MOVE。

```
if ( e.GlobalCommandName == "MOVE" )
{
}
```

如果 MOVE 命令开始执行的话，我们要相应地设置 Boolean 变量 bEditCommand 的值，这样我们可以知道我们所监视的命令是活动的。同样地，我们应该把另外一个 Boolean 变量 bDoRepositioning 设置为 false 来忽略 ObjectOpenedForModify 事件处理函数。两个变量设置好以后，在命令活动期间，我们必须要获得所选块索引的信息。

我们还应该把两个集合对象的内容清空。我们只关心当前选择的对象。

第 3 步： 创建数据库事件处理函数（回调函数）

无论什么时候一个对象被打开并要被修改时，数据库事件处理函数会被调用。当然，如果这时我们监视的命令不是活动的，我们就应该跳过任何被这个回调函数调用的内容。

```
if ( bEditCommand == false )
{
   return;
}
```

同样地，如果我们监视的命令已经结束，而 ObjectOpenedForModify 事件被另一个回调函数再次触发的话，而这时有对象被修改时，我们要阻止所有由这个回调函数执行的动作。

```
if ( bDoRepositioning == true )
{
   return;
}
```

这个回调函数剩余部分的代码用来验证我们是否正在处理 EMPLOYEE 块索引。如果是的话，我们就获取它的 ObjectID 和位置（三维点）。下面的代码可以被粘贴到这个事件处理函数函数。

```
public void objOpenedForMod(object o, ObjectEventArgs e)
{
   if ( bEditCommand == false )
   {              return;          }
   if ( bDoRepositioning == true )
   {             return;          }
   ObjectId objId;
   objId = e.DBObject.ObjectId;
   Transaction  trans;
```

```
        Database  db;
        db = HostApplicationServices.WorkingDatabase;
        trans = db.TransactionManager.StartTransaction();
        using(Entity ent  = (Entity)trans.GetObject(objId, OpenMode.ForRead, f
alse))
        {
           if ( ent.GetType().FullName.Equals( "Autodesk.AutoCAD.DatabaseServi
ces.BlockReference" ) )
           { //We use .NET//s RTTI to establish type.
               BlockReference br   = (BlockReference)ent;
               //Test whether it is an employee block
               //open its extension dictionary
               if ( br.ExtensionDictionary.IsValid )
               {
                  using(DBDictionary brExtDict  = (DBDictionary)trans.GetObjec
t(br.ExtensionDictionary, OpenMode.ForRead))
                  {
                     if ( brExtDict.GetAt("EmployeeData").IsValid )
                     {     //successfully got "EmployeeData" so br is employee
 block ref
                        //Store the objectID and the position
                        changedObjects.Add(objId);
                        employeePositions.Add(br.Position);
                        //Get the attribute references,if any
                        AttributeCollection atts;
                        atts = br.AttributeCollection;
                        if ( atts.Count > 0 )
                        {
                           foreach(ObjectId attId in atts )
                           {
                              AttributeReference att;
                             using(att = (AttributeReference)trans.GetObject(a
ttId, OpenMode.ForRead, false))
                              {
                                 changedObjects.Add(attId);
                                 employeePositions.Add(att.Position);
                              }
                           }
                        }
                     }
                  }
               }
           }
        }
```

```
        trans.Commit();
    }
```

第 4 步： 创建第二个文档事件处理函数（回调函数）

当一个命令结束时，第三个事件处理函数被调用。同样地，我们要检查全局变量来验证这个将要结束的命令是我们监视的命令。如果是我们监视的，那么我们要重置这个变量：

```
        if ( bEditCommand == false )
        {             return;          }
        bEditCommand = false;
```

这个回调函数执行的动作将会再次触发 ObjectOpenedForModify 事件。我们必须确定在这个回调函数中跳过了所有与此事件有关的动作。

```
        //设置标志来跳过 OpenedForModify 处理函数
        bDoRepositioning = true;
```

这个回调函数的剩余代码用来把 EMPLOYEE 块索引和它的关联属性引用的当前（修改过的）位置与它们的初始位置作比较。如果位置改变了，我们在这个回调函数中把它们重置这初始的位置。下面的代码可以被粘贴到这个事件处理函数中。

```
    public void cmdEnded(object o  , CommandEventArgs e)
    {
        //Was our monitored command active?
        if ( bEditCommand == false )
        {
            return;
        }
        bEditCommand = false;
        //Set flag to bypass OpenedForModify handler
        bDoRepositioning = true;
        Database db   = HostApplicationServices.WorkingDatabase;
        Transaction trans ;
        BlockTable bt;
        Point3d oldpos;
        Point3d newpos;
        int i ;
        for ( i = 0; i< changedObjects.Count; i++)
        {
            trans = db.TransactionManager.StartTransaction();
            using( bt = (BlockTable)trans.GetObject(db.BlockTableId, OpenMode.F
orRead) )
            {
                using(Entity ent   = (Entity)trans.GetObject(changedObjects[i],
 OpenMode.ForWrite))
                {
                    if ( ent.GetType().FullName.Equals("Autodesk.AutoCAD.Databas
eServices.BlockReference") )
                    { //We use .NET//s RTTI to establish type.
                        BlockReference br = (BlockReference)ent;
```

```
                        newpos = br.Position;
                        oldpos = employeePositions[i];

                        //Reset blockref position
                        if ( !oldpos.Equals(newpos) )
                        {
                           using( trans.GetObject(br.ObjectId, OpenMode.ForWrite)
 )
                           {
                              br.Position = oldpos;
                           }
                        }
                     }
                     else if ( ent.GetType().FullName.Equals("Autodesk.AutoCAD.Da
tabaseServices.AttributeReference") )
                     {
                        AttributeReference att = (AttributeReference)ent;
                        newpos = att.Position;
                        oldpos = employeePositions[i];

                        //Reset attref position
                        if ( !oldpos.Equals(newpos) )
                        {
                           using( trans.GetObject(att.ObjectId, OpenMode.ForWrit
e))
                           {
                              att.Position = oldpos;
                           }
                        }
                     }
                  }
               }
               trans.Commit();
            }
         }
```

第 5 步　创建命令来注册/断开事件处理函数

创建一个 ADDEVENTS 命令，使用+=语句来把上面的 3 个事件处理函数连接到各自的事件。在这个命令中，我们还应该设置全局 Boolean 变量：

```
bEditCommand = false;
bDoRepositioning = false;
```

创建另外一个命令 REMOVEEVENTS，使用-=语句把事件处理函数与事件断开。

第 6 步：　测试工程

要测试这个工程，请使用 CREATE 命令创建一个或多个 EMPLOYEE 块索引。如果你要作比较的话，你也可以插入一些非 EMPLOYEE 的块索引。

在命令行中键入 ADDEVENTS 命令来执行它。
在命令行中输入 MOVE 命令，然后选择你想要的块索引。注意，当 MOVE 命令结束时，EMPLOYEE 块索引（包括属性）还留在原处。
执行 REMOVEEVENTS 命令，然后在试一下 MOVE 命令。注意，EMPLOYEE 块索引现在可以被移动了。

附加的问题：添加一个附加的回调函数，当用户改变 EMPLOYEE 块索引的”Name”属性时，这个回调函数被触发。