

# 47

## Организация очередей сообщений

### В ЭТОЙ ГЛАВЕ...

- Архитектура Message Queuing
- Использование административных инструментов для работы с Message Queuing
- Программное создание очередей сообщений
- Отправка и прием сообщений
- Пример приложения заказа курсов
- Использование Message Queuing вместе с WCF

### ***Загружаемый код для этой главы***

Загружаемый код для этой главы содержит следующие основные примеры:

- WorkingWithQueues
- CourseOrderApplication
- CourseOrderApplicationWCF

## Краткий обзор

В пространстве имен `System.Messaging` определены классы, которые позволяют выполнять чтение и запись сообщений с использованием средства для организации сообщений `Message Queuing`, доступного в среде операционной системы `Windows`. Обмен сообщениями можно применять в отключенном (автономном) сценарии, при котором не требуется, чтобы клиент и сервер обязательно запускались в одно и то же время.

В этой главе сначала рассказывается об архитектуре и возможных сценариях использования `Message Queuing`, а затем о классах из пространства имен `System.Messaging`, которые можно применять для создания очередей, а также отправки и получения сообщений. Здесь будет показано, как организовать обработку поступающих от сервера сообщений с помощью очередей подтверждающих и ответных сообщений и использовать `Message Queuing` с привязками к очередям сообщений `WCF`.

Прежде чем углубляться в детали программирования с использованием `Message Queuing`, следует ознакомиться с основными концепциями организации очередей сообщений и сравнить их с концепциями синхронного и асинхронного программирования. В случае синхронного программирования код, вызвавший метод, должен дожидаться завершения работы этого метода. При асинхронном программировании вызывающий поток запускает метод и параллельно продолжает свою работу. Асинхронное программирование основано на применении делегатов, библиотек классов, которые уже поддерживают асинхронные методы (например, прокси-классы веб-служб и классы из пространств имен `System.Net` и `System.IO`), либо специальных потоков и задач (глава 21). Как при синхронном, так и при асинхронном программировании клиент и сервер должны работать одновременно.

Хотя `Message Queuing` работает асинхронно, поскольку клиент (отправитель) не ожидает прочтения получателем (сервером) отправленных ему данных, между `Message Queuing` и асинхронным программированием существует принципиальная разница: `Message Queuing` может выполняться в отключенной (автономной) среде. На момент отправки данных их получатель может быть отключен. Позднее, когда получатель подключится, он получит данные без вмешательства со стороны отправляющего приложения.

Программирование для подключенной и отключенной среды можно сравнить с разговором по телефону и отправкой почтовых сообщений. При разговоре с кем-либо по телефону оба участника должны быть подключены одновременно; это синхронная коммуникация. В случае обмена электронной почтой отправитель не знает, когда его сообщение будет прочитано. Люди, использующие эту технологию, работают в отключенном режиме. Конечно, может случиться так, что почта никогда не будет прочитана, а просто проигнорирована. Такова природа отключенных коммуникаций. Чтобы избежать этой проблемы, можно запросить подтверждение факта прочтения письма. Если ответ не придет в течение определенного времени, возможно, придется как-то справляться с таким "исключением". Все это также возможно в `Message Queuing`.

В сущности, `Message Queuing` можно считать технологией для обмена электронными сообщениями между приложениями, а не людьми. Она обладает множеством функциональных возможностей, которые в других службах обмена сообщениями не доступны: гарантированием доставки, применением транзакций, получением подтверждений, экспресс-режимом использования памяти, и т.д. Как будет показано в следующем разделе, `Message Queuing` предлагает массу полезных средств для коммуникаций между приложениями.

С помощью `Message Queuing` можно отправлять, принимать и маршрутизировать сообщения в подключенной и отключенной среде. На рис. 47.1 показан очень простой способ использования сообщений. Отправитель посылает сообщения в очередь сообщений, а получатель принимает их из этой очереди.

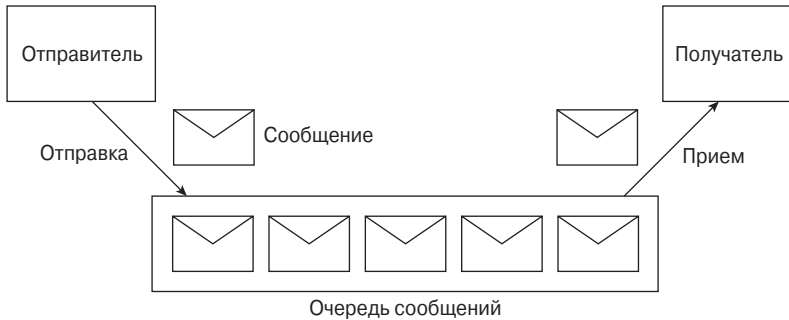


Рис. 47.1. Простой способ использования сообщений

## Когда следует использовать Message Queuing

Одна из ситуаций, в которых удобно применять Message Queuing — это когда клиентское приложение часто отключается от сети (например, у коммивояжера, навещающего заказчиков на местах). Коммивояжер может вводить данные заказа непосредственно у заказчика. Приложение ставит сообщение о каждом заказе в очередь сообщений, находящуюся в клиентской системе (рис. 47.2). Как только коммивояжер возвращается в свой офис, заказ автоматически передается из очереди сообщений клиентской системы в очередь сообщений целевой системы, где и обрабатывается.

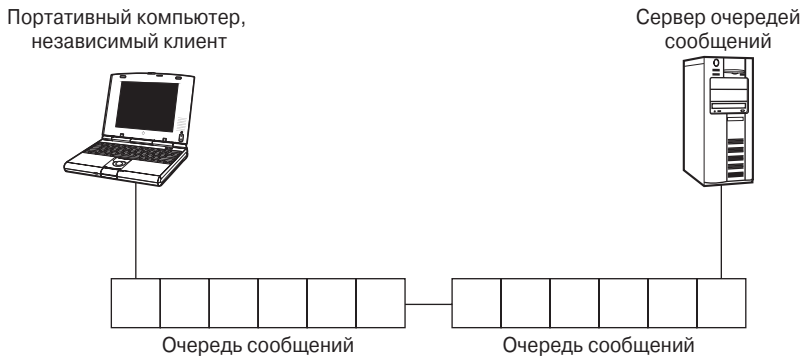


Рис. 47.2. Отключенная среда и Message Queuing

Помимо портативного компьютера, коммивояжер может использовать и меньшее по размеру устройство, на котором также доступна функциональность Message Queuing.

Технология Message Queuing может быть полезна и в подключенной среде. Представьте сайт электронной коммерции (рис. 47.3), где в определенные периоды времени сервер полностью загружен обработкой заказов (например, ранним вечером и в выходные), а по ночам нагрузка значительно уменьшается. Решение проблемы может состоять в приобретении более быстрого сервера или в добавлении дополнительных серверов к системе, чтобы они справлялись с пиковыми нагрузками. Однако существует более дешевое решение: сгладить пиковые нагрузки, сдвинув транзакции со времени максимальных нагрузок на время с низкой загрузкой. В такой схеме заказы отправляются в очередь сообщений, а принимающая сторона читает их тогда, когда это удобно системе управления базой данных. Таким образом, нагрузка на систему сглаживается по времени, так что сервер, обрабатывающий транзакции, может быть дешевле и не требовать модернизации.

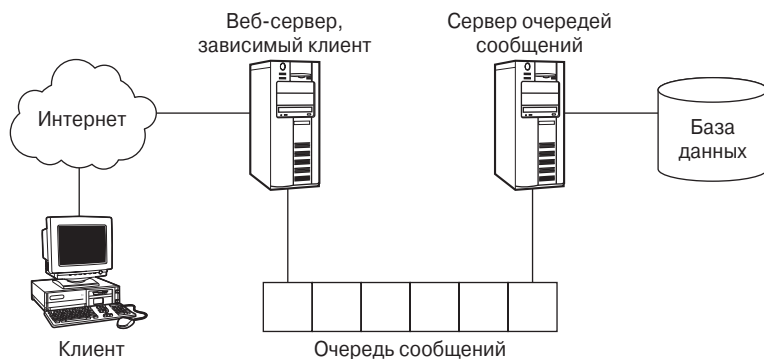


Рис. 47.3. Сайт электронной коммерции, использующий Message Queuing

## Функциональные возможности Message Queuing

Технология Message Queuing является службой, которая поставляется как часть операционной системы Windows 8. Ниже перечислены ее основные функциональные возможности.

- Сообщения могут пересылаться в отключенной среде. То есть приложению-отправителю и приложению-получателю вовсе не обязательно выполняться в одно и то же время.
- В экспресс-режиме сообщения могут пересылаться очень быстро. В этом режиме сообщения просто сохраняются в памяти.
- Для механизма восстановления сообщения могут отправляться с гарантированной доставкой. Такие сообщения сохраняются в файлах и доставляются даже в случае перезагрузки сервера.
- Очереди сообщений могут защищаться с применением списков контроля доступа и указания в них, каким пользователям разрешено отправлять или получать сообщения из очереди. Кроме того, сообщения могут шифроваться для исключения вероятности их прочтения с помощью сетевых анализаторов пакетов, а также снабжаться приоритетами, чтобы те из них, которые имеют более высокий приоритет, обрабатывались быстрее.
- В Message Queuing 3.0 поддерживается возможность отправки групповых сообщений.
- В Message Queuing 4.0 поддерживается возможность распознавания *вредоносных сообщений* (т.е. сообщений, которые не удалось преобразовать). Для таких сообщений может быть определена специальная *очередь вредоносных сообщений*. Например, в случае, если после прочтения сообщения из обычной очереди оно должно вставляться в базу данных, но по какой-то причине этого не происходит, такое сообщение может быть отправлено в очередь вредоносных сообщений. Впоследствии этой очередью вредоносных сообщений должен кто-нибудь заняться и выяснить, по какой причине адрес сообщения не удалось распознать.
- В Message Queuing 5.0 поддерживаются более безопасные алгоритмы аутентификации, и может обрабатываться большее количество очередей. (В Message Queuing 4.0 при обработке нескольких тысяч очередей начинали возникать проблемы с производительностью.)

**На заметку!** Поскольку Message Queuing является частью операционной системы, установить версию Message Queuing 5.0 в системе Windows Vista или Windows Server 2008 не получится. Message Queuing 5.0 является частью операционных систем Windows 7, Windows 8, Windows Server 2008 R2 и Windows Server 2012.

В остальных разделах главы будет показано, как пользоваться всеми этими функциональными возможностями.

## Продукты Message Queuing

Message Queuing 5.0 является частью операционной системы Windows, начиная с версий Windows 7 и Windows Server 2008 R2. В состав Windows 2000 входила версия Message Queuing 2.0, в которой не было поддержки протокола HTTP и групповых сообщений. Версия Message Queuing 3.0 поставлялась в Windows XP и Windows Server 2003, а версия Message Queuing 4.0 — в Windows Vista и Windows Server 2008.

При использовании ссылки Turn Windows Features On or Off (Включение или отключение компонентов Windows) в окне Configuring Programs and Features (Программы и компоненты) Windows 8 можно обнаружить отдельный раздел с опциями, касающимися Message Queuing. В этом разделе доступны для выбора перечисленные ниже компоненты.

- Microsoft Message Queue (MSMQ) Server Core (Основные компоненты сервера очереди сообщений (MSMQ)). Основные компоненты MSMQ необходимы для получения базовой функциональности Message Queuing.
- Active Directory Domain Services Integration (Интеграция доменных служб Active Directory). Это средство позволяет записывать имена очередей сообщений в Active Directory. С помощью этой опции можно находить очереди в Active Directory и защищать их на основе пользователей и групп пользователей Windows.
- MSMQ HTTP Support (Поддержка протокола HTTP MSMQ). Поддержка MSMQ HTTP позволяет отправлять и принимать сообщения, используя протокол HTTP.
- Triggers (Триггеры MSMQ). С помощью триггеров создаются экземпляры приложений при поступлении нового сообщения.
- Multicast Support (Поддержка групповой рассылки). Позволяет отправлять сообщения группам серверов.
- MSMQ DCOM Proxy (Прокси MSMQ DCOM). С помощью DCOM-прокси система может подключаться к удаленному серверу с применением API-интерфейса DCOM.

После установки Message Queuing в системе должна быть обязательно запущена служба Message Queuing (рис. 47.4). Эта служба читает и записывает сообщения, а также взаимодействует с другими серверами Message Queuing для осуществления маршрутизации сообщений по сети.

## Архитектура Message Queuing

В Message Queuing сообщения записываются и читаются из специальной очереди сообщений. Сами сообщения и очереди сообщений имеют несколько атрибутов, которые требуют пояснений.

### Сообщения

Сообщение отправляется в очередь сообщений. Сообщение включает тело, содержащее пересылаемые данные, и метку, являющуюся заголовком сообщения. В тело сообщения может быть помещена любая информация.

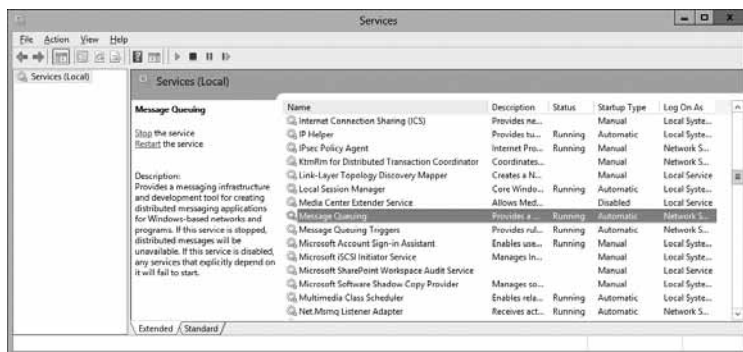


Рис. 47.4. Функционирующая служба Message Queuing

В .NET имеется набор форматировщиков, преобразующих данные, которые помещаются в тело. Кроме метки и тела сообщение может содержать дополнительную информацию от отправителя, конфигурации таймаута, идентификаторе транзакции или приоритете.

Очереди сообщений содержат сообщения нескольких типов.

- *Нормальное сообщение* отправляется любым приложением.
- *Подтверждающее сообщение* уведомляет о состоянии нормального сообщения. Подтверждающие сообщения отправляются в административные очереди, чтобы уведомить об успехе или сбое при отправке нормальных сообщений.
- Ответные сообщения отправляются принимающим приложением, когда исходный отправитель требует некоторого специального ответа.
- *Отчетные сообщения* генерируются системой Message Queuing. К этой категории относятся тестовые сообщения и сообщения отслеживания маршрутизации.

Сообщение может обладать приоритетом, определяющим порядок, в котором сообщения будут читаться из очереди. Сообщения сортируются в очереди в соответствии с приоритетами, поэтому следующим из очереди всегда читается сообщение, имеющее наивысший приоритет.

Сообщения имеют два режима доставки: *экспресс* и *восстановимый*. Экспресс-сообщения доставляются очень быстро, потому что в качестве хранилища очереди используется оперативная память. Восстановимые сообщения сохраняются в файлах на каждом шаге маршрута — до тех пор, пока они не будут доставлены. Таким образом, доставка сообщений гарантируется, даже если компьютер будет перегружен или произойдет сбой в сети.

*Транзакционные сообщения* — это специальная версия восстановимых сообщений. Благодаря транзакционным сообщениям гарантируется, что сообщения будут доставлены только однажды, и в том же порядке, в каком были отправлены. С транзакционными сообщениями приоритеты не используются.

## Очередь сообщений

Очередь сообщений представляет собой своего рода “накопительный бункер” для сообщений. Сообщения, сохраняемые на диске, размещаются в каталоге `<windows>system32\msmq\storage`.

Для отправки сообщений обычно применяются открытые или закрытые очереди, но существуют также и другие типы очередей.

*Открытая очередь* публикуется в Active Directory. Информация о таких очередях реплицируется в доменах Active Directory. Для получения информации о таких очередях можно воспользоваться средствами просмотра и поиска.

К открытой очереди можно обращаться, не зная имени компьютера, на котором она расположена. Такую очередь можно перемещать с одной системы на другую и клиент этого не заметит. Открытые очереди нельзя создавать в среде рабочей группы, потому что им нужна служба Active Directory.

*Закрытые* очереди не публикуются в Active Directory. Эти очереди доступны, только когда известны их полные путевые имена. Их можно использовать в среде рабочей группы.

*Журнальные* очереди служат для хранения копий сообщений после того, как они были получены или отправлены. Включение протоколирования для открытой или закрытой очереди автоматически создает журнальную очередь. Журнальные очереди бывают двух типов: исходное протоколирование и целевое протоколирование. *Исходное протоколирование* включается свойствами сообщения; журнальные сообщения сохраняются на исходной системе. *Целевое протоколирование* включается свойствами очереди; эти сообщения сохраняются в журнальной очереди целевой системы.

*Очереди пропавших писем* хранят сообщения, если они не появляются на целевой системе по истечении определенного периода времени. В противоположность синхронному программированию, где ошибки обнаруживаются немедленно, при использовании Message Queuing ошибки должны обрабатываться иначе. Очередь пропавших писем может быть проверена на наличие недоставленных сообщений.

*Административные очереди* содержат подтверждения об отправленных сообщениях. Отправитель может указать административную очередь, из которой он будет получать уведомления об успешной отправке сообщений.

*Очередь ответов* применяется, когда в качестве ответа со стороны получателя требуется нечто большее, чем простое подтверждение факта отправки. Принимающее приложение может отправлять ответные сообщения отправителю исходного сообщения.

*Очередь отчетов* используется для тестовых сообщений. Очереди отчетов могут быть созданы изменением типа (или категории) открытой или закрытой очереди на предопределенный идентификатор {55EE8F33-CCE9-11CF-B108-0020AFD61CE9}. Очереди отчетов удобны в качестве инструмента тестирования для отслеживания сообщений на их маршруте.

*Системные очереди* являются закрытыми и используются самой системой Message Queuing. Эти очереди предназначены для административных сообщений, хранения уведомлений и обеспечения правильного порядка доставки транзакционных сообщений.

## Административные инструменты для работы с Message Queuing

Прежде чем переходить к рассмотрению программного взаимодействия с Message Queuing, следует ознакомиться с административными инструментами, которые поставляются в составе операционной системы Windows для создания и управления очередями и сообщениями.

**На заметку!** Описанные здесь инструменты применяются для работы не только с Message Queuing. Возможности, касающиеся Message Queuing, становятся доступными в этих инструментах только после установки Message Queuing в системе.

### Создание очередей сообщений

Очереди сообщений могут создаваться с помощью оснастки Computer Management (Управление компьютером) консоли управления MMC. В панели древовидного представления Message Queuing находится ниже элемента Services and Applications (Службы и приложения). Выбрав Private Queues (Закрытые очереди) или Public Queues (Открытые очереди), можно создать новую очередь из меню Action (Действие), как показано на рис. 47.5.

С открытыми очередями можно работать, только если служба Message Queuing сконфигурирована в режиме Active Directory.

## Свойства очереди сообщений

После создания очереди в оснастке Computer Management можно модифицировать ее свойства, выделив очередь в древовидной панели и выбрав в меню пункт Action⇒Properties (Действие⇒Свойства), как показано на рис. 47.6.

Здесь для конфигурирования доступно несколько опций.

- Label (Метка) — имя очереди, которое может использоваться для ее поиска.
- Type ID (Идентификатор типа), который по умолчанию устанавливается в {00000000-0000-0000-0000-000000000000}, предназначен для отображения множества очередей на единственную категорию или тип. Отчетные очереди, как было сказано ранее, используют специфический идентификатор типа. Идентификатор типа представляет собой универсальный уникальный идентификатор (UUID) или глобально уникальный идентификатор (GUID).

**На заметку!** Специальные идентификаторы типа могут быть созданы с помощью утилиты uuidgen.exe или guidgen.exe. Утилита uuidgen.exe — это инструмент командной строки, служащий для создания уникальных идентификаторов, а guidgen.exe — ее графическая версия, предназначенная для создания UUID.

- Во избежание переполнения диска, максимальный размер всех сообщений (Limit message storage to (KB)) в очереди может быть ограничен.
- Установленный флажок Authenticated (Аутентифицированные) позволяет записывать и читать сообщения в очереди только аутентифицированным пользователям.
- Опция Privacy Level (Уровень приватности) позволяет шифровать содержимое сообщения. Возможные значения: None (Нет), Optional (Необязательно) или Body (Тело). None означает, что зашифрованные сообщения не принимаются, Body — принимаются только зашифрованные сообщения, а стандартное значение Optional — что принимаются те и другие сообщения.

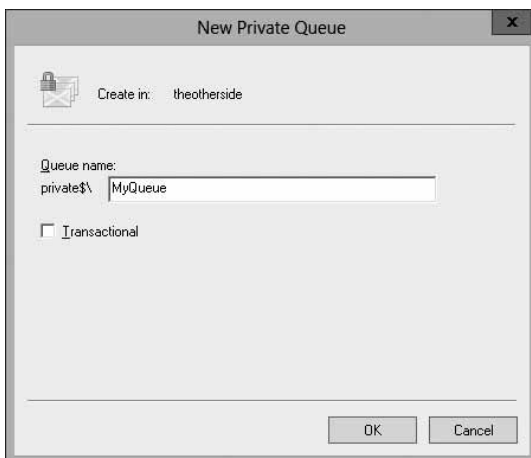


Рис. 47.5. Создание новой очереди



Рис. 47.6. Изменение свойств очереди

- Целевое протоколирование может быть сконфигурировано посредством опций раздела Journal (Журнал). С помощью этих опций обеспечивается сохранение в журнале копий принятых сообщений. Для журнальных сообщений очереди может быть указан максимальный размер занимаемого дискового пространства. По достижении максимального размера целевой журнал очищается.
- В случае опции конфигурации Multicast (Групповой) можно определить групповой IP-адрес для очереди. Один и тот же групповой IP-адрес может применяться разными узлами в сети, так что сообщение, отправленное по одному адресу, принимается множеством очередей.

## Программирование с использованием Message Queuing

Теперь, когда вы ознакомились с архитектурой Message Queuing, можно приступить к программированию. В последующих разделах будет показано, как создавать и управлять очередями, а также как отправлять и принимать сообщения.

Вдобавок будет построено простое приложение заказа курсов, состоящее из отправляющей и принимающей части.

### Создание очереди сообщений

Вы уже видели, как очереди сообщений создаются утилитой Computer Management. Но очереди сообщений могут быть созданы и программно вызовом метода `Create()` класса `MessageQueue`.

Методу `Create()` должен быть передан путь к новой очереди. Этот путь состоит из имени хоста, где расположена очередь, и имени очереди. В этом примере очередь `MyNewPublicQueue` создается на локальном хосте. Чтобы создать закрытую очередь, путь должен включать `Private$`; например, `\Private$\MyNewPrivateQueue`.

После вызова метода `Create` свойства очереди могут быть изменены. Например, используя свойство `Label`, установим метку очереди в `Demo Queue`. В примере программы путь очереди и форматное имя выводятся на консоль. Форматное имя создается автоматически с `UUID`, который может применяться для доступа к очереди без указания имени сервера (файл `WorkingWithQueues\CreateMessageQueue\Program.cs`):

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            using (var queue = MessageQueue.Create(@".\MyNewPublicQueue"))
            {
                queue.Label = "Demo Queue";
                Console.WriteLine("Queue created:");
                Console.WriteLine("Path: {0}", queue.Path);
                Console.WriteLine("FormatName: {0}", queue.FormatName);
            }
        }
    }
}
```

**На заметку!** Для создания очереди необходимы административные привилегии. Обычно нельзя рассчитывать, что пользователь приложения будет их иметь. Именно поэтому очереди обычно создаются программой установки. В разделе “Установка очередей сообщений” будет показано, как создавать очереди сообщений с помощью класса `MessageQueueInstaller`.

## Нахождение очереди

Для идентификации очередей можно использовать путевое имя и форматное имя. При поиске очереди необходимо делать различия между открытыми и закрытыми очередями. Открытые очереди публикуются в Active Directory. Для таких очередей не обязательно знать систему, в которой они расположены. Закрытые очереди могут быть найдены только в случае, если известно имя системы, в которой расположена очередь.

Открытые очереди в домене Active Directory можно искать по метке очереди, категории или по форматному имени. Можно также получить все очереди, имеющиеся на компьютере. В классе `MessageQueue` предусмотрены статические методы для поиска: `GetPublicQueuesByLabel()`, `GetPublicQueuesByCategory()` и `GetPublicQueuesByMachine()`. Метод `GetPublicQueues()` возвращает массив всех открытых очередей в домене (файл `WorkingWithQueues\FindQueues\Program.cs`):

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            foreach (var queue in MessageQueue.GetPublicQueues())
            {
                Console.WriteLine(queue.Path);
            }
        }
    }
}
```

Метод `GetPublicQueues()` перегружен. Одна из версий позволяет передавать экземпляр класса `MessageQueueCriteria`. С помощью этого класса можно искать очереди, созданные или модифицированные до или после определенного момента времени, а также учитывать категорию, метку или имя компьютера.

Закрытые очереди можно искать с помощью статического метода `GetPrivateQueuesByMachine()`. Этот метод возвращает все закрытые очереди из определенной системы.

## Открытие известных очередей

Если имя очереди известно, то искать ее не обязательно. Очереди могут открываться с использованием пути или форматного имени. И то, и другое может быть установлено в конструкторе класса `MessageQueue`.

### Путевое имя

Путь указывает имя компьютера и имя очереди для ее открытия. В следующем примере кода открывается очередь `MyPublicQueue` на локальном хосте. Чтобы удостовериться в существовании очереди, следует применить статический метод `MessageQueue.Exists()` (файл `WorkingWithQueues\OpenQueue\Program.cs`):

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            if (MessageQueue.Exists(@".\MyPublicQueue"))
            {
                var queue = new MessageQueue(@".\MyPublicQueue");
                //...
            }
            else
            {
                Console.WriteLine("Queue .\MyPublicQueue not existing");
            }
        }
    }
}
```

В зависимости от типа очереди при открытии очередей должны использоваться разные идентификаторы. В табл. 47.1 представлен синтаксис имен очередей разных типов.

**Таблица 47.1. Синтаксис имен очередей разных типов**

| Тип очереди                                       | Синтаксис                          |
|---|------------------------------------|
| Открытая очередь                                  | ИмяКомпьютера\ИмяОчереди           |
| Закрытая очередь                                  | ИмяКомпьютера\Private\$\ИмяОчереди |
| Журнальная очередь                                | ИмяКомпьютера\ИмяОчереди\Journal\$ |
| Журнальная очередь компьютера                     | ИмяКомпьютера\Journal\$            |
| Очередь пропавших писем компьютера                | ИмяКомпьютера\DeadLetter\$         |
| Транзакционная очередь пропавших писем компьютера | ИмяКомпьютера\XactDeadLetter\$     |

Если для открытия открытой очереди используется путевое имя, необходимо передавать имя компьютера. Если имя компьютера не известно, вместо него может быть указано форматное имя. Путевое имя закрытой очереди может применяться только на локальных системах. Форматное имя должно использоваться для удаленного доступа к закрытым очередям.

### Форматное имя

Вместо путевого имени для открытия очереди можно применять форматное имя. Форматное имя применяется для поиска очереди в Active Directory, чтобы получить хост, на котором расположена очередь. В отключенной среде, где очередь недоступна во время отправки сообщения, нужно использовать форматное имя:

```
var queue = new MessageQueue(
    @"FormatName:PUBLIC=09816AFF-3608-4c5d-B892-69754BA151FF");
```

Форматное имя применяется по-другому. С его помощью можно открывать закрытые очереди и указывать нужный протокол.

- Для доступа к закрытой очереди строка, передаваемая конструктору, выглядит так: `FormatName:PRIVATE=GUID_Компьютера\НомерОчереди`. Номер `НомерОчереди` для закрытых очередей генерируется при их создании. Номера очередей можно просмотреть в каталоге `<windows>\System32\msmq\storage\lqs`.
- Строка `FormatName:DIRECT=Protocol:АдресКомпьютера\ИмяОчереди` позволяет указать протокол, который должен использоваться для отправки сообщений. Протокол HTTP поддерживается, начиная с версии Message Queuing 3.0.
- `FormatName:DIRECT=OS:ИмяКомпьютера\ИмяОчереди` — еще один способ задания очереди с применением форматного имени. При этом протокол указывать не нужно, но в форматном имени по-прежнему можно использовать имя компьютера.

## Отправка сообщения

Для отправки сообщения в очередь используется метод `Send()` класса `MessageQueue`. Объект, переданный в качестве аргумента методу `Send()`, сериализируется в ассоциированную очередь. Метод `Send()` перегружен так, что ему можно передавать метку и объект `MessageQueueTransaction`. Транзакционное поведение Message Queuing описано в разделе “Транзакционные очереди”.

В приведенном ниже примере кода сначала выполняется проверка существования очереди. Если очередь не существует, она создается. Затем очередь открывается и с помощью метода `Send()` в нее отправляется сообщение `Sample Message`.

В путевом имени вместо имени сервера указана точка, что означает локальную систему. Путь к закрытым очередям работают только в локальной системе (файл `WorkingWithQueues\SendMessage\Program.cs`):

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            try
            {
                if (!MessageQueue.Exists(@".\Private$\MyPrivateQueue"))
                {
                    MessageQueue.Create(@".\Private$\MyPrivateQueue");
                }
                var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
                queue.Send("Sample Message", "Label");
            }
            catch (MessageQueueException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

На рис. 47.7 показано окно оснастки Computer Management, в котором можно просматривать сообщения, появляющиеся в очереди.

Открыв сообщение и выбрав в диалоговом окне вкладку `Body` (Тело), которая показана на рис. 47.8, можно увидеть, что сообщение сформатировано с использованием XML. Способ форматирования сообщения — функция форматировщика, ассоциированного с очередью сообщений.

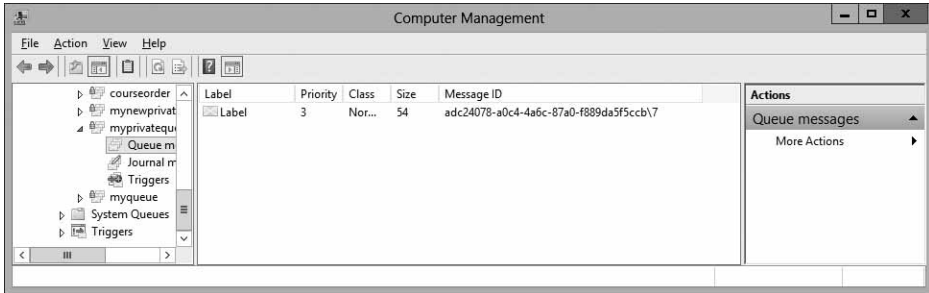


Рис. 47.7. Просмотр сообщений, появляющихся в очереди



Рис. 47.8. Вкладка Body

### Форматировщик сообщений

Формат, в котором передаются сообщения в очередь, зависит от используемого форматировщика. Класс `MessageQueue` имеет свойство `Formatter`, через которое очередь может быть назначен объект-форматировщик. Стандартный форматировщик — `XmlMessageFormatter` — форматирует сообщение в синтаксисе XML, как показано в предыдущем примере.

Форматировщик сообщений реализует интерфейс `IMessageFormatter`. В пространстве имен `System.Messaging` доступны три форматировщика сообщений.

- `XmlMessageFormatter` — стандартный форматировщик. Он сериализует объекты, используя XML. Подробнее форматирование XML описано в главе 34.
- Форматировщик `BinaryMessageFormatter` сериализует сообщения в двоичный формат. Эти сообщения короче, чем сформатированные с применением XML.
- `ActiveXMessageFormatter` — двоичный форматировщик, обеспечивающий чтение или запись сообщений объектами COM. Используя этот форматировщик, можно записывать сообщения в очередь с помощью классов .NET и читать их оттуда объектами COM, и наоборот.

Простое сообщение, показанное на рис. 47.8 в формате XML, на рис. 46.9 сформатировано с помощью `BinaryMessageFormatter`.

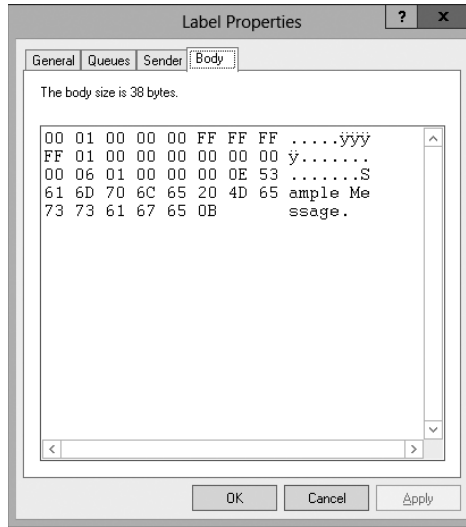


Рис. 47.9. Просмотр сообщения, сформатированного `BinaryMessageFormatter`

### Отправка сложных сообщений

Вместо строк методу `Send()` класса `MessageQueue` можно передавать объекты. Тип класса таких объектов должен соответствовать определенным требованиям, но они зависят от форматировщика.

Для двоичного форматировщика класс должен быть сериализуемым и снабжен атрибутом `[Serializable]`. При сериализации исполняющей средой .NET сериализуются все поля (включая приватные). Специальная сериализация может быть определена посредством реализации интерфейса `ISerializable`. Более подробно сериализация времени выполнения .NET описана в главе 24.

Сериализация XML выполняется с помощью XML-форматировщика. При сериализации XML сериализуются все открытые поля и свойства. На сериализацию XML могут оказывать влияние атрибуты из пространства имен `System.Xml.Serialization`. Более подробно сериализация XML рассматривается в главе 34.

### Прием сообщений

Для чтения сообщений можно использовать класс `MessageQueue`. Метод `Receive()` читает одиночное сообщение и удаляет его из очереди. Если сообщения отправлены с разными приоритетами, читается сообщение с наивысшим приоритетом. Чтение сообщений с одинаковым приоритетом не обеспечивает поступление сообщений в порядке их отправки, поскольку этот порядок не гарантируется в сети. Для получения гарантированного порядка следует применять транзакционные очереди сообщений.

В следующем примере сообщение читается из закрытой очереди `MyPrivateQueue`. Ранее в сообщение была передана простая строка. При чтении сообщения с использованием `XmlMessageFormatter` типы считываемых объектов необходимо передавать конструктору форматировщика. В данном примере в массив аргументов конструктора `XmlMessageFormatter` передается тип `System.String`. Этот конструктор принимает либо массив `String`, содержащий передаваемые типы в виде строк, либо массив объектов `Type`.

Сообщение читается методом `Receive()`, после чего тело сообщения выводится на консоль (файл `WorkingWithQueues\SendMessage\Program.cs`):

```
using System;
using System.Messaging;
namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
            queue.Formatter = new XmlMessageFormatter(
                new string[] { "System.String" });
            Message message = queue.Receive();
            Console.WriteLine(message.Body);
        }
    }
}
```

Метод `Receive` ведет себя синхронно и ожидает появления сообщения в очереди, если на момент его вызова там было пусто.

### Перечисление сообщений

Вместо чтения сообщений одного за другим с помощью метода `Receive()` можно применить перечислитель для прохождения по всем сообщениям в очереди. Класс `MessageQueue` реализует интерфейс `IEnumerable`, и потому может быть использован в операторе `foreach`. В этом случае сообщения не удаляются из очереди, а только просматриваются для извлечения их содержимого.

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(new string[] { "System.String" });
foreach (Message message in queue)
{
    Console.WriteLine(message.Body);
}
```

Вместо интерфейса `IEnumerable` можно применять класс `MessageEnumerator`, который реализует интерфейс `IEnumerator`, но предоставляет ряд дополнительных возможностей. При использовании интерфейса `IEnumerable` сообщения не удаляются из очереди. Метод `RemoveCurrent()` класса `MessageEnumerator` удаляет сообщение из текущей позиции курсора перечислителя.

В приведенном ниже примере метод `GetMessageEnumerator()` класса `MessageQueue` используется для доступа к `MessageEnumerator`. Метод `MoveNext()` перебирает сообщение за сообщением с помощью `MessageEnumerator`. Метод `MoveNext()` перегружен и принимает дополнительный аргумент — период времени. Это одно из значительных преимуществ использования перечислителя. Здесь поