ГЛАВА 18

Динамические типы и исполняющая среда динамического языка

В версии .NET 4.0 язык C# получил новое ключевое слово — `dynamic`. Это ключевое слово позволяет включать поведение, подобное сценариям, в строго типизированный мир точек с запятой и фигурных скобок. Используя эту слабую типизацию, можно значительно упростить некоторые сложные задачи кодирования и получить возможность взаимодействия с множеством динамических языков (таких как IronRuby и IronPython), которые поддерживают .NET.

В этой главе будет описано ключевое слово `dynamic`, а также показано, как слабо типизированные вызовы отображаются на корректные объекты в памяти, благодаря DLR (Dynamic Language Runtime — исполняющая среда динамического языка). После рассмотрения служб, предоставляемых DLR, будут приведены примеры использования динамических типов для облегчения выполнения вызовов методов с поздним связыванием (через службы рефлексии) и для упрощения взаимодействия с унаследованными библиотеками COM.

На заметку! Не путайте ключевое слово C# `dynamic` с концепцией *динамической сборки* (см. главу 17). Хотя ключевое слово `dynamic` может использоваться при построении динамической сборки, все же это две совершенно независимые концепции.

Роль ключевого слова C# dynamic

В главе 3 вы узнали о ключевом слове `var`, которое позволяет объявлять локальную переменную таким образом, что ее действительный тип данных определяется начальным присваиванием (это называется *неявной типизацией*). Как только начальное присваивание выполнено, вы получаете строго типизированную переменную, и любая попытка присвоить ей несовместимое значение приведет к ошибке компиляции.

Чтобы приступить к исследованию ключевого слова C# `dynamic`, создадим консольное приложение по имени `DynamicKeyword`. После этого поместим в класс `Program` показанный ниже метод и удостоверимся, что финальный оператор кода действительно инициирует ошибку во время компиляции, если убрать с него символы комментария.

```
static void ImplicitlyTypedVariable()
{
  // a имеет тип List<int>.
  var a = new List<int>();
  a.Add(90);

  // Это вызовет ошибку во время компиляции!
  // a = "Hello";
}
```

Использование неявной типизации просто потому, что она возможна, считается плохим стилем (если известно, что нужен тип List<int>, то его и следует указывать). Однако, как было показано в главе 13, неявная типизация очень полезна в сочетании с LINQ, поскольку многие запросы LINQ возвращают перечисления анонимных классов (через проекции), которые объявить явно в коде C# не получится. Тем не менее, даже в этих случаях неявно типизированная переменная на самом деле является строго типизированной.

Как уже известно из главы 6, System.Object находится на вершине иерархии классов в .NET Framework и может представлять все, что угодно. После объявления переменной типа object получается строго типизированный элемент данных, однако то, на что он указывает в памяти, может отличаться в зависимости от присваивания ссылки. Для того чтобы получить доступ к членам объекта, на который установлена ссылка в памяти, необходимо выполнять явное приведение.

Предположим, что есть простой класс по имени Person, в котором определены два автоматических свойства (FirstName и LastName), инкапсулирующие string. Теперь взгляните на следующий код:

```
static void UseObjectVarible()
{
  // Предположим, что есть класс по имени Person.
  object o = new Person() { FirstName = "Mike", LastName = "Larson" };

  // Для получения доступа к свойствам Person необходимо приводить object к Person.
  Console.WriteLine("Person's first name is {0}", ((Person)o).FirstName);
}
```

В версии .NET 4.0 язык C# стал поддерживать ключевое слово dynamic. На самом высоком уровне dynamic можно рассматривать как специализированную форму System.Object, в том смысле, что типу данных dynamic может быть присвоено любое значение. На первый взгляд, это порождает ужасную путаницу, поскольку теперь получается, что доступны три способа определения данных, внутренний тип которых явно не указан в коде. Например, следующий метод:

```
static void PrintThreeStrings()
{
  var s1 = "Greetings";
  object s2 = "From";
  dynamic s3 = "Minneapolis";
  Console.WriteLine("s1 is of type: {0}", s1.GetType());
  Console.WriteLine("s2 is of type: {0}", s2.GetType());
  Console.WriteLine("s3 is of type: {0}", s3.GetType());
}
```

будучи вызванным в Main(), выведет на консоль следующее:

```
s1 is of type: System.String
s2 is of type: System.String
s3 is of type: System.String
```

Динамическую переменную от переменной, объявленной неявно или через ссылку System.Object, значительно отличает то, что она *не является строго типизированной*. Другими словами, динамические данные *не типизированы статически*. Для компилятора C# это выглядит так, что элемент данных, объявленный с ключевым словом dynamic, может получить какое угодно начальное значение, и на протяжении времени его существования это значение может быть заменено новым (и возможно, не связанным с первоначальным). Рассмотрим следующий метод и его результирующий вывод:

```
static void ChangeDynamicDataType()
{
  // Объявить одиночный элемент данных dynamic по имени t.
  dynamic t = "Hello!";
  Console.WriteLine("t is of type: {0}", t.GetType());
  t = false;
  Console.WriteLine("t is of type: {0}", t.GetType());
  t = new List<int>();
  Console.WriteLine("t is of type: {0}", t.GetType());
}
```

Вот как выглядит вывод:

```
t is of type: System.String
t is of type: System.Boolean
t is of type: System.Collections.Generic.List`1[System.Int32]
```

Имейте в виду, что приведенный выше код успешно бы скомпилировался и дал идентичные результаты, если бы переменная t была объявлена с типом System.Object. Тем не менее, как вскоре будет показано, ключевое слово dynamic предоставляет много дополнительных возможностей.

Вызов членов на динамически объявленных данных

Теперь, учитывая, что тип данных dynamic может на лету принимать идентичность любого типа (как переменная типа System.Object), следующий вопрос, который наверняка возник, связан с вызовом членов на динамической переменной (свойств, методов, индексаторов, регистрации событий и т.п.). В отношении синтаксиса никаких отличий нет. Нужно просто применить операцию точки к динамической переменной, указать общедоступный член и передать ему необходимые аргументы.

