



51

Технология Enterprise Services

В ЭТОЙ ГЛАВЕ...

- Средства Enterprise Services
- Использование Enterprise Services
- Создание обслуживаемого компонента
- Развертывание приложений COM+
- Использование транзакций с COM+
- Создание фасада WCF для Enterprise Services
- Использование Enterprise Services из клиента WCF

Enterprise Services (службы предприятия) — это название серверной технологии Microsoft, предоставляющей службы для распределенных решений. Enterprise Services основаны на технологии COM+, которая уже используется многие годы. Однако вместо помещения объектов .NET в оболочку объектов COM, для применения этих служб .NET предлагает расширения компонентов .NET, позволяющие напрямую воспользоваться преимуществами этих служб. В результате вы получаете легкий доступ к службам COM+ из компонентов .NET.

Технология Enterprise Services также замечательно интегрируется с платформой Windows Communication Foundation (WCF). С помощью ряда инструментов можно автоматически строить пользовательский интерфейс службы WCF для обслуживаемого компонента, а также можно обращаться к службе WCF из клиента COM+.



В этой главе используется база данных примеров Northwind, которая доступна для загрузки на странице www.microsoft.com/downloads.

Использование Enterprise Services

Всю сложность Enterprise Services и разнообразные конфигурационные опции (многие из которых не нужны, если все компоненты решения разработаны на .NET) понять легче, если знать историю появления Enterprise Services. Глава начинается с изложения этой истории. После этого будет дан обзор различных служб, предлагаемых этой технологией, что позволит определить, какие средства могут понадобиться приложению.

В следующем разделе будет описана история Enterprise Services, контексты, лежащие в основе ее функциональности, а также ключевые средства, такие как автоматические транзакции, организация пула объектов, безопасность на основе ролей, компоненты с очередями и слабо связанные события.

История

Enterprise Services восходит к серверу транзакций Microsoft (Microsoft Transaction Server — MTS), который появился в виде дополнения для Windows NT 4.0. Сервер MTS расширяет COM, предоставляя такие службы, как транзакции для объектов COM. Службы могут быть легко использованы для конфигурирования метаданных: конфигурация компонента должна указывать, требуются ли ему транзакции. В MTS нет необходимости иметь дело с транзакциями программно. Однако с MTS связан и большой недостаток. Технология COM не была задумана как расширяемая, поэтому MTS расширяет ее посредством перезаписи конфигурации реестра COM, чтобы напрямую создавать экземпляры компонентов MTS, а некоторые специальные вызовы API MTS требуют создания объектов COM внутри MTS. Эта проблема была решена с выходом ОС Windows 2000.

Одним из наиболее важных средств Windows 2000 стала интеграция MTS и COM в новую технологию по имени COM+. В Windows 2000 базовые службы COM+ осведомлены о контексте, необходимом службам COM+ (ранее — службам MTS), поэтому специальные вызовы MTS API стали не нужны. В службах COM+ была представлена некоторая новая функциональность служб в дополнение к распределенным транзакциям.

В состав Windows 2000 входит COM+ 1.0. Версия COM+ 1.5 доступна, начиная с Windows XP и Windows Server 2003. В COM+ 1.5 добавлены дополнительные средства для повышения масштабируемости и доступности, включая организацию пула приложений и повторное использование, а также конфигурируемые уровни изоляции.

Технология .NET Enterprise Services позволяет использовать службы COM+ внутри компонентов .NET. Поддержка предоставляется для ОС Windows 2000 и последующих версий. Когда компоненты .NET выполняются внутри приложений COM+, никаких вызываемых

оболочек COM не используется (см. главу 26); вместо этого приложение выполняется как компонент .NET. После установки исполняющей среды .NET, к службам COM+ добавляются некоторые расширения времени выполнения. Если в Enterprise Services установлены два компонента .NET, и компонент А использует компонент В, то маршализация COM не применяется; вместо этого компоненты .NET могут вызывать друг друга напрямую.

Где применяется технология Enterprise Services

Бизнес-приложения могут разделяться логически на уровни представления, бизнес-служб и данных. *Уровень службы представления* (presentation service layer) отвечает за взаимодействие с пользователем. Здесь пользователь может взаимодействовать с приложением для ввода и просмотра данных. На этом уровне применяются технологии Windows Forms и ASP.NET Web Forms. *Уровень бизнес-службы* (business service layer) состоит из бизнес-правил и правил данных. *Уровень службы данных* (data service layer) взаимодействует с постоянным хранилищем. Здесь можно применять компоненты, использующие ADO.NET. Технология Enterprise Services подходит как для уровня бизнес-службы, так и для уровня службы данных.

На рис. 51.1 показаны два типичных сценария приложений. Технология Enterprise Services может использоваться непосредственно из “толстого” клиента с применением Windows Forms или WPF либо же из веб-приложения ASP.NET.

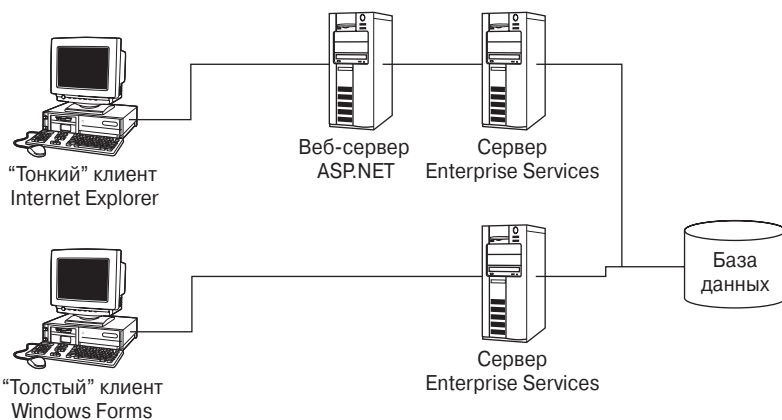


Рис. 51.1. Типичные сценарии приложений

Технология Enterprise Services также является масштабируемой. Применение балансировки загрузки компонентов обеспечивает возможность распределения нагрузки от клиентов между разными системами.

Использовать Enterprise Services допускается также и на клиентской системе, потому что эта технология входит в состав ОС Windows 7 и Windows Vista.

Ключевые средства

Давайте начнем анализ преимуществ Enterprise Services с рассмотрения ключевых средств этой технологии, таких как автоматические транзакции, безопасность на основе ролей, компоненты с очередями и слабо связанные события.

Контексты

Базовая функциональность, положенная в основу служб, предоставляемых Enterprise Services — это контекст, который является основой всех средств Enterprise Services. Контекст обеспечивает возможность перехвата вызова метода и выполнения некоторой

служебной функциональности до осуществления самого вызова. Например, перед вызовом метода, реализованного компонентом, может быть создан транзакционный контекст или контекст синхронизации.

На рис. 51.2 показаны объекты А и В, выполняющиеся в двух разных контекстах — X и Y. Между контекстами вызов перехватывается прокси-объектом. Прокси-объект может использовать службы, предлагаемые Enterprise Services, которые объясняются далее в главе.

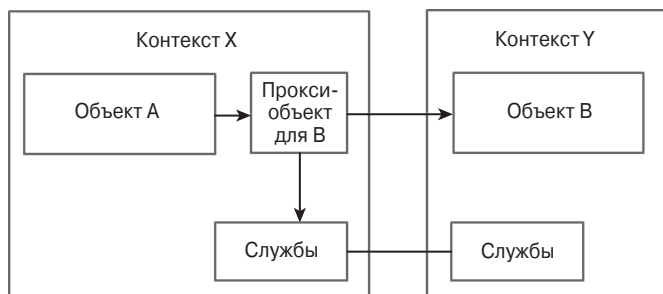


Рис. 51.2. Объекты, выполняющиеся в разных контекстах

С помощью контекстов компонент COM и компонент .NET могут участвовать в одной транзакции. Все это делается с помощью базового класса `ServiceComponent`, который наследуется от `MarshalByRefObject` и предназначен для интеграции контекстов .NET и COM+.

Автоматические транзакции

Наиболее часто используемым средством Enterprise Services являются *автоматические транзакции*. При автоматических транзакциях не обязательно запускать и фиксировать транзакцию в коде; вместо этого к классу может быть просто применен атрибут. Используя атрибут `[Transaction]` с опциями `Required`, `Supported`, `RequiresNew` и `NotSupported`, можно пометить класс требованиями, которые он предъявляет к транзакциям. Если класс помечен опцией `Required`, то транзакция создается автоматически при запуске метода и фиксируется или отменяется, когда корневой компонент транзакции завершен.

Такой декларативный способ программирования особенно полезен, когда разрабатывается сложная объектная модель. Здесь автоматические транзакции обеспечивают значительное преимущество перед ручным программированием транзакций. Предположим, что имеется объект `Person` и множество объектов `Address` и `Document`, ассоциированных с `Person`, и нужно сохранить объект `Person` вместе со всеми ассоциированными объектами в единой транзакции. Программная реализация транзакций должна означать передачу транзакционного объекта всем взаимосвязанным объектам, чтобы они могли принять участие в единой транзакции. Декларативное применение транзакций избавляет от необходимости передачи транзакционного объекта, потому что все происходит “за кулисами” с использованием контекста.

Распределенные транзакции

Технология Enterprise Services поддерживает не только автоматические транзакции, но также транзакции, распределенные по нескольким базам данных. Транзакции Enterprise Services учитываются координатором распределенных транзакций — `Distributed Transaction Coordinator (DTC)`. Координатор DTC поддерживает базы данных, которые используют протокол `XA` — распространенный двухфазный протокол, реализованный в `SQL Server` и `Oracle`. Единственная транзакция может охватывать запись данных в обе базы данных — `SQL Server` и `Oracle`.

Распределенные транзакции удобны не только для баз данных: единственная транзакция может также охватывать запись данных в базу и запись в очередь сообщений. Если одно из этих двух действий даст сбой, откат будет выполнен для обоих. Очереди сообщений рассматриваются в главе 46.



Технология Enterprise Services поддерживает распространяемые транзакции. Если используется SQL Server и внутри одной транзакции активно только одно соединение, то создается локальная транзакция. Если внутри той же транзакции активен другой транзакционный ресурс, то такая транзакция распространяется на транзакцию DTC.

Далее в этой главе будет показано, как создается компонент, требующий транзакции.

Организация пула объектов

Организация пула (pooling) — это еще одно средство, предоставляемое Enterprise Services. Пул потоков используется службами для ответов на клиентские запросы. Пул объектов может применяться для объектов с длительным временем инициализации. При организации пула объектов объекты создаются заранее, так что клиентам не приходится каждый раз ждать их инициализации.

Безопасность на основе ролей

Применение *безопасности на основе ролей* позволяет определять роли декларативно и указывать методы и компоненты, которые могут быть использованы этими ролями. Системный администратор назначает эти роли пользователям или группам пользователей. Программе нет необходимости иметь дело со списками контроля доступа; вместо этого могут применяться роли, представленные с помощью простых строк.

Компоненты с очередями

Компоненты с очередями — это уровень абстракции для организации очередей сообщений. Вместо отправки сообщения в очередь сообщений клиент может вызывать методы объекта-писателя, предоставляющего те же методы, что и класс .NET, сконфигурированный в Enterprise Services. Объект-писатель, в свою очередь, создает сообщения, которые передаются через очередь сообщений серверному приложению.

Компоненты с очередями и очереди сообщений удобны, если клиентское приложение запускается в автономной среде (например, на портативном компьютере, который не всегда подключен к серверу), или если запрос, отправляемый серверу, должен быть кэширован перед тем, как будет передан другому серверу (например, серверу компании-партнера).

Слабо связанные события

В главе 8 объяснялась модель событий .NET, а в главе 26 рассматривалось использование событий в среде COM. В обоих этих механизмах сообщений клиент и сервер поддерживают между собой тесное соединение. Это отличается от модели *слабо связанных событий* (loosely coupled events — LCE). В модели LCE средство COM+ вставляется между клиентом и сервером (рис. 51.3). Издатель регистрирует в COM+ события, которые он будет посылать, определяя класс событий. Вместо отправки событий непосредственно клиенту, издатель посылает события классу событий, зарегистрированному в службе LCE. Служба LCE переадресует их подписчику, которым является клиентское приложение, зарегистрированное и подписанное на событие.

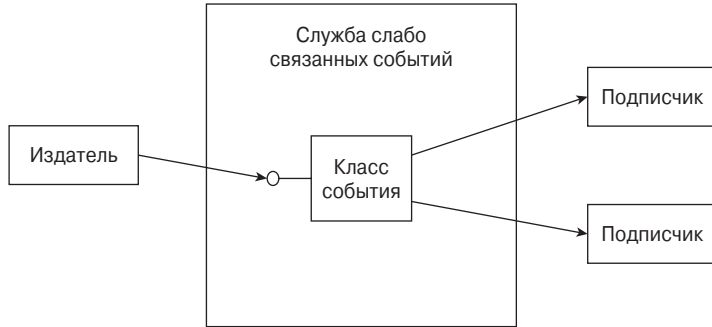


Рис. 51.3. Модель LCE

Создание простого приложения COM+

Для создания класса .NET, который может быть сконфигурирован для Enterprise Services, необходимо сослаться на сборку `System.EnterpriseServices` и добавить оператор `using` для пространства имен `System.EnterpriseServices`. Наиболее важным используемым классом является `ServiceComponent`.

Первый пример демонстрирует базовые требования для создания обслуживаемых компонентов. Вы начинаете с создания приложения — библиотеки C#. Все приложения COM+ должны быть написаны в виде библиотечного приложения, независимо от того, запускаются они в своем собственном процессе или в процессе клиента. Назовите библиотеку `SimpleServer`. Сошлитесь на сборку `System.EnterpriseServices` и добавьте объявление `using System.EnterpriseServices`; в файлы `assemblyinfo.cs` и `class1.cs`.

В проекте библиотеке должно быть назначено строгое имя. Для некоторых средств Enterprise Services также необходимо установить сборку в GAC (global assembly cache — глобальный кэш сборок). Строгие имена и глобальный кэш сборок обсуждались в главе 18.

Класс `ServiceComponent`

Каждый класс обслуживаемого компонента должен наследоваться от `ServiceComponent`. Сам `ServiceComponent` унаследован от класса `ContextBoundObject`, поэтому экземпляр привязан контексту .NET Remoting.

Класс `ServiceComponent` имеет ряд защищенных методов, которые могут быть переопределены (табл. 51.1).

Таблица 51.1. Защищенные методы `ServiceComponent`

| Защищенный метод | Описание |
|----------------------------|---|
| <code>Activate()</code> | Методы <code>Activate()</code> и <code>Deactivate()</code> вызываются, если объект сконфигурирован для использования в организации пула объектов. Когда объект берется из пула, вызывается метод <code>Activate()</code> . Перед тем, как объект возвращается в пул, вызывается метод <code>Deactivate()</code> . |
| <code>CanBePooled()</code> | Это еще один метод для организации пула объектов. Если объект находится в несогласованном состоянии, в переопределенной реализации <code>CanBePooled()</code> можно вернуть значение <code>false</code> . В этом случае объект не будет помещен обратно в пул, а вместо этого уничтожен. Для пула создается новый объект. |
| <code>Construct()</code> | Этот метод вызывается во время создания экземпляра и ему может быть передана конструирующая строка. Конструирующая строка может быть модифицирована системным администратором. Позднее в этой главе эта строка будет использоваться для определения подключения к базе данных. |

Атрибуты сборок

Для сборок также необходимы некоторые атрибуты Enterprise Services. Атрибут `ApplicationName` определяет имя приложения, как оно выглядит в Component Services Explorer. Значение атрибута `Description` появляется в виде описания внутри инструмента конфигурирования приложения.

`ApplicationActivation` позволяет определять, что приложение должно быть сконфигурировано как библиотечное приложение или серверное приложение — соответственно, с помощью значений `ActivationOption.Library` или `ActivationOption.Server`. Библиотечное приложение загружается в процесс клиента. В этом случае клиентом может быть исполняющая среда ASP.NET. Серверное приложение приводит к запуску специального процесса приложения. Имя процесса выглядит как `dllhost.exe`. С помощью атрибута `ApplicationAccessControl` можно отключить защиту, чтобы любой пользователь мог использовать компонент.

Переименуйте `Class1.cs` в `SimpleClient.cs` и добавьте следующие атрибуты вне объявления пространства имен:

```
[assembly: ApplicationName("Wrox EnterpriseDemo")]
[assembly: Description("Wrox Sample Application for Professional C#")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]
```

В табл. 51.2 перечислены наиболее важные атрибуты сборки, которые могут быть определены в приложениях Enterprise Services.

Таблица 51.2. Атрибуты приложений Enterprise Service

| Атрибут | Описание |
|---|--|
| <code>[ApplicationName]</code> | Атрибут <code>[ApplicationName]</code> определяет имя приложения COM+, которое отображается в Component Services Explorer после конфигурирования компонента. |
| <code>[ApplicationActivation]</code> | Атрибут <code>[ApplicationActivation]</code> определяет, что приложение должно запускаться как библиотека, или же для него должен быть запущен отдельный процесс. Опции конфигурирования определены в перечислении <code>ActivationOption</code> . Значение <code>ActivationOption.Library</code> определяет запуск приложения в клиентском процессе, а <code>ActivationOption.Server</code> обеспечивает запуск для него собственного процесса <code>dllhost.exe</code> . |
| <code>[ApplicationAccessControl]</code> | Атрибут <code>[ApplicationAccessControl]</code> определяет конфигурацию безопасности для приложения. Используя булевское значение, вы можете включить или отключить контроль доступа. Свойством <code>Authentication</code> можно устанавливать уровень приватности — должен клиент аутентифицироваться при каждом вызове метода или только при установке соединения, — а также необходимость шифрования данных. |

Создание компонента

В файле `SimpleComponent.cs` можно создать класс обслуживаемого компонента. С такими компонентами лучше всего определять интерфейсы, используемые в качестве контракта между клиентом и компонентом. Это не строгое требование, но некоторые из средств Enterprise Services (такие как настройка безопасности на основе ролей на уровне метода или интерфейса) требуют определения интерфейсов. Создайте интерфейс