Однако (и это очень важно) корректность указываемых членов компилятором не проверяется! Помните, что в отличие от переменной, объявленной как System.Object, динамические данные не являются статически типизированными. Вплоть до времени выполнения не известно, поддерживают ли вызываемые динамические данные указанный член, переданы ли корректные параметры, правильно ли указан член, и т.д. Поэтому, как бы странно это не выглядело, следующий метод скомпилируется без ошибок:

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";
  Console.WriteLine(textData1.ToUpper());
  // Здесь следовало ожидать ошибку компилятора!
  // Но все компилируется нормально.
  Console.WriteLine(textData1.toupper());
  Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
}
```

Обратите внимание, что во втором вызове WriteLine() производится обращение к методу по имени toupper() на динамической переменной. Как видите, textData1 имеет тип string, и потому известно, что у этого типа нет метода с таким именем в

нижнем регистре. Более того, тип string определенно не имеет метода по имени Foo(), который принимает int, string и DataTime!

Тем не менее, компилятор C# не о каких ошибках не сообщает. Однако если вызвать этот метод в Main(), возникнет ошибка времени выполнения с примерно таким сообщением:

```
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
'string' does not contain a definition for 'toupper'
```

Необработанное исключение: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: 'string' не содержит определения 'toupper'

Другое значительное отличие между вызовом членов на динамических и строго типизированных данных состоит в том, что после применения операции точки к элементу динамических данных средство IntelliSense в Visual Studio 2010 не активизируется. Вместо этого отображается следующее общее сообщение (рис. 18.1).

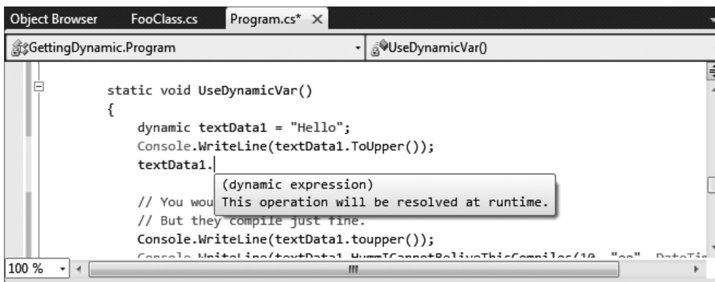

Рис. 18.1. Динамические данные не активизируют средство IntelliSense

То, что средство IntelliSense недоступно с динамическими данными, имеет смысл. Однако это означает, что при наборе кода C# с такими переменными следует соблюдать исключительную осторожность. Любая опечатка или некорректный регистр символов в имени члена приведет к ошибке времени выполнения, а именно — к генерации экземпляра класса RuntimeBinderException.

Роль сборки Microsoft.CSharp.dll

Сразу же после создания нового проекта C# в Visual Studio 2010 автоматически получается комплект ссылок на новую сборку .NET 4.0 по имени Microsoft.CSharp.dll (в этом легко убедиться, заглянув в папку References (Ссылки) в проводнике решений). Эта очень маленькая библиотека определяет единственное пространство имен (Microsoft.CSharp.RuntimeBinder) с двумя классами (рис. 18.2).

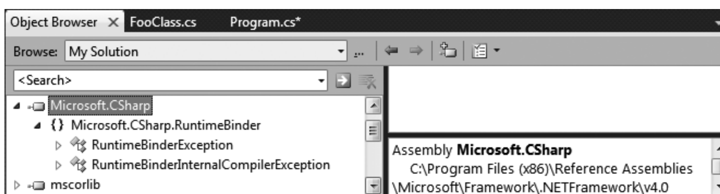

Рис. 18.2. Сборка Microsoft.CSharp.dll

Как можно догадаться по их именам, оба класса представляют собой строго типизированные исключения. Более общий класс — RuntimeBinderException — представляет ошибку, которая будет сгенерирована при попытке вызова несуществующего члена на ди-

намическом типе данных (как в случае методов toupper() и Foo()). Та же ошибка будет инициирована, если будут указаны неверные данные параметров для существующего члена.

Поскольку динамические данные столь изменчивы, каждый вызов члена на переменной, объявленной с ключевым словом dynamic, должен быть помещен в правильный блок try/catch, и предусмотрена соответствующая обработка ошибок.

```
static void InvokeMembersOnDynamicData()
{
  dynamic textData1 = "Hello";
  try
  {
    Console.WriteLine(textData1.ToUpper());
    Console.WriteLine(textData1.toupper());
    Console.WriteLine(textData1.Foo(10, "ee", DateTime.Now));
  }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Вызвав этот метод вновь, можно увидеть, что вызов ToUpper() (обратите внимание на регистр "T" и "U") работает корректно, однако на консоль выводится следующее сообщение об ошибке:

```
HELLO
'string' does not contain a definition for 'toupper'
```

```
HELLO
'string' не содержит определения 'toupper'
```

Разумеется, процесс помещения всех динамических вызовов методов в блоки try/catch довольно утомителен. Если вы тщательно следите за написанием кода и передачей параметров, то это делать не обязательно. Однако перехват исключений удобен, когда заранее не известно, будет ли член представлен в целевом типе.

Область применения ключевого слова dynamic

Вспомните, что неявно типизированные данные возможны только для локальных переменных в области определения члена. Ключевое слово var никогда не может использоваться в качестве возвращаемого значения, параметра или члена класса/структуры. Однако это не касается ключевого слова dynamic. Взгляните на следующее определение класса:

```
class VeryDynamicClass
{
  // Поле dynamic.
  private static dynamic myDynamicField;
  // Свойство dynamic.
  public dynamic DynamicProperty { get; set; }
  // Тип возврата dynamic и тип параметра dynamic.
  public dynamic DynamicMethod(dynamic dynamicParam)
  {
    // Локальная переменная dynamic.
    dynamic dynamicLocalVar = "Local variable";
    int myInt = 10;
    if (dynamicParam is int)
    {
      return dynamicLocalVar;
    }
```

```
      else
      {
        return myInt;
      }
    }
  }
```

Теперь можно вызывать общедоступные члены, как ожидалось, однако, при оперировании с динамическими методами и свойствами нет полной уверенности в том, каким именно будет тип данных! По правде говоря, определение VeryDynamicClass может оказаться не особенно полезным в реальном приложении, но оно иллюстрирует область применения ключевого слова dynamic.

Ограничения ключевого слова dynamic

Хотя с использованием ключевого слова dynamic можно определить очень много вещей, с ним связаны свои ограничения. Хотя они не так уж существенны, имейте в виду, что элементы динамических данных не могут использовать лямбда-выражения или анонимные методы C# при вызове метода. Например, следующий код всегда приводит к ошибке, даже если целевой метод на самом деле принимает параметр-делегат, который, в свою очередь, принимает значение string и возвращает void:

```
dynamic a = GetDynamicObject();
// Ошибка! Методы на динамических данных не могут использовать лямбда-выражения!
a.Method(arg => Console.WriteLine(arg));
```

Чтобы обойти это ограничение, понадобится работать с лежащим в основе делегатом напрямую, используя технику, описанную в главе 11 (анонимные методы и лямбда-выражения, и т.д.). Другое ограничение состоит в том, что динамический элемент данных не может воспринимать расширяющие методы (см. главу 12). К сожалению, это касается также всех расширяющих методов из API-интерфейсов LINQ. Поэтому переменная, объявленная с ключевым словом dynamic, имеет очень ограниченное применение в рамках LINQ to Objects и других технологий LINQ:

```
dynamic a = GetDynamicObject();
// Ошибка! Динамические данные не могут найти расширяющий метод Select()!
var data = from d in a select d;
```

Практическое применение ключевого слова dynamic

Учитывая тот факт, что динамические данные не являются строго типизированными, не проверяются во время компиляции, не имеют возможности инициировать средство IntelliSense и не могут быть целью запроса LINQ, совершенно корректно предположить, что использование ключевого слова dynamic только потому, что оно существует — это очень плохая программистская практика.

Однако в редких случаях ключевое слово dynamic может радикально сократить объем кода, который придется вводить вручную. В частности, при построении приложения .NET, которое интенсивно использует позднее связывание (через рефлексию), ключевое слово dynamic может сэкономить время на наборе кода. Точно также, при разработке приложения .NET, которое должно взаимодействовать с унаследованными библиотеками COM (такими как продукты Microsoft Office), можно значительно упростить код за счет применения ключевого слова dynamic.

Как с любым “сокращением”, прежде чем его применять, необходимо взвесить все “за” и “против”. Применение ключевого слова dynamic — это компромисс между краткостью кода и безопасностью типов. Хотя C# в основе своей является строго типизированным языком, можно выбирать, стоит ли пользоваться динамическим поведением,

от вызова к вызову. Помните, что вы не обязаны применять ключевое слово `dynamic`. Всегда можно получить тот же конечный результат, написав альтернативный код вручную (обычно существенно большего объема).

Исходный код. Проект `DynamicKeyword` доступен в подкаталоге `Chapter 18`.

Роль исполняющей среды динамического языка (DLR)

Теперь, когда прояснилась суть "динамических данных", давайте исследуем, как они обрабатываются. В версии .NET 4.0 общеязыковая исполняющая среда (Common Language Runtime — CLR) получила дополняющую среду времени выполнения, которая называется *исполняющей средой динамического языка* (Dynamic Language Runtime — DLR). Концепция "динамической исполняющей среды" определенно не нова. На самом деле ее много лет используют такие языки программирования, как Smalltalk, LISP, Ruby и Python. В основе своей динамическая исполняющая среда предоставляет динамическим языкам возможность обнаруживать типы целиком во время выполнения, без каких-либо проверок при компиляции.

При наличии опыта работы со строго типизированными языками (включая C# без динамических типов), может показаться нежелательным само понятие такой исполняющей среды. В конце концов, обычно, когда только возможно, лучше получать ошибки во время компиляции, а не во время выполнения. Тем не менее, динамические языки и исполняющие среды предлагают ряд интересных возможностей, включая перечисленные ниже.

- Исключительно гибкая кодовая база. Можно изменять код, не внося многочисленных модификаций в типы данных.
- Очень простой способ взаимодействия с разнообразными типами объектов, построенными на разных платформах и языках программирования.
- Способ добавления или удаления членов типа в памяти во время выполнения.

Роль DLR состоит в том, чтобы позволить различным динамическим языка работать с исполняющей средой .NET и предоставлять им возможность взаимодействия с другим кодом .NET. Два популярных динамических языка, которые используют DLR — это IronPython и IronRuby. Эти языки живут в "динамической вселенной", где типы определяются исключительно во время выполнения. К тому же эти языки имеют доступ ко всему богатству библиотек базовых классов .NET. Еще лучше то, что их кодовая база может взаимодействовать с C# (и наоборот), благодаря включению ключевого слова `dynamic`.

Роль деревьев выражений

Среда DLR использует *деревья выражений* для описания динамического вызова в нейтральных терминах. Например, когда DLR встречает код C#, подобный следующему:

```
dynamic d = GetSomeData();
d.SuperMethod(12);
```

то автоматически строит дерево выражения, которое, по сути, гласит: "Вызвать метод по имени `SuperMethod` на объекте `d`, передав `12` в качестве аргумента". Эта информация (формально называемая *рабочей нагрузкой* (payload)) затем передается корректному средству привязки времени выполнения, которое, опять-таки, может быть динамическим средством привязки C#, динамическим средством привязки IronPython или даже (как будет показано ниже) унаследованными объектами COM.

Отсюда запрос отображается на необходимую структуру вызовов для целевого объекта. Замечательным в деревьях выражений является то (помимо того факта, что вам не нужно создавать их вручную), что они позволяют нам написать фиксированный оператор кода C#, не заботясь о том, что собой представляет его реальная цель (объект COM, код IronPython или IronRuby, и т.п.). На рис. 18.3 иллюстрируется концепция деревьев выражений на наивысшем уровне.

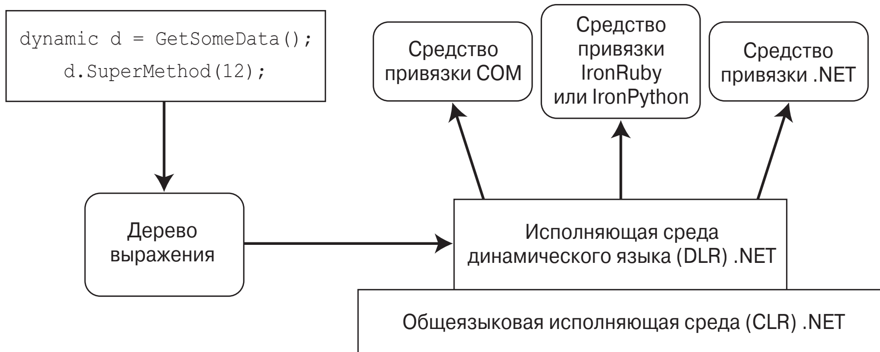

Рис. 18.3. Деревья выражений фиксируют динамические вызовы в нейтральных терминах и обрабатываются средствами привязки

Роль пространства имен `System.Dynamic`

В версии .NET 4.0 к сборке `System.Core.dll` было добавлено пространство имен `System.Dynamic`. По правде говоря, шансы, что вам когда-либо придется непосредственно использовать типы из этого пространства имен, весьма невелики. Однако если вы — разработчик языка, который желает обеспечить своему динамическому языку возможность взаимодействия с DLR, пространство имен `System.Dynamic` пригодится для построения специального средства привязки времени выполнения.

Подробные сведения о типах `System.Dynamic` можно найти в документации .NET Framework 4.0 SDK. Для практических нужд просто знайте, что это пространство имен предоставляет необходимую инфраструктуру, позволяющую обеспечить динамическим языкам взаимодействие с .NET.

Динамический поиск в деревьях выражений во время выполнения

Как уже объяснялось, среда DLR передает деревья выражений целевому объекту, однако на этот процесс оказывает влияние несколько факторов. Если динамический тип данных указывает в памяти на объект COM, то дерево выражения посылается низкоуровневому интерфейсу COM по имени `IDispatch`. Как вам может быть известно, этот интерфейс представляет собой способ, которым COM включает собственный набор динамических служб. Объекты COM, однако, могут использоваться в приложении .NET без применения DLR или ключевого слова C# `dynamic`. Однако это (как вы убедитесь) ведет к более сложному кодированию C#.

Если динамические данные не указывают на объект COM, то дерево выражения может быть передано объекту, реализующему интерфейс `IDynamicObject`. Этот интерфейс используется “за кулисами”, чтобы позволить такому языку, как IronRuby, принять дерево выражения DLR и отобразить его на специфику языка Ruby.

Наконец, если динамические данные указывают на объект, который не является объектом COM и не реализует интерфейс `IDynamicObject`, то это — нормальный,

повседневный объект .NET. В этом случае дерево выражения передается на обработку средству привязки исполняющей среды C#. Процесс отображения дерева выражений на специфику .NET включает участие служб рефлексии.

Как только дерево выражения обработано определенным средством привязки, динамические данные разрешаются в реальный тип данных в памяти, после чего вызывается корректный метод со всеми необходимыми параметрами. Теперь давайте рассмотрим несколько практических применений DLR, начав с упрощения вызовов позднего связывания .NET.

Упрощение вызовов позднего связывания с использованием динамических типов

Одним из случаев, когда имеет смысл использовать ключевое слово `dynamic`, может быть работа со службами рефлексии, а именно — выполнение вызовов методов позднего связывания. В главе 15 приводилось несколько примеров, когда такого рода вызовы методов могут быть очень полезны — чаще всего при построении расширяемого приложения. Там было показано, как использовать метод `Activator.CreateInstance()` для создания экземпляра `object`, о котором ничего не известно во время компиляции (помимо его отображаемого имени). Затем с помощью типов из пространства имен `System.Reflection` можно обращаться к членам через механизм позднего связывания. Вспомните следующий пример из главы 15.

```
static void CreateUsingLateBinding(Assembly asm)
{
  try
  {
    // Получение метаданных типа Minivan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");
    // Создание экземпляра Minivan на лету.
    object obj = Activator.CreateInstance(miniVan);
    // Получение информации о TurboBoost.
    MethodInfo mi = miniVan.GetMethod("TurboBoost");
    // Вызов метода (null означает отсутствие параметров).
    mi.Invoke(obj, null);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Хотя этот код работает, как и ожидалось, нельзя не отметить его громоздкость. Здесь приходится вручную создавать класс `MethodInfo`, вручную запрашивать метаданные и т.д. Ниже приведена версия этого метода, в которой используется ключевое слово `dynamic` и DLR:

```
static void InvokeMethodWithDynamicKeyword(Assembly asm)
{
  try
  {
    // Получение метаданных типа Minivan.
    Type miniVan = asm.GetType("CarLibrary.MiniVan");
    // Создание экземпляра Minivan на лету и вызов метода.
    dynamic obj = Activator.CreateInstance(miniVan);
    obj.TurboBoost();
  }
```

```
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Объявив переменную obj с ключевым словом dynamic, всю рутинную работу, связанную с рефлексией, вы возлагаете на среду DLR!

Использование ключевого слова dynamic для передачи аргументов

Польза от DLR становится еще более очевидной, когда нужно выполнять связанные вызовы методов, принимающих параметры. Когда используются "длинные" вызовы рефлексии, аргументы приходится упаковывать в массив элементов object, который передается методу Invoke() класса MethodInfo.

Чтобы проиллюстрировать это на примере, создадим новое консольное приложение C# по имени LateBindingWithDynamic. Добавим к текущему решению проект библиотеки классов (используя пункт меню File⇨Add⇨New Project (Файл⇨Добавить⇨Новый проект)) и назовите его MathLibrary. Переименуйте начальный класс Class1.cs в проекте MathLibrary на SimplaMath.cs и реализуйте класс, как показано ниже:

```
public class SimpleMath
{
  public int Add(int x, int y)
  { return x + y; }
}
```

Скомпилировав сборку MathLibrary.dll, поместите ее копию в папку bin\Debug проекта LateBindingWithDynamic (щелкнув на кнопке Show All Files (Показать все файлы) для каждого проекта в Solution Explorer, можно просто перетащить файл между проектами). После этого окно Solution Explorer должно выглядеть примерно как на рис. 18.4.

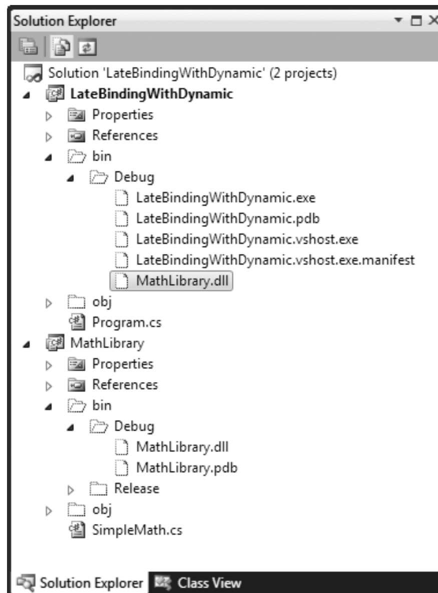

Рис. 18.4. Проект LateBindingWithDynamic имеет приватную копию сборки MathLibrary.dll

На заметку! Помните, что главная цель позднего связывания — позволить приложению создать объект, который не имеет записи `MANIFEST`. Именно поэтому нужно вручную скопировать сборку `MathLibrary.dll` в выходную папку консольного проекта, а не устанавливать ссылку на сборку через Visual Studio.

Теперь импортируем пространство имен `System.Reflection` а файл `Program.cs` проекта консольного приложения. Добавим следующий метод в класс `Program`, который вызывает метод `Add()`, используя типичные вызовы API-интерфейса рефлексии:

```
private static void AddWithReflection()
{
  Assembly asm = Assembly.Load("MathLibrary");
  try
  {
    // Получение метаданных типа SimpleMath.
    Type math = asm.GetType("MathLibrary.SimpleMath");
    // Создание экземпляра SimpleMath на лету.
    object obj = Activator.CreateInstance(math);
    // Получение информации для метода Add.
    MethodInfo mi = math.GetMethod("Add");
    // Вызов метода (с параметрами).
    object[] args = { 10, 70 };
    Console.WriteLine("Result is: {0}", mi.Invoke(obj, args)); // вывод результата
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Ниже показано, как предыдущая логика метода упрощается за счет использования ключевого слова `dynamic`:

```
private static void AddWithDynamic()
{
  Assembly asm = Assembly.Load("MathLibrary");
  try
  {
    // Получение метаданных типа SimpleMath.
    Type math = asm.GetType("MathLibrary.SimpleMath");
    // Создание экземпляра SimpleMath на лету.
    dynamic obj = Activator.CreateInstance(math);
    Console.WriteLine("Result is: {0}", obj.Add(10, 70)); // вывод результата
  }
  catch (Microsoft.CSharp.RuntimeBinder.RuntimeBinderException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Выглядит неплохо! Вызов обоих методов в `Main()` дает идентичный вывод. Однако при использовании ключевого слова `dynamic` сокращается объем работ по кодированию. Для динамически определенных данных больше не нужно вручную упаковывать аргументы в массив `object`, запрашивать метаданные сборки и иметь дело с прочими деталями подобного рода.

Исходный код. Проект `LastBindingWithDynamic` доступен в подкаталоге `Chapter 18`.

Упрощение взаимодействия с COM посредством динамических данных

Теперь давайте рассмотрим другое полезное применение ключевого слова `dynamic` — в контексте проекта взаимодействия с COM. Если нет опыта разработки для COM, то имейте в виду, что следующий пример, в котором компилируется библиотека COM, содержит метаданные, подобно библиотеке .NET. Тем не менее, ее формат совершенно отличается. По этой причине если программа .NET нуждается в использовании объекта COM, то первое, что нужно сделать — это сгенерировать то, что называется "сборкой взаимодействия" (interop assembly), используя Visual Studio 2010. Делается это довольно легко. Просто откройте диалоговое окно Add Reference (Добавить ссылку), перейдите на вкладку COM и найдите библиотеку COM, которую необходимо использовать (рис. 18.5).

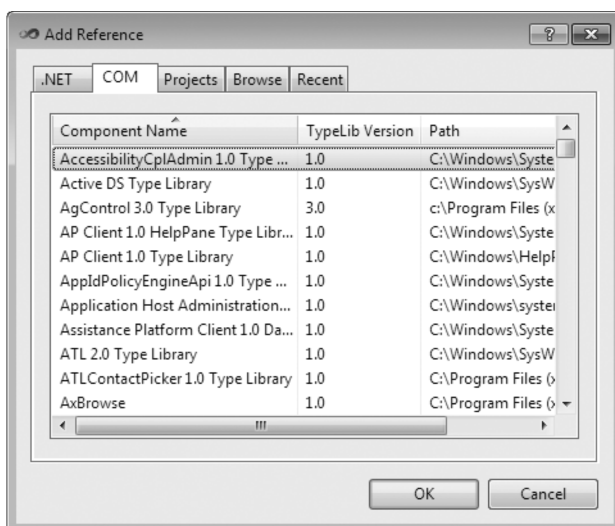

Рис. 18.5. На вкладке COM диалогового окна Add Reference отображаются все зарегистрированные на машине библиотеки COM

В случае выбора библиотеки COM среда Visual Studio 2010 отреагирует генерацией совершенно нового описания .NET для метаданных COM. Формально они называются "сборками взаимодействия" и не содержат никакого кода реализации, помимо самого минимума, который помогает транслировать события COM в события .NET. Однако эти сборки взаимодействия очень полезны в том, что защищают кодовую базу .NET от сложностей внутреннего механизма COM.

В коде C# можно напрямую работать со сборкой взаимодействия, позволяя CLR (и DLR, если используется ключевое слово `dynamic`) автоматически отображать типы данных .NET на типы COM и наоборот. "За кулисами" данные маршализуются между приложениями .NET и COM с использованием оболочки исполняющей среды (Runtime Callable Wrapper — RCW), которая на самом деле является динамически сгенерированным прокси. RCW маршализует и трансформирует типы данных .NET в типы COM, изменяя счетчик ссылок COM-объекта и отображая все возвращаемые COM значения на их эквиваленты в .NET.

На рис. 18.6 показана общая картина взаимодействия .NET с COM.

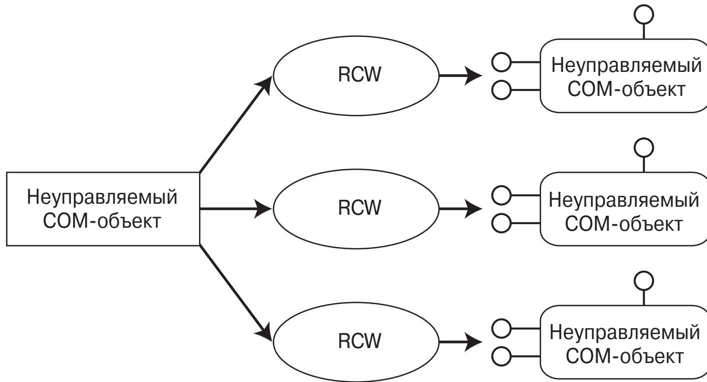

Рис. 18.6. Программы .NET взаимодействуют с объектами COM, используя прокси под названием RCW

Роль первичных сборок взаимодействия

Многие поставщики библиотек COM (таких как библиотеки Microsoft COM, обеспечивающие доступ к объектной модели продуктов Microsoft Office), предоставляют “официальную” сборку взаимодействия, которая называется *первичной сборкой взаимодействия* (primary interop assembly — PIA). Сборки PIA — это оптимизированные сборки взаимодействия, которые делают яснее (и возможно, расширяют) код, обычно генерируемый при ссылке на библиотеку COM через диалоговое окно Add Reference.

Сборки PIA обычно перечисляются на вкладке .NET диалогового окна Add Reference, подобно базовым библиотекам .NET. Фактически, если вы ссылаетесь на библиотеку COM из вкладки COM диалогового окна Add Reference, то Visual Studio не генерирует новой библиотеки взаимодействия, как делает это обычно, а вместо этого использует предоставленную сборку PIA. На рис. 18.7 показана сборка PIA объектной модели Microsoft Office Excel, которая будет использоваться в следующем примере.

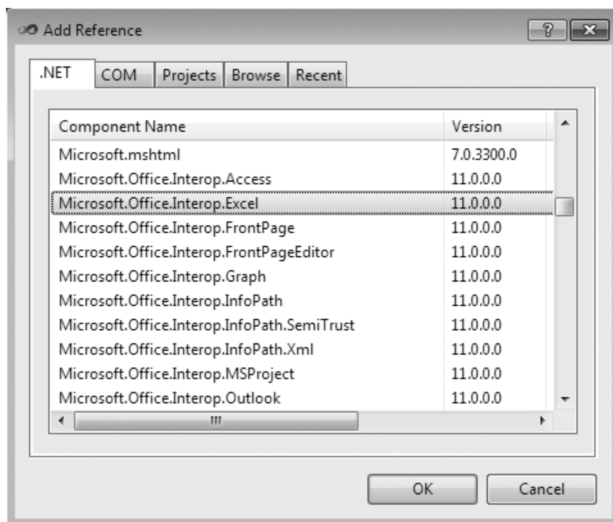

Рис. 18.7. Сборки PIA перечислены на вкладке .NET диалогового окна Add Reference

Встраивание метаданных взаимодействия

До появления .NET 4.0, когда приложение C# использовало библиотеку COM (в виде сборки PIA или нет), нужно было обеспечить наличие на клиентской машине копии сборки взаимодействия. Это увеличивало размер установочного пакета приложения, к тому же в сценарии установки должно было проверяться существование сборки PIA и в случае ее отсутствия — установка копии в GAC.

Однако в .NET 4.0 теперь можно встраивать данные взаимодействия непосредственно в скомпилированное приложение .NET. В этом случае поставлять копию сборки взаимодействия вместе с приложением .NET необязательно, поскольку все необходимые метаданные взаимодействия жестко встраиваются в приложение .NET.

По умолчанию, после выбора библиотеки COM (PIA или нет) в диалоговом окне Add Reference интегрированная среда разработки автоматически устанавливает свойство Embed Interop Types (Встраивать типы взаимодействия) библиотеки в `True`. Чтобы увидеть эту установку, необходимо выбрать ссылаемую сборку взаимодействия в папке References (Ссылки) окна Solution Explorer и открыть ее окно свойств (рис. 18.8).

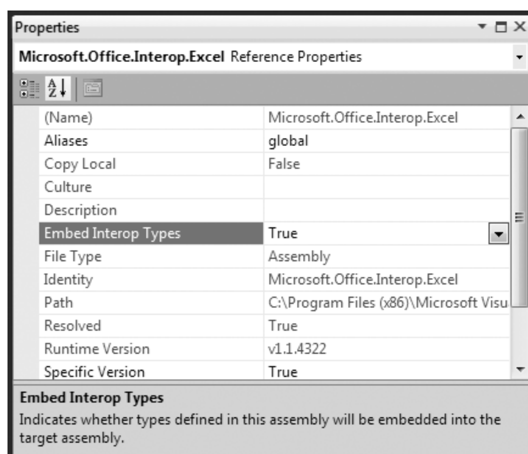

Рис. 18.8. Среда .NET 4.0 позволяет встраивать часть используемых сборок взаимодействия в создаваемую сборку .NET

Компилятор C# включит только те части библиотеки взаимодействия, которые действительно используются. Таким образом, даже если реальная библиотека взаимодействия содержит .NET-описания сотен COM-объектов, будут получены определения только подмножества, которое действительно используется в написанном коде C#. Помимо сокращения размеров приложения, поставляемого клиенту, также упрощается процесс установки, поскольку не понадобится копировать лишние сборки PIA на целевую машину.

Общие сложности взаимодействия с COM

До выхода версии .NET 4.0 при написании кода C#, имеющего дело с библиотекой COM (через сборку взаимодействия), неизбежно возникало множество сложностей. Например, многие COM-библиотеки определяют методы, принимающие необязательные аргументы, что вплоть до нынешнего выпуска в C# не поддерживалось. Это требовало указания значения `Type.Missing` для каждого появления необязательного аргумента. Например, если метод COM принимал пять аргументов, и все они были необязательны, приходилось писать следующий код C#, чтобы принять значения по умолчанию:

```
myComObj.SomeMethod(
      Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

К счастью, в .NET 4.0 теперь можно писать упрощенный код, учитывая, что значения `Type.Missing` будут вставлены во время компиляции, если не указано реальное значение:

```
myComObj.SomeMethod();
```

В связи с этим стоит отметить, что многие методы COM предлагают поддержку именованных аргументов, которые, как было показано в главе 4, позволяют передавать значения членам в любом порядке. Поскольку в .NET 4.0 язык C# поддерживает и это средство, можно очень просто "пропускать" множество необязательных аргументов и устанавливать только те, которые важны в данном случае.

Другая сложность взаимодействия с COM была связана с тем фактом, что многие методы COM спроектированы так, чтобы принимать и возвращать очень специфический тип данных по имени `Variant`. Во многом подобный ключевому слову C# `dynamic`, типу данных `Variant` может быть присвоен любой тип данных COM на лету (строка, интерфейсная ссылка, числовое значение и т.п.). До появления ключевого слова `dynamic` передача или прием элементов данных типа `Variant` требовал значительных ухищрений, обычно связанных с многочисленными операциями приведения.

С появлением .NET 4.0 и Visual Studio 2010, кода свойство Embed Interop Types устанавливается в `True`, все типы `Variant` из COM автоматически отображаются на динамические данные. Это не только сокращает потребность в излишних операциях приведения при работе с типом данных `Variant`, но также еще более скрывает некоторые сложности, присущие COM, вроде работы с индексаторами COM.

Для того чтобы продемонстрировать упрощение взаимодействия с COM за счет совместной работы необязательных аргументов, именованных аргументов и ключевого слово `dynamic` в C#, построим приложение, в котором используется объектная модель Microsoft Office. При работе с этим примером вы получите шанс применить новые средства, а также обойтись без них, и затем сравнить объем работ в обоих случаях.

На заметку! В предыдущих изданиях этой книги детально рассматривалась работа с унаследованными объектами COM в проектах .NET с использованием пространства имен `System.InteropServices`. В нынешнем издании это не описано, поскольку ключевое слово `dynamic` обеспечивает гораздо более простое взаимодействие с объектами COM. Исчерпывающие сведения о взаимодействии с объектами COM в строго типизированной (длинной) нотации могут быть найдены в документации .NET Framework 4.0 SDK.

Взаимодействие с COM с использованием средств языка C# 4.0

Предположим, что имеется приложение Windows Forms с графическим интерфейсом пользователя (`ExportDataToOfficeApp`), главная форма которого определяет элемент управления `DataGridView` по имени `dataGridCars`. В той же форме находятся два элемента управления `Button`, один из которых обеспечивает открытие диалогового окна для вставки новой строки данных в сетку, а другой отвечает за экспорт данных сетки в электронную таблицу Excel. Учитывая тот факт, что приложение Excel предоставляет программную модель через COM, к ней можно привязаться с использованием уровня взаимодействия. На рис. 18.9 показан завершенный графический интерфейс пользователя.

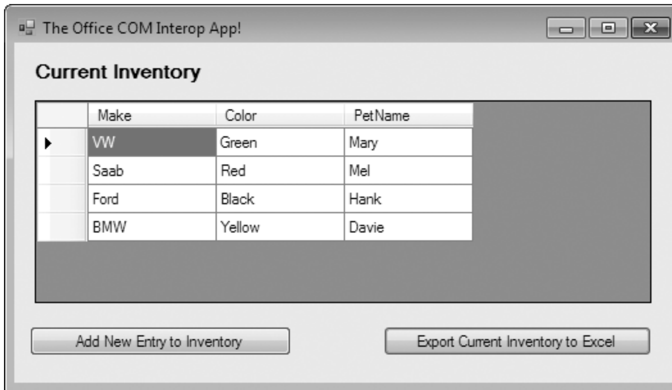

Рис. 18.9. Графический интерфейс пользователя для примера взаимодействия с COM

Сетка будет заполняться некоторыми начальными данными в обработчике события `Load` формы (класс `Car`, используемый в качестве параметра типа для обобщенного `List<T>` — это простой класс в проекте, имеющий свойства `Color`, `Make` и `PetName`):

```
public partial class MainForm : Form
{
  List<Car> carsInStock = null;
  public MainForm()
  {
    InitializeComponent();
  }

  private void MainForm_Load(object sender, EventArgs e)
  {
    carsInStock = new List<Car>
    {
       new Car {Color="Green", Make="VW", PetName="Mary"},
       new Car {Color="Red", Make="Saab", PetName="Mel"},
       new Car {Color="Black", Make="Ford", PetName="Hank"},
       new Car {Color="Yellow", Make="BMW", PetName="Davie"}
    };
    UpdateGrid();
  }

  private void UpdateGrid()
  {
    // Сбросить источник данных.
    dataGridCars.DataSource = null;
    dataGridCars.DataSource = carsInStock;
  }
}
```

В обработчике события `Click` кнопки Add New Entry to Inventory (Добавить новую запись в инвентарную ведомость) открывается специальное диалоговое окно, которое позволяет пользователю ввести новые данные для объекта `Car`; после щелчка на кнопке OK данные добавляются в сетку. Код этого диалогового окна в книге не показан, поэтому за подробностями обращайтесь к доступному решению. Если хотите повторить пример самостоятельно, включите файлы `NewCarDialog.cs`, `NewCarDialog.designer.cs` и `NewCarDialog.resx` в проект (их можно найти в составе кода примеров для этой главы). Затем реализуйте обработчик щелчка на кнопке Add New Entry to Inventory, как показано ниже:

```
private void btnAddNewCar_Click(object sender, EventArgs e)
{
  NewCarDialog d = new NewCarDialog();
  if (d.ShowDialog() == DialogResult.OK)
  {
    // Добавить новый автомобиль в список.
    carsInStock.Add(d.theCar);
    UpdateGrid();
  }
}
```

Ядром этого примера является обработчик события `Click` для кнопки Export Current Inventory to Excel (Экспортировать текущую инвентарную ведомость в Excel). На вкладке .NET диалогового окна Add Reference добавьте ссылку на первичную сборку взаимодействия `Microsoft.Office.Interop.Excel.dll` (как было показано ранее на рис. 18.7). Добавьте приведенный ниже псевдоним пространства имен в главный файл кода формы. Имейте в виду, что при взаимодействии с библиотеками COM псевдоним определять не обязательно. Однако, поступив так, вы получите удобный квалификатор для всех импортированных объектов COM, что очень пригодится, если некоторые из этих COM-объектов будут иметь имена, конфликтующие с типами .NET.

```
// Создать псевдоним для объектной модели Excel.
using Excel = Microsoft.Office.Interop.Excel;
```

Реализуйте следующий обработчик события `Click`, чтобы он вызывал вспомогательную функцию по имени `ExportToExcel()`:

```
private void btnExportToExcel_Click(object sender, EventArgs e)
{
  ExportToExcel(carsInStock);
}
```

Поскольку библиотека COM была импортирована в Visual Studio 2010, сборка PIA автоматически сконфигурирована так, что используемые метаданные будут включены в приложение .NET (вспомните роль свойства Embed Interop Types). Таким образом, все COM-типы `Variant` будут реализованы как типы данных `dynamic`. Более того, поскольку код пишется на C# 4.0, можно использовать необязательные и именованные аргументы. С учетом всего сказанного вот как будет выглядеть реализация `ExportToExcel()`:

```
static void ExportToExcel(List<Car> carsInStock)
{
  // Загрузить Excel, затем создать новую пустую рабочую книгу.
  Excel.Application excelApp = new Excel.Application();

  excelApp.Workbooks.Add();
  // В этом примере используется единственная рабочий лист.
  Excel._Worksheet workSheet = excelApp.ActiveSheet;

  // Установить заголовки столбцов в ячейках.
  workSheet.Cells[1, "A"] = "Make";
  workSheet.Cells[1, "B"] = "Color";
  workSheet.Cells[1, "C"] = "Pet Name";

  // Отобразить все данные в List<Car> на ячейки электронной таблицы.
  int row = 1;
  foreach (Car c in carsInStock)
  {
    row++;
    workSheet.Cells[row, "A"] = c.Make;
    workSheet.Cells[row, "B"] = c.Color;
    workSheet.Cells[row, "C"] = c.PetName;
  }
```

```
  // Придадить симпатичный вид табличным данным.
  workSheet.Range["A1"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

  // Сохранить файл, выйти из Excel и отобразить сообщение пользователю.
  workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx", Environment.CurrentDirectory));
  excelApp.Quit();
  MessageBox.Show("The Inventory.xslx file has been saved to your app folder",
    "Export complete!"); // файл Inventory.xslx сохранен в папке приложения
}
```

Метод начинается с загрузки приложения Excel в память, однако на рабочем столе компьютера оно не покажется. В данном приложении интересует только использование объектной модели Excel. Если же необходимо отобразить пользовательский интерфейс Excel, дополните метод следующей строкой кода:

```
static void ExportToExcel(List<Car> carsInStock)
{
  // Загрузить Excel, затем создать новую пустую рабочую книгу.
  Excel.Application excelApp = new Excel.Application();

  // Сделать приложение Excel видимым.
  excelApp.Visible = true;
...
}
```

После создания пустого рабочего листа к нему добавляются три столбца, названные по именам свойств класса `Car`. После этого ячейки заполняются данными `List<Car>` и файл сохраняется под жестко закодированным именем `Inventory.xlsx`.

Если теперь запустить приложение, добавить несколько записей и экспортировать их в Excel, в папке `bin\Debug` приложения Windows Forms появится файл `Inventory.xlsx`, который можно открыть в приложении Excel (рис. 18.10).

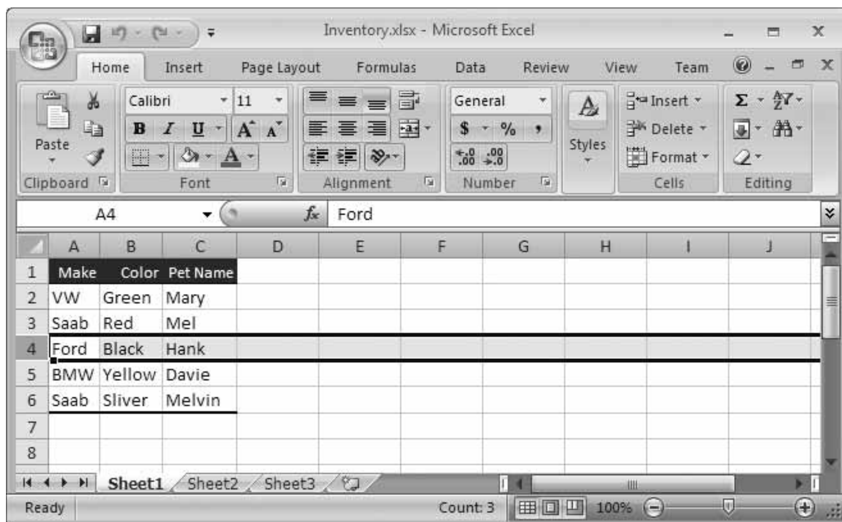

Рис. 18.10. Экспортированные данные в файле Excel

Взаимодействие с COM без использования средств языка C# 4.0

Если теперь выбрать сборку Microsoft.Office.Interop.Excel.dll в Solution Explorer и установить свойство Embed Interop Type в False, появятся сообщения об ошибках компиляции, поскольку COM-данные Variant будут трактоваться не как динамические данные, а как переменные System.Object. Это потребует добавления в ExportToExcel() нескольких операций приведения. Кроме того, если проект скомпилировать в версии Visual Studio 2008, утратятся преимущества необязательных/именованных параметров, и в этом случае понадобится явно помечать все пропущенные аргументы. Ниже показана версия метода ExportToExcel() для ранних версий C#.

```
static void ExportToExcel2008(List<Car> carsInStock)
{
  Excel.Application excelApp = new Excel.Application();

  // Нужно пометить пропущенные параметры!
  excelApp.Workbooks.Add(Type.Missing);

  // Нужно привести Object к _Worksheet!
  Excel._Worksheet workSheet = (Excel._Worksheet)excelApp.ActiveSheet;

  // Нужно привести каждый Object к объекту Range,
  // затем вызвать низкоуровневое свойство Value2!
  ((Excel.Range)excelApp.Cells[1, "A"]).Value2 = "Make";
  ((Excel.Range)excelApp.Cells[1, "B"]).Value2 = "Color";
  ((Excel.Range)excelApp.Cells[1, "C"]).Value2 = "Pet Name";
  int row = 1;
  foreach (Car c in carsInStock)
  {
    row++;
    // Нужно привести каждый Object к объекту Range,
    // и вызвать низкоуровневое свойство Value2!
    ((Excel.Range)workSheet.Cells[row, "A"]).Value2 = c.Make;
    ((Excel.Range)workSheet.Cells[row, "B"]).Value2 = c.Color;
    ((Excel.Range)workSheet.Cells[row, "C"]).Value2 = c.PetName;
  }

  // Нужно вызвать метод get_Range и указаать все пропущенные аргументы!
  excelApp.get_Range("A1", Type.Missing).AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2,
    Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing);

  // Нужно указать все пропущенные аргументы!
  workSheet.SaveAs(string.Format(@"{0}\Inventory.xlsx", Environment.CurrentDirectory),
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);
  excelApp.Quit();
  MessageBox.Show("The Inventory.xslx file has been saved to your app folder",
    "Export complete!"); // файл Inventory.xslx сохранен в папке приложения
}
```

Хотя конечный результат идентичен, очевидно, что данная версия метода намного более многословна. К тому же, поскольку ранние версии C# (точнее, предшествовавшие .NET 4.0) не позволяют встраивать данные взаимодействия COM, обнаружится, что выходная папка теперь содержит локальные копии множества сборок взаимодействия, которые должны будут поставлены на машину конечного пользователя (рис. 18.11).

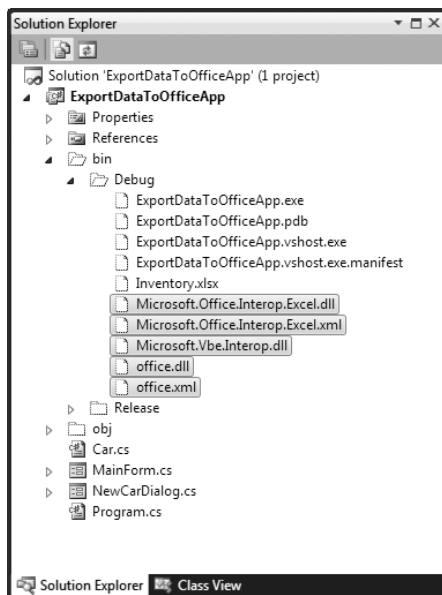

Рис. 18.11. Если данные взаимодействия не встроены, потребуется поставлять автономные сборки взаимодействия

На этом рассмотрение ключевого слова C# `dynamic` и среды DLR завершено. Наверняка вы смогли оценить, насколько новые средства .NET 4.0 могут упростить решение сложных задач программирования, и (что возможно, более важно) поняли сопутствующие компромиссы. Выбор в пользу динамических данных приводит к утере безопасности типов, поэтому код становится уязвимым для гораздо большего числа ошибок времени выполнения.

Исходный код. Проект `ExportDataToOfficeApp` доступен в подкаталоге `Chapter 18`.

Резюме

Ключевое слово `dynamic` в C# 4.0 позволяет определять данные, истинная идентичность которых не известна вплоть до времени выполнения. При работе новой исполняющей среды динамического языка (DLR) автоматически создаваемое “дерево выражения” передается соответствующему средству привязки динамического языка, причем рабочие данные будут распакованы и отправлены корректному члену объекта.

За счет использования динамических данных и DLR многие сложные задачи программирования C# могут быть радикально упрощены, а особенно — включение библиотек COM в приложения .NET. Кроме того, как было показано в этой главе, .NET 4.0 предлагает ряд дальнейших упрощений взаимодействия с COM (которые не имеют отношения к динамическим данным) — встраивание данных взаимодействия COM в разрабатываемые приложения, а также необязательные и именованные аргументы.

Хотя все эти средства могут упростить код, не забывайте, что динамические данные существенно снижают безопасность кода C# в отношении типов и открывают путь для ошибок времени выполнения. Поэтому тщательно взвешивайте все “за” и “против” использования динамических данных в проектах C# и соответствующим образом тестируйте их.