IGreeting с методом Welcome(). Атрибут [ComVisible] требуется для классов обслуживаемых компонентов и интерфейсов, которые могут быть доступны из средств Enterprise Services.

```
using System.Runtime.InteropServices;
namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IGreeting
    {
        string Welcome(string name);
    }
}
```

Фрагмент кода SimpleServer\IGreeting.cs

Класс SimpleComponent унаследован от базового класса ServicedComponent и реализует интерфейс IGreeting. Класс ServicedComponent служит базовым классом для всех классов обслуживаемых компонентов и предоставляет некоторые методы для фаз активации и конструирования. Применение атрибута [EventTrackingEnabled] к этому классу обеспечивает возможность проведения мониторинга объектов с помощью Component Services Explorer. По умолчанию мониторинг отключен, поскольку применение этого средства снижает производительность. В атрибуте [Description] указан только текст, который появляется в Explorer.

```
using System.EnterpriseServices;
using System.Runtime.InteropServices;
namespace Wrox.ProCSharp.EnterpriseServices
{
    [EventTrackingEnabled(true)]
    [ComVisible(true)]
    [Description("Simple Serviced Component Sample")]
    public class SimpleComponent: ServicedComponent, IGreeting
    {
        public SimpleComponent()
        {
        }
    }
}
```

Фрагмент кода SimpleServer\SimpleComponent.cs

Метод Welcome() возвращает только строку "Hello, " с именем, переданным в аргументе. Чтобы можно было наблюдать некоторый видимый результат в Component Services Explorer, пока компонент работает, Thread.Sleep() эмулирует задержку, вызванную обработкой:

```
public string Welcome(string name)
{
    // Эмуляция обработки
    System.Threading.Thread.Sleep(1000);
    return "Hello, " + name;
}
}
```

Помимо применения некоторых атрибутов и наследования класса от ServicedComponent, никаких специальных требований к классам, использующим средства Enterprise Services, не предъявляется. Все, что остается сделать — это построить и развернуть клиентское приложение.

В первом примере компонента установлен атрибут [EventTrackingEnabled]. Некоторые наиболее часто используемые атрибуты, зависящие от конфигурации обслуживаемых компонентов, описаны в табл. 51.3.

Таблица 51.3. Атрибуты классов компонентов

| Атрибут | Описание |
|------------------------|--|
| [EventTrackingEnabled] | Установка атрибута [EventTrackingEnabled] позволяет организовать мониторинг компонента в Component Services Explorer. Установка этого атрибута в true влечет за собой некоторые накладные расходы, вот почему по умолчанию отслеживание событий отключено. |
| [JustInTimeActivation] | С помощью этого атрибута компонент может быть сконфигурирован так, чтобы не активизироваться, когда вызывающий код создает экземпляр класса, а вместо этого активизироваться при первом вызове метода. Также посредством этого атрибута компонент может деактивизировать себя. |
| [ObjectPooling] | Если время инициализации компонента достаточно длительное по сравнению со временем вызова метода, с помощью атрибута [ObjectPooling] можно сконфигурировать пул объектов. |
| [Transaction] | Атрибут [Transaction] определяет транзакционные характеристики компонента. Здесь компонент определяет, обязательна ли транзакция, и поддерживается она или нет. |

Развертывание

Сборки с обслуживаемыми компонентами должны быть сконфигурированы для COM+. Это может быть сделано автоматически или посредством ручной регистрации сборки.

Автоматическое развертывание

Если запущено клиентское приложение .NET, использующее обслуживаемый компонент, то приложение COM+ конфигурируется автоматически. Это верно для всех классов, унаследованных от `ServicedComponent`. Атрибуты приложения и класса, такие как [EventTrackingEnabled], определяют характеристики конфигурации.

Автоматическое развертывание обладает одним важным недостатком. Чтобы оно работало, клиентское приложение нуждается в административных привилегиях. Если клиентское приложение, вызывающее обслуживаемый компонент, относится к типу ASP.NET, то исполняющая среда ASP.NET обычно не имеет административных прав. С учетом этого недостатка автоматическое развертывание удобно применять только в период разработки. Однако во время разработки это средство чрезвычайно удобно, поскольку избавляет от необходимости вручную развертывать приложение после каждого его построения.

Ручное развертывание

Сборку можно развернуть и вручную с помощью утилиты командной строки — инструмента установки .NET под названием `regsvcs.exe`. Запуск следующей команды:

```
regsvcs SimpleServer.dll
```

регистрирует сборку `SimpleServer` в качестве приложения COM+ и конфигурирует включенные в нее компоненты согласно их атрибутам; при этом также создается библиотека типов, которая может использоваться клиентами COM, обращающимися к компоненте .NET.

После конфигурирования сборки запустите Component Service Explorer, выбрав в меню Start (Пуск) пункт Administrative Tools⇒Component Services (Администрирование⇒Службы компонентов). В древовидном представлении слева выберите элемент Component

Services⇒Computers⇒My Computer⇒COM+ Applications (Службы компонентов⇒Компьютеры⇒Мой компьютер⇒Приложения COM+) и удостоверьтесь, что приложение успешно сконфигурировано.



В Windows Vista для получения доступа к Component Services Explorer потребуется запустить консоль управления Microsoft (MMC) и добавить в ней оснастку Component Services (Службы компонентов).

Создание установочного пакета

В Component Service Explorer можно создавать установочные пакеты для серверных или клиентских систем. Установочный пакет для сервера включает сборки и конфигурационные настройки для установки приложения на другом сервере. Если обслуживаемый компонент вызывается из приложения, запущенного на других системах, то на клиентской системе должен быть установлен прокси. Установочный пакет для клиента включает сборки и конфигурацию прокси.

Чтобы создать установочный пакет, можно запустить Component Service Explorer, выбрать приложение COM+ и выбрать в меню пункт Action⇒Export (Действие⇒Экспорт). Откроется окно мастера экспорта приложений COM+ (COM+ Application Export Wizard), показанное на рис. 51.4. На этом экране можно выбрать экспорт либо как серверного приложения (Server application), либо как прокси приложения (Application proxy). Выбрав опцию Server application, можно также сконфигурировать экспорт пользовательских идентичностей с ролями. Эта опция должна выбираться только в том случае, если целевая система находится в том же домене, что и система, на которой создается пакет, поскольку сконфигурированные идентичности пользователей помещаются в установочный пакет. При выборе опции Application proxy создается установочный пакет для клиентской системы.

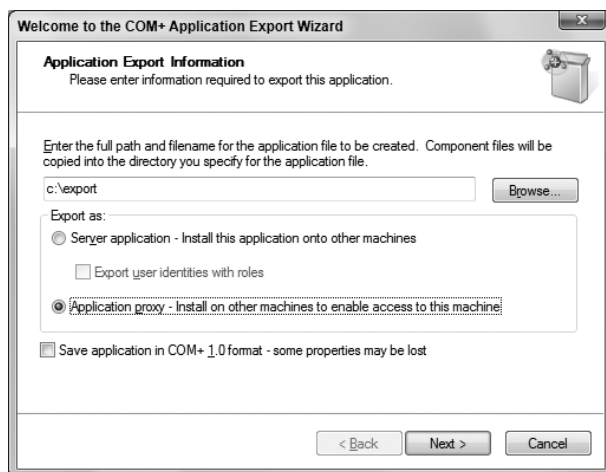


Рис. 51.4. Мастер генерации установочных пакетов



Опция создания прокси приложения не доступна, если приложение сконфигурировано как библиотечное.

Чтобы установить прокси, потребуется просто запустить программу `setup.exe` из установочного пакета. Имейте в виду, что прокси приложения не может устанавливаться на той же системе, где установлено само приложение. После установки прокси приложения в Component Services Explorer появляется соответствующий элемент, который его представляет. Для прокси приложения единственной опцией, которая может быть сконфигурирована, является имя сервера на вкладке **Activation** (Активизация), о чем будет сказано в следующем разделе.

Component Services Explorer

После успешного конфигурирования имя приложения `EnterpriseDemo` можно видеть в древовидном представлении Component Services Explorer. Это имя устанавливается атрибутом `[ApplicationName]`. Выбор в меню пункта **Action** ⇒ **Properties** (Действие ⇒ Свойства) приводит к открытию диалогового окна, показанного на рис. 51.5. Имя и описание сконфигурированы с помощью атрибутов. Перейдя на вкладку **Activation** (Активизация), вы увидите, что приложение сконфигурировано как серверное, поскольку так определено в атрибуте `[ApplicationActivation]`, а на вкладке **Security** (Безопасность) имеется отмеченный флажок **Enforce access checks for this application** (Применить проверки доступа для этого приложения), поскольку атрибут `[ApplicationAccessControl]` был установлен в `false`.

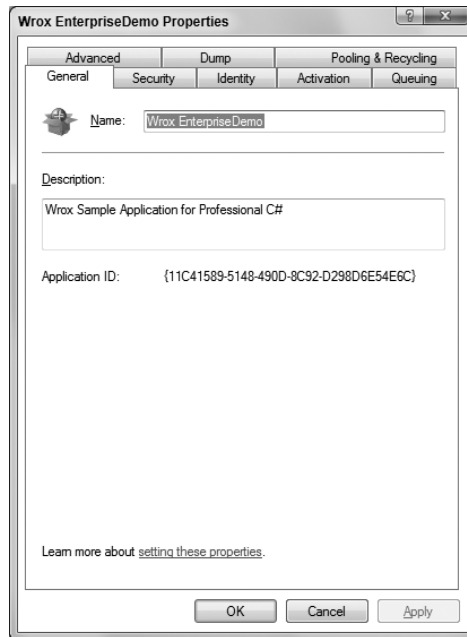


Рис. 51.5. Диалоговое окно свойств приложения

Ниже приведен список дополнительных опций, которые могут быть установлены в этом приложении.

- **Security** (Безопасность). На вкладке **Security** можно включать или отключать проверки доступа. Если защита включена, проверки доступа могут быть установлены на уровне приложения, компонента, интерфейса и метода. Также возможно шифровать сообщения, пересылаемые по сети, посредством приватности пакетов в каче-

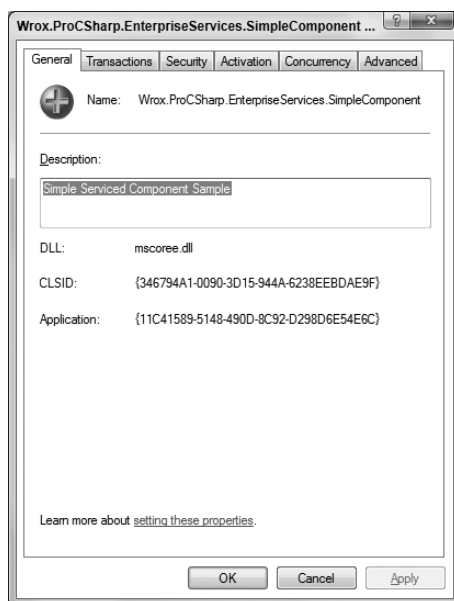
стве уровня аутентификации для вызовов. Разумеется, это увеличивает накладные расходы.

- **Identity** (Идентичность). В серверных приложениях вкладку **Identity** можно использовать для конфигурирования пользовательской учетной записи, которая будет применяться для процесса, в котором размещается приложение. По умолчанию это будет интерактивный пользователь. Такая настройка очень полезна при отладке приложения, но не может использоваться в рабочей системе, если приложение запускается на сервере, поскольку на него не может входить кто угодно. Прежде чем устанавливать приложение в рабочей системе, его необходимо протестировать с применением специфического пользователя для этого приложения.
- **Activation** (Активизация). Вкладка **Activation** позволяет конфигурировать приложение как библиотеку, либо как серверное приложение. В COM+ 1.5 есть две новые опции — возможность запускать приложение в виде службы Windows (Windows Service) и применять SOAP для доступа к приложению. Службы Windows обсуждаются в главе 25. В случае выбора опции SOAP для доступа к компоненту используется механизм .NET Remoting, сконфигурированный внутри сервера IIS. Вместо применения .NET Remoting далее в этой главе обращение к компоненту будет осуществляться с использованием WCF. Технология WCF рассматривается в главе 43.
- **Queuing** (Очереди). Конфигурация **Queuing** необходима для компонентов службы, использующих очереди сообщений.
- **Advanced** (Дополнительно). На этой вкладке можно указать, что приложение должно быть остановлено после определенного периода бездействия пользователя. Допускается также заблокировать определенную конфигурацию, чтобы ее невозможно было непреднамеренно изменить.
- **Dump** (Дамп). Если приложение потерпит крах, здесь можно указать каталог, куда должен быть записан дамп. Это удобно для компонентов, написанных на C++.
- **Pooling and Recycling** (Организация пула и повторное использование). Организация пула и повторное использование — новая опция в версии COM+ 1.5. С помощью этой опции можно конфигурировать необходимость перезапуска (повторного использования) приложения в зависимости от времени жизни, потребностей в памяти, количества вызовов и т.д.

В Component Services Explorer можно также просматривать и конфигурировать сам компонент. Среди дочерних элементов приложения имеется компонент `Wrox.ProCSharp.EnterpriseServices.SimpleComponent`. После выбора в меню пункта **Action⇒Properties** (Действие⇒Свойства) откроется диалоговое окно, показанное на рис. 51.6.

Ниже перечислены опции, доступные для конфигурирования в этом диалоговом окне.

- **Transactions** (Транзакции). На вкладке **Transactions** можно указать, требует ли компонент транзакции. Это средство применяется в следующем примере.
- **Security** (Безопасность). Если для приложения включена защита, можно определить, каким ролям разрешено использовать компонент с этой конфигурацией.
- **Activation** (Активизация). Вкладка **Activation** позволяет настроить организацию пула объектов и присвоить конструирующую строку.
- **Concurrency** (Параллелизм). Если компонент не является безопасным в отношении потоков, настройки параллелизма могут быть установлены в **Required** (Требуется) или **Requires New** (Требуется новый). В результате исполняющая среда COM+ позволит только одному потоку в единицу времени обращаться к компоненту.



*Рис. 51.6. Диалоговое окно свойств компонента
Wrox.ProCSharp.EnterpriseServices.SimpleComponent*

Клиентское приложение

После построения библиотеки обслуживаемого компонента можно приступить к созданию клиентского приложения. Это может быть простым консольным приложением C#. После создания проекта необходимо добавить ссылку на сборку из обслуживаемого компонента, SimpleServer, и на сборку System.EnterpriseServices. Затем можно писать код создания нового экземпляра SimpleComponent и вызывать метод Welcome(). В следующем коде метод Welcome() вызывается 10 раз. Оператор using помогает освобождать ресурсы, выделенные для экземпляра, прежде чем сборщик мусора выполнит свою работу. При использовании оператора using метод Dispose() обслуживаемого компонента вызывается по завершении области действия using.

```

using System;
namespace Wrox.ProCSharp.EnterpriseServices
{
    class Program
    {
        static void Main()
        {
            using (SimpleComponent obj = new SimpleComponent())
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine(obj.Welcome("Stephanie"));
                }
            }
        }
    }
}

```

Фрагмент кода ClientApplication\Program.cs

Если вы запустите клиентское приложение до того, как сконфигурируете сервер, то сервер будет сконфигурирован автоматически. Автоматическое конфигурирование сервера осуществляется со значениями, заданными с помощью атрибутов. Для проверки можете отменить регистрацию обслуживаемого компонента и снова запустить приложение. Если обслуживаемый компонент сконфигурирован при старте клиентского приложения, то запуск потребует больше времени. Помните, что это средство полезно только в период разработки. Для автоматического развертывания также необходимы административные права. Если вы запускаете приложение из среды Visual Studio, это значит, что и Visual Studio потребуются запустить с административными правами.

Пока приложение работает, можно наблюдать за обслуживаемыми компонентами в Component Services Explorer. Выбрав **Components** (Компоненты) в древовидном представлении и выбрав в меню пункт **View**⇒**Detail** (Вид⇒Подробно), можно увидеть количество созданных экземпляров объектов, если был установлен атрибут `[EventTrackingEnabled]`.

Транзакции

Атомарные транзакции являются чаще всего используемым средством Enterprise Services. Технология Enterprise Services позволяет пометчать компоненты как требующие транзакции, и транзакция будет создана исполняющей средой COM+. Все поддерживающие транзакции объекты внутри компонента, такие как соединения ADO.NET, выполняются внутри транзакции.



За дополнительными сведениями о транзакциях обращайтесь в главу 23.

Атрибуты транзакции

Обслуживаемые компоненты могут помечаться атрибутом `[Transaction]` для определения того, нужна ли транзакция компоненту, и каким образом.

На рис. 51.7 показано множество компонентов с различными конфигурациями, относящимися к транзакциям. Клиент вызывает компонент А. Поскольку компонент А сконфигурирован со значением `Required` (транзакция требуется) в `[Transaction]` и никакой транзакции не существовало ранее, создается новая транзакция 1.

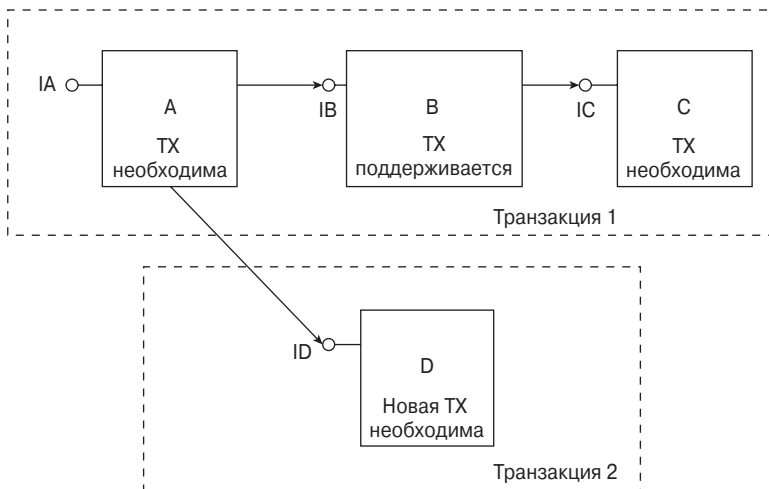


Рис. 51.7. Взаимодействия компонентов с различными установками `[Transaction]`

Компонент А обращается к компоненту В, который, в свою очередь, вызывает компонент С. Поскольку компонент В сконфигурирован со значением `Supported` (транзакция поддерживается), а конфигурация компонента С установлена в `Required`, все три компонента (А, В и С) используют один и тот же контекст транзакции. Если бы компонент В был сконфигурирован с установкой `NotSupported`, то компонент С получил бы новую транзакцию. Компонент D сконфигурирован с установкой `RequiredNew` (требуется новая транзакция), поэтому новая транзакция создается, когда он вызывается компонентом А.

В табл. 51.4 описаны различные значения, которые можно устанавливать с помощью перечисления `TransactionOption`.

Таблица 51.4. Значения перечисления `TransactionOption`

| Значение | Описание |
|---------------------------|---|
| <code>Required</code> | Установка атрибута <code>[Transaction]</code> в <code>TransactionOption</code> . <code>Required</code> означает, что компонент выполняется внутри транзакции. Если транзакция была создана ранее, компонент выполняется в ней. Если же никакой транзакции нет, будет создана новая. |
| <code>RequiredNew</code> | <code>TransactionOption.RequiresNew</code> всегда дает в результате новую транзакцию. Компонент никогда не участвует в той же самой транзакции, что и вызвавший его. |
| <code>Supported</code> | При значении <code>TransactionOption.Supported</code> сам компонент не нуждается в транзакции. Однако транзакция охватывает вызывающий и вызываемый компоненты, если эти компоненты требуют ее наличия. |
| <code>NotSupported</code> | Опция <code>TransactionOption.NotSupported</code> означает, что компонент никогда не выполняется в транзакции, независимо от того, участвует ли в транзакции вызывающий компонент. |
| <code>Disabled</code> | <code>TransactionOption.Disabled</code> означает, что возможная транзакция текущего контекста игнорируется. |

Результаты транзакции

Транзакция может зависеть от установки битов контекста *consistent* (согласован) и *done* (готов). Если бит *consistent* установлен в `true`, компонент удовлетворен исходом транзакции. Транзакция может быть зафиксирована, если все компоненты, участвовавшие в ней, также удовлетворены. Если же бит *consistent* установлен в `false`, значит, компонент не удовлетворен исходом транзакции, и потому транзакция будет отменена, когда завершит работу корневой объект, который ее запустил. Если установлен бит *done*, значит, объект может быть деактивизирован после завершения вызова метода. Новый экземпляр будет создан при следующем вызове метода.

Биты *consistent* и *done* могут устанавливаться четырьмя методами класса `ContextUtil`, приводя к результатам, перечисленным в табл. 51.5.

Таблица 51.5. Значения битов *consistent* и *done* после вызовов методов `ContextUtil`

| Метод | Бит <i>consistent</i> | Бит <i>done</i> |
|----------------------------|-----------------------|--------------------|
| <code>SetComplete</code> | <code>true</code> | <code>true</code> |
| <code>SetAbort</code> | <code>false</code> | <code>true</code> |
| <code>EnableCommit</code> | <code>true</code> | <code>false</code> |
| <code>DisableCommit</code> | <code>false</code> | <code>false</code> |

В .NET также возможно устанавливать биты *consistent* и *done* из-за применения атрибута `[AutoComplete]` к методу вместо вызова методов `ContextUtil`. При таком атрибуте метод `ContextUtil.SetComplete()` будет вызван автоматически, если метод завершился успешно. Если же метод дал сбой, и было сгенерировано исключение, то при атрибуте `[AutoComplete]` будет вызван метод `ContextUtil.SetAbort()`.

Пример приложения

Этот пример приложения эмулирует сценарий, который записывает новые заказы в базу примеров Northwind. Как показано на рис. 51.8, в приложении COM+ присутствует множество компонентов. Класс `OrderControl` вызывается из клиентского приложения для создания новых заказов. `OrderControl` использует компонент `OrderData`. Компонент `OrderData` отвечает за создание новой записи в таблице `Order` базы данных Northwind. Компонент `OrderData` применяет компонент `OrderLineData` для записи вхождений `Order Detail` в базу данных. И `OrderData`, и `OrderLineData` должны участвовать в одной и той же транзакции.

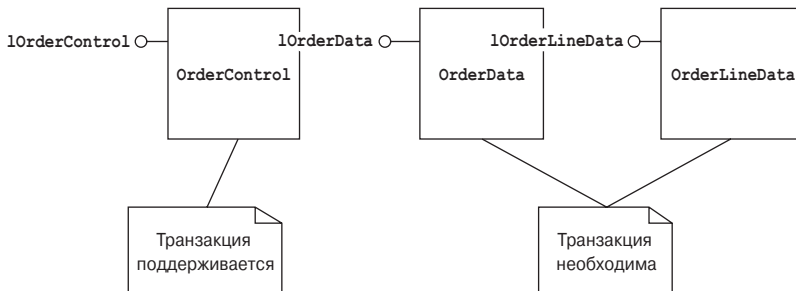


Рис. 51.8. Компоненты примера приложения

Начните с создания приложения C# Component Library по имени `NorthwindComponent`. Подпишите сборку с помощью файла ключей и определите атрибуты приложения Enterprise Services, как показано ниже:

```

[assembly: ApplicationName("Wrox.NorthwindDemo")]
[assembly: ApplicationActivation(ActivationOption.Server)]
[assembly: ApplicationAccessControl(false)]
  
```

Фрагмент кода NorthwindComponents\AssemblyInfo.cs

Сущностные классы

Теперь добавьте сущностные классы `Order` и `OrderLine`, представляющие столбцы таблиц `Order` и `Order Details` базы данных Northwind. Сущностные классы предназначены для хранения данных, являющихся важными для домена приложения, в данном случае — заказов. Класс `Order` имеет статический метод `Create()`, который создает и возвращает новый экземпляр класса `Order`, а также инициализирует этот экземпляр аргументами, переданными методу. Также класс `Order` имеет доступные только для чтения свойства, такие как `OrderId`, `CustomerId`, `OrderData`, `ShipAddress`, `ShipCity` и `ShipCountry`. Значение свойства `OrderId` во время создания объекта класса `Order` не известно, но поскольку таблица `Order` базы данных Northwind имеет атрибут автоинкремента, значение этого свойства становится известным после того, как заказ записан в базу данных.

Метод `SetOrderId()` служит для установки соответствующего идентификатора после того, как заказ сохранен в базе. Поскольку этот метод вызывается классом из той же сборки, уровень доступа к этому методу установлен как `internal`. Метод `AddOrderLine()` добавляет детальную информацию к заказу.

```

using System;
using System.Collections.Generic;
namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class Order
    {
        public static Order Create(string customerId, DateTime orderDate,
                                   string shipAddress, string shipCity, string shipCountry)
        {
            return new Order()
            {
                CustomerId = customerId,
                OrderDate = orderDate,
                ShipAddress = shipAddress,
                ShipCity = shipCity,
                ShipCountry = shipCountry
            }
        }
        public Order()
        {
        }

        internal void SetOrderId(int orderId)
        {
            this.OrderId = orderId;
        }
        public void AddOrderLine(OrderLine orderLine)
        {
            orderLines.Add(orderLine);
        }
        private readonly List<OrderLine> orderLines = new List<OrderLine>();
        public int OrderId { get; private set; }
        public string CustomerId { get; private set; }
        public DateTime OrderDate { get; private set; }
        public string ShipAddress { get; private set; }
        public string ShipCity { get; private set; }
        public string ShipCountry { get; private set; }
        public OrderLine[] OrderLines
        {
            get
            {
                OrderLine[] ol = new OrderLine[orderLines.Count];
                orderLines.CopyTo(ol);
                return ol;
            }
        }
    }
}

```

Фрагмент кода *NorthwindComponents\Orders.cs*

Второй сущностный класс — `OrderLine`. Этот класс, подобно классу `Order`, имеет статический метод `Create()`. Помимо этого класс имеет лишь несколько свойств `productId`, `unitPrice` и `quantity`.

```
using System;
namespace Wrox.ProCSharp.EnterpriseServices
{
    [Serializable]
    public class OrderLine
    {
        public static OrderLine Create(int productId, float unitPrice, int quantity)
        {
            return new OrderLine
            {
                ProductId = productId,
                UnitPrice = unitPrice,
                Quantity = quantity
            };
        }
        public OrderLine()
        {
        }
        public int ProductId { get; set; }
        public float UnitPrice { get; set; }
        public int Quantity { get; set; }
    }
}
```

Фрагмент кода *NorthwindComponents\OrderLine.cs*

Компонент OrderControl

Класс OrderControl представляет простой компонент бизнес-служб. В этом примере в интерфейсе IOrderControl определен только один метод NewOrder(). Реализация NewOrder() не выполняет ничего помимо создания нового экземпляра компонента бизнес-служб OrderData и вызова метода Insert() для записи объекта Order в базу данных. В более сложных сценариях этот метод мог быть расширен для внесения журнальной записи в базу данных или для вызова компонента, работающего с очередью, для отправки объекта Order в очередь сообщений.

```
using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderControl
    {
        void NewOrder(Order order);
    }
    [Transaction(TransactionOption.Supported)]
    [EventTrackingEnabled(true)]
    [ComVisible(true)]
    public class OrderControl: ServicedComponent, IOrderControl
    {
        [AutoComplete()]
        public void NewOrder(Order order)
        {
            using (OrderData data = new OrderData())
            {
                data.Insert(order);
            }
        }
    }
}
```

Фрагмент кода *NorthwindComponents\OrderControl.cs*

Компонент OrderData

Класс OrderData отвечает за запись значений объектов Order в базу данных. Интерфейс IOrderUpdate определяет метод Insert(). Этот интерфейс можно расширить поддержкой метода Update(), который обновляет существующую запись в базе данных.

```

using System;
using System.Data.SqlClient;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderUpdate
    {
        void Insert(Order order);
    }
}

```

Фрагмент кода NorthwindComponents\OrderData.cs

Класс OrderData имеет атрибут [Transaction] со значением TransactionOption.Required. Это значит, что компонент в любом случае будет запущен в рамках транзакции. Транзакция либо создается вызывающим кодом и OrderData использует ее, либо создается новая транзакция. Здесь создается новая транзакция, поскольку вызывающий компонент OrderControl транзакции не имеет.

С обслуживаемыми компонентами можно использовать только конструкторы по умолчанию. С помощью Component Services Explorer легко сконфигурировать строку конструирования, отправляемую компоненту (рис. 51.9).

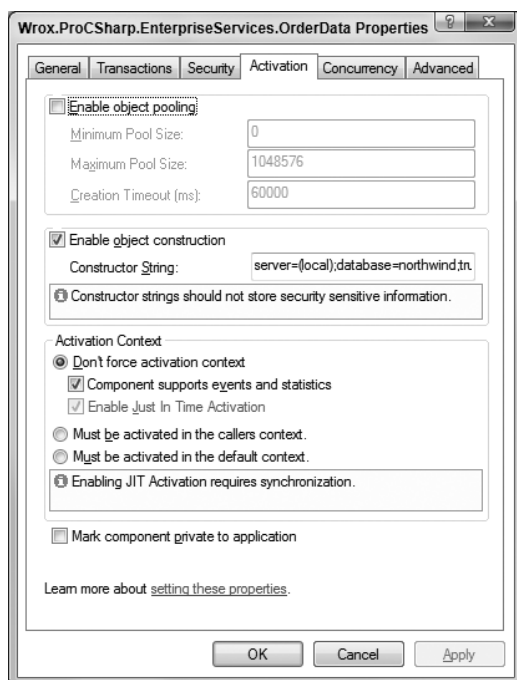


Рис. 51.9. Вкладка Activation конфигурации компонента

На вкладке **Activation** конфигурации компонента можно изменить конструирующую строку. Флажок **Enable object construction** (Разрешить конструирование объекта) отмечен, когда установлен атрибут `[ConstructionEnabled]`, как это было с классом `OrderData`. Свойство `Default` атрибута `[ConstructionEnabled]` определяет строку соединения по умолчанию, показанную в настройках **Activation** после регистрации сборки. Установка этого атрибута также требует перегрузки метода `Construct()` из базового класса `ServicedComponent`. Данный метод вызывается исполняющей средой COM+ при создании экземпляра объекта, при этом строка конструирования передается в качестве аргумента. Строка конструирования устанавливается в переменную `connectionString`, которая позднее используется для подключения к базе данных.

```
[Transaction(TransactionOption.Required)]
[EventTrackingEnabled(true)]
[ConstructionEnabled
    (true, Default="server=(local);database=northwind;trusted_connection=true")]

[ComVisible(true)]
public class OrderData: ServicedComponent, IOrderUpdate
{
    private string connectionString;
    protected override void Construct(string s)
    {
        connectionString = s;
    }
}
```

В ядре компонента находится метод `Insert()`. Здесь для записи объекта `Order` в базу данных применяется ADO.NET. (Технология ADO.NET детально обсуждается в главе 30.) В этом примере создается объект `SqlConnection`, в котором строка соединения, установленная методом `Construct()`, используется для инициализации объекта.

Атрибут `[AutoComplete()]` применен к методу для применения автоматической обработки транзакции, как было описано ранее:

```
[AutoComplete()]
public void Insert(Order order)
{
    var connection = new SqlConnection(connectionString);
```

Метод `connection.CreateCommand()` создает объект `SqlCommand` со свойством `CommandText`, установленным в SQL-оператор `INSERT` для добавления новой записи в таблицу `Orders`. Метод `ExecuteNonQuery()` выполняет SQL-оператор.

```
try
{
    var command = connection.CreateCommand();
    command.CommandText = "INSERT INTO Orders (CustomerId, " +
        "OrderDate, ShipAddress, ShipCity, ShipCountry)" +
        "VALUES(@CustomerId, @OrderDate, @ShipAddress, @ShipCity, " +
        "@ShipCountry)";

    command.Parameters.AddWithValue("@CustomerId", order.CustomerId);
    command.Parameters.AddWithValue("@OrderDate", order.OrderDate);
    command.Parameters.AddWithValue("@ShipAddress", order.ShipAddress);
    command.Parameters.AddWithValue("@ShipCity", order.ShipCity);
    command.Parameters.AddWithValue("@ShipCountry", order.ShipCountry);
    connection.Open();
    command.ExecuteNonQuery();
}
```

Поскольку `OrderId` определен в базе данных как автоинкрементное значение, и этот идентификатор необходим для записи `Order Details` в базу данных, `OrderId` читается из `@@IDENTITY`. Затем вызовом метода `SetOrderId()` он устанавливается в объект `Order`:

```
command.CommandText = "SELECT @@IDENTITY AS 'Identity'";
object identity = command.ExecuteScalar();
order.SetOrderId(Convert.ToInt32(identity));
```

После того, как заказ записан в базу данных, все строки с позициями заказа записываются с использованием компонента `OrderLineData`:

```
using (OrderLineData updateOrderLine = new OrderLineData())
{
    foreach (OrderLine orderLine in order.OrderLines)
    {
        updateOrderLine.Insert(order.OrderId, orderLine);
    }
}
```

И, наконец, независимо от того, успешно был выполнен блок `try` или же возникло исключение, соединение закрывается:

```
finally
{
    connection.Close();
}
}
```

Компонент `OrderLineData`

Компонент `OrderLineData` реализован аналогично компоненту `OrderData`. Строка соединения с базой данных определяется с помощью атрибута `[ConstructionEnabled]`.

```
using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;
using System.Data;
using System.Data.SqlClient;
namespace Wrox.ProCSharp.EnterpriseServices
{
    [ComVisible(true)]
    public interface IOrderLineUpdate
    {
        void Insert(int orderId, OrderLine orderDetail);
    }
    [Transaction(TransactionOption.Required)]
    [EventTrackingEnabled(true)]
    [ConstructionEnabled
        (true, Default="server=(local);database=northwind;trusted_
connection=true")]
    [ComVisible(true)]
    public class OrderLineData: ServicedComponent, IOrderLineUpdate
    {
        private string connectionString;
        protected override void Construct(string s)
        {
            connectionString = s;
        }
    }
}
```

Фрагмент кода `NorthwindComponents\OrderLineData.cs`

В этом примере атрибут `[AutoComplete]` не используется с методом `Insert()` класса `OrderLineData`, чтобы продемонстрировать другой способ определения исхода тран-

закции. Этот пример показывает, как установить биты *consistent* и *done* с помощью класса `ContextUtil`. Метод `SetComplete()` вызывается в конце метода, в зависимости от того, удалась ли вставка данных в базу. Если при этом произошла ошибка, то метод `SetAbort()` устанавливает бит *consistent* в `false`, так что транзакция отменяется вместе со всеми изменениями компонентов, принимавших в ней участие.


```
public void Insert(int orderId, OrderLine orderDetail)
{
    var connection = new SqlConnection(connectionString);
    try
    {
        var command = connection.CreateCommand();
        command.CommandText = "INSERT INTO [Order Details] (OrderId, " +
            "ProductId, UnitPrice, Quantity)" +
            "VALUES (@OrderId, @ProductId, @UnitPrice, @Quantity)";
        command.Parameters.AddWithValue("@OrderId", orderId);
        command.Parameters.AddWithValue("@ProductId", orderDetail.ProductId);
        command.Parameters.AddWithValue("@UnitPrice", orderDetail.UnitPrice);
        command.Parameters.AddWithValue("@Quantity", orderDetail.Quantity);
        connection.Open();
        command.ExecuteNonQuery();
    }
    catch (Exception)
    {
        ContextUtil.SetAbort();
        throw;
    }
    finally
    {
        connection.Close();
    }
    ContextUtil.SetComplete();
}
}
```

Клиентское приложение

Имея построенный компонент, можно перейти к созданию клиентского приложения. Для целей тестирования вполне подойдет консольное приложение. После ссылки на сборки `NorthwindComponent` и `System.EnterpriseServices` можно создать новый заказ с помощью статического метода `Order.Create()`. Вызов `order.AddOrderLine()` добавляет строку к заказу. `OrderLine.Create()` принимает идентификатор продукта, цену и количество, чтобы создать новую строку заказа. В реальное приложение было бы полезно добавить класс `Product` вместо использования идентификатора продукта, но целью настоящего примера является демонстрация транзакции в целом.

И, наконец, создается класс обслуживаемого компонента `OrderControl` для вызова метода `NewOrder()`:

```

 var order = Order.Create("PICCO", DateTime.Today, "Georg Pippas",
                           "Salzburg", "Austria");
order.AddOrderLine(OrderLine.Create(16, 17.45F, 2));
order.AddOrderLine(OrderLine.Create(67, 14, 1));
using (var orderControl = new OrderControl())
{
    orderControl.NewOrder(order);
}
```

Фрагмент кода *ClientApplication\Program.cs*

Вы можете попытаться записать в `OrderLine` несуществующий продукт (используя идентификатор продукта, которого нет в таблице `Products`). В этом случае транзакция будет отменена, и никакие данные в базу не запишутся.

Пока транзакция активна, вы можете видеть ее в `Component Services Explorer`, выбрав `Distributed Transaction Coordinator` (Координатор распределенных транзакций) в древовидном представлении (рис. 51.10).



Чтобы увидеть активную транзакцию, понадобится добавить время ожидания к методу `Insert()` класса `OrderData`; в противном случае транзакция слишком быстро завершится, чтобы ее можно было заметить.

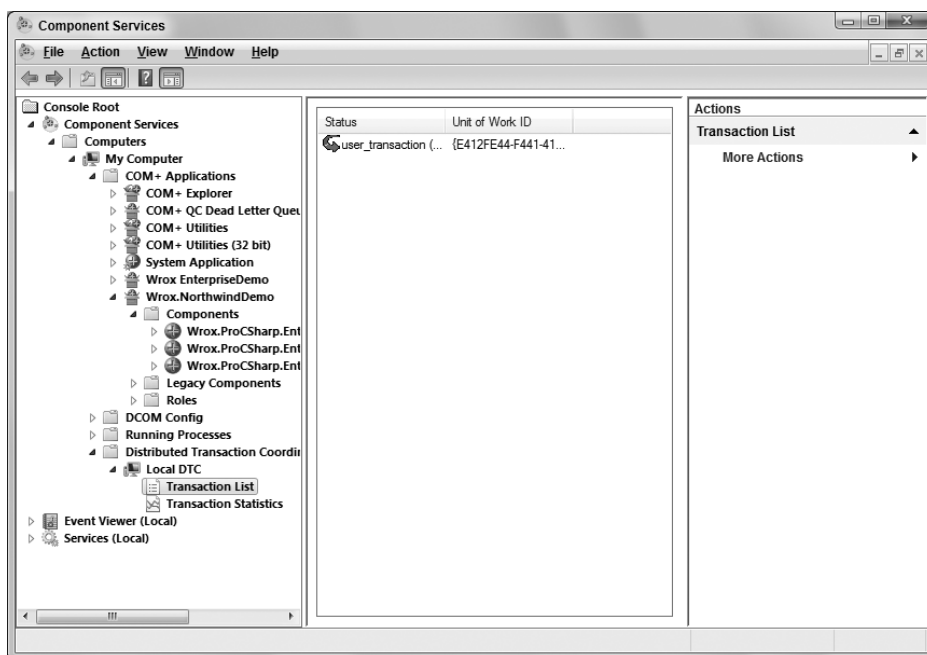


Рис. 51.10. Координатор распределенных транзакций



При отладке обслуживаемого компонента внутри транзакции имейте в виду, что таймаут транзакции по умолчанию для обслуживаемых компонентов составляет 60 секунд. Вы можете изменить значение по умолчанию для всей системы в `Component Services Explorer`, щелкнув на `My Computer` (Мой компьютер), выбрав пункт `Action` → `Properties` (Действие → Свойства) и открыв вкладку `Options` (Параметры). Вместо изменения значения для всей системы таймаут транзакции также может быть сконфигурирован на уровне компонентов с помощью опций `Transaction` (Транзакция) конкретного компонента.

Интеграция WCF и Enterprise Services

Windows Communication Foundation (WCF) — это предпочтительная технология коммуникаций для .NET Framework 4. Она детально рассматривается в главе 43. Службы .NET Enterprise Services предлагают великолепную модель интеграции с WCF.

WCF-фасад службы

Добавление фасада (façade) WCF к приложению Enterprise Services позволяет применять клиенты WCF для доступа к обслуживаемым компонентам. Вместо протокола DCOM с помощью WCF можно использовать разные протоколы, такие как HTTP с SOAP или TCP с двоичным форматированием.

Чтобы создать фасад WCF в Visual Studio 2010, выберите пункт Tools⇒WCF SvcConfigEditor (Сервис⇒WCF SvcConfigEditor). После запуска этого инструмента выберите в меню File⇒Integrate⇒COM+ Application (Файл⇒Интеграция⇒Приложение COM+). Затем раскройте узел приложения COM+ Wrox.NorthwindDemo, узел компонента Wrox.ProCSharp.EnterpriseServices.OrderControl и выберите интерфейс IOrderControl, как показано на рис. 51.11.



Вместо Visual Studio для создания фасада WCF можно использовать утилиту командной строки `comsvcsconfig.exe`. Эта утилита находится в каталоге `<Windows>\Microsoft.NET\Framework\v4.0`

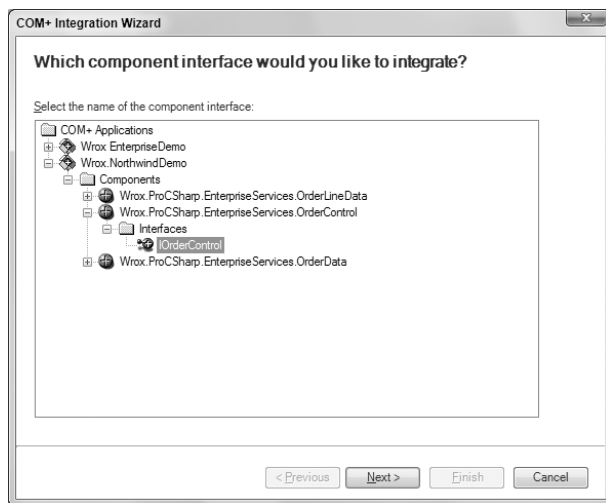


Рис. 51.11. Мастер интеграции COM+

После щелчка на кнопке **Next** (Далее) откроется экран, на котором можно выбрать методы интерфейса `IOrderControl`, которые должны быть доступны клиентам WCF. Для интерфейса `IOrderControl` отображается только один метод — `NewOrder()`.

Следующее окно мастера, показанное на рис. 51.12, позволяет конфигурировать опции хостинга. Здесь можно указать, в каком процессе должна запускаться служба WCF. Когда выбирается опция **COM+ hosted** (Хостинг COM+), фасад WCF запускается внутри процесса `COM+ dllhost.exe`. Эта опция возможна, только если приложение сконфигурировано как серверное: `[ApplicationActivation(ActivationOption.Server)]`.

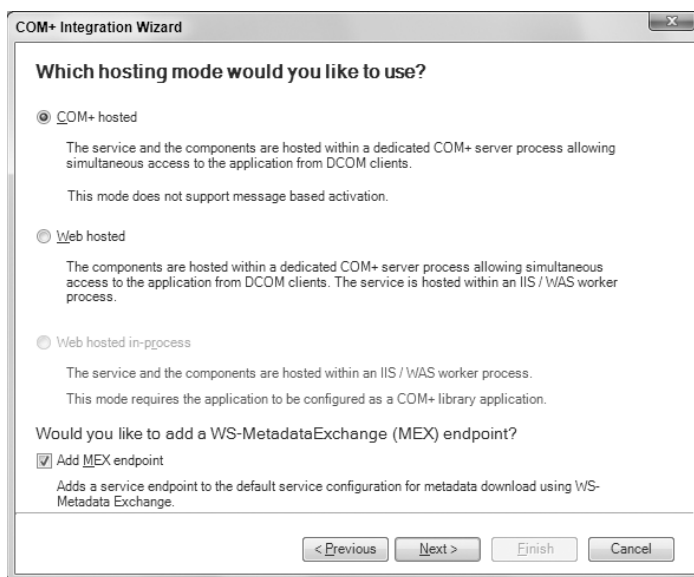


Рис. 51.12. Конфигурирование опций хостинга

Опция **Web hosted** (Веб-хостинг) специфицирует, что канал WCF прослушивается внутри процесса IIS или рабочего процесса WAS (Windows Activation Services — Службы активации Windows). Средство WAS доступно, начиная с Windows Vista и Windows Server 2008. Выбор **Web hosted in-process** (Внутрипроцессный веб-хостинг) означает, что библиотека компонента Enterprise Services запускается внутри IIS или рабочего процесса WAS. Такая конфигурация возможна, только если приложение сконфигурировано как библиотека: [ApplicationActivation(ActivationOption.Library)].

Отметка флажка **Add MEX endpoint** (Добавить конечную точку MEX) приводит к добавлению конечной точки MEX (Metadata Exchange) в конфигурационный файл WCF, чтобы программист на стороне клиента имел доступ к метаданным службы, используя WS-Metadata Exchange.



Конечные точки MEX объясняются в главе 43.

На следующем экране мастера, показанном на рис. 51.13, можно указать режим коммуникации для доступа к фасаду WCF. В зависимости от существующих требований, если клиент обращается к службе через брандмауэр или когда требуется независимые от платформы коммуникации, оптимальным выбором будет протокол HTTP. Протокол TCP обеспечивает более быстрые коммуникации между машинами для клиентов .NET, а протокол Named Pipes (Именованные каналы) — самый быстрый вариант, если клиентское приложение работает на той же машине, что и служба.

На следующем экране мастера (рис. 51.14) вводится информация о базовом адресе службы, который зависит от выбранного коммуникационного протокола.

На последнем экране мастера отображается местоположение конфигурации конечной точки. Базовым каталогом для конфигураций является <Program Files>\ComPlus Applications, за которым следует уникальный идентификатор приложения. В этом каталоге можно найти файл application.config. Этот конфигурационный файл описывает поведение и конечные точки WCF.

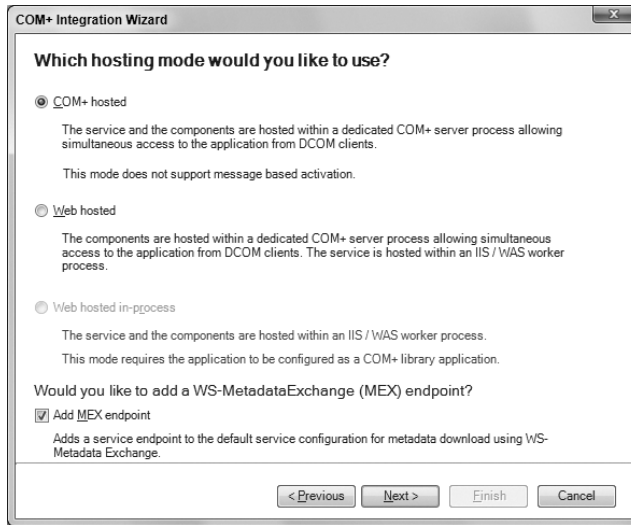


Рис. 51.13. Установка режима коммуникации для доступа к фасаду WCF

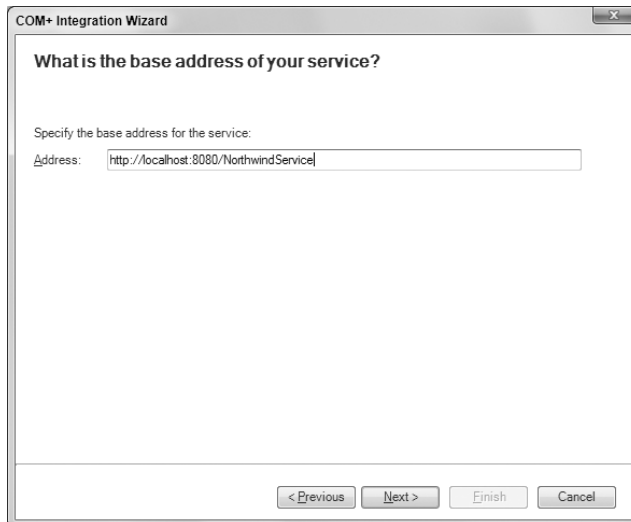


Рис. 51.14. Установка базового адреса службы

Элемент `<service>` позволяет указать представленную службу WCF с конфигурацией конечной точки. Привязка устанавливается в `wsHttpBinding` с конфигурацией `comTransactionalBinding`, так что транзакция может исходить от вызывающего кода к обслуживаемому компоненту. При других требованиях к сети и клиенту можно специфицировать другую привязку, но это все описано в главе 43.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
```

```

    <behavior name="ComServiceMexBehavior">
      <serviceMetadata httpGetEnabled="false" />
      <serviceDebug />
    </behavior>
  </serviceBehaviors>
</behaviors>
<bindings>
  <netNamedPipeBinding>
    <binding name="comNonTransactionalBinding" />
    <binding name="comTransactionalBinding" transactionFlow="true" />
  </netNamedPipeBinding>
  <wsHttpBinding>
    <binding name="comNonTransactionalBinding" />
    <binding name="comTransactionalBinding" transactionFlow="true" />
  </wsHttpBinding>
</bindings>
<comContracts>
  <comContract contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}"
    name="IOrderControl"
    namespace=
      "http://tempuri.org/E1B02E09-EE48-3B6B-946F-E6A8BAEC6340"
    requiresSession="true">
    <exposedMethods>
      <add exposedMethod="NewOrder" />
    </exposedMethods>
  </comContract>
</comContracts>
<services>
  <service behaviorConfiguration="ComServiceMexBehavior"
    name="{196F39D0 - 4F47 - 454A - BC16 - 955C2C54B6F5},
      {A16C0740-C2A0-38C9-9FD3-7C583B3B42FA}">
    <endpoint address="IOrderControl" binding="wsHttpBinding"
      bindingConfiguration="comTransactionalBinding"
      contract="{E1B02E09-EE48-3B6B-946F-E6A8BAEC6340}" />
    <endpoint address="mex" binding="mexHttpBinding"
      contract="IMetadataExchange" />
    <host>
      <baseAddresses>
        <add baseAddress=
          "net.pipe://localhost/Wrox.NorthwindDemo/Wrox.ProCSharp.
EnterpriseServices.OrderControl" />
        <add baseAddress=
          "http://localhost:8088/NorthwindService" />
      </baseAddresses>
    </host>
  </service>
</services>
</system.serviceModel>
</configuration>

```

Прежде запускать серверное приложение, необходимо изменить настройки безопасности, чтобы позволить пользователю, запустившему приложение, регистрировать порты для прослушивания. В противном случае нормальный пользователь не сможет зарегистрировать порт прослушивания. В Windows 7 это можно сделать командой `netsh`, как показано ниже. Опция `http` изменяет ACL для протокола HTTP. Номер порта и наименование службы определяются в URL, а в опции `user` указывается имя пользователя, который запускает службу прослушивания.

```
netsh http add urlacl url=http://+:8088/NorthwindService user=username
```

Клиентское приложение

Создайте новое консольное приложение по имени `WCFClientApp`. Поскольку служба предоставляет конечную точку MEX, можете добавить ссылку из Visual Studio, выбрав в меню **Project** → **Add Service Reference** (Проект → Добавить ссылку на службу), как показано на рис. 51.15.



Если служба развернута в хосте COM+, для получения доступа к данным MEX потребуется запустить приложение. Если служба развернута внутри WAS, приложение стартует автоматически.

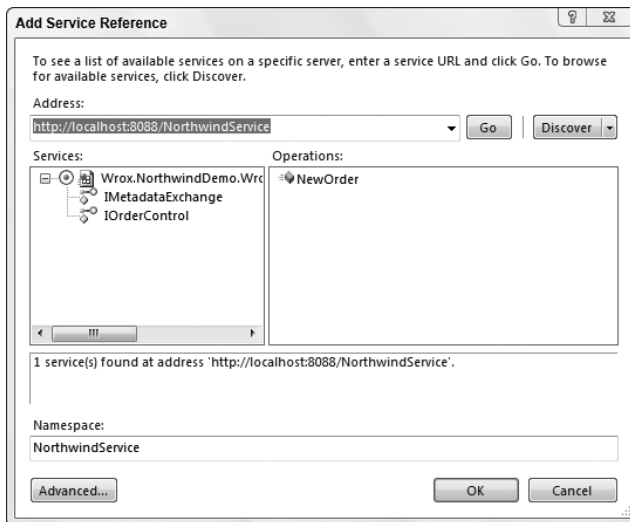


Рис. 51.15. Добавление ссылки на службу

При добавлении ссылки на службу создается прокси-класс, добавляются ссылки на сборки `System.ServiceModel` и `System.Runtime.Serialization`, а к клиентскому приложению добавляется конфигурационный файл приложения, ссылающийся на службу.

Теперь можно использовать сгенерированные сущностные классы и прокси-класс `OrderControlClient` для отправки запроса заказа к обслуживаемому компоненту:

```
static void Main()
{
    var order = new Order
    {
        CustomerId = "PICCO",
        OrderDate = DateTime.Today,
        ShipAddress = "Georg Pipps",
        ShipCity = "Salzburg",
        ShipCountry = "Austria"
    };
    var line1 = new OrderLine
    {
        ProductId = 16,
        UnitPrice = 17.45F,
        Quantity = 2
    };
};
```

```
var line2 = new OrderLine
{
    ProductId = 67,
    UnitPrice = 14,
    Quantity = 1
}
OrderLine[] orderLines = { line1, line2 };
order.orderLines = orderLines;
var occ = new OrderControlClient();
occ.NewOrder(order);
}
```

Резюме

В этой главе мы обсудили богатейшие средства, предлагаемые Enterprise Services, такие как автоматические транзакции, организация пула объектов, компоненты с очередями и слабо связанные события.

Чтобы создать обслуживаемые компоненты, необходимо сослаться на сборку System.EnterpriseServices. Базовым классом для всех обслуживаемых компонентов является ServicedComponent. В этом классе контекст обеспечивает возможность перехвата вызовов методов. Для указания используемых перехватов можно использовать атрибуты. Вы также узнали о том, как конфигурировать приложение и его компоненты с применением атрибутов, а также как управлять транзакциями и специфицировать транзакционные требования компонентов, используя атрибут [Transaction]. Вы также удостоверились, насколько хорошо Enterprise Services интегрируется с коммуникационной технологией WCF за счет создания фасада WCF.

В этой главе было показано, как пользоваться Enterprise Services — средством, предоставляемым операционной системой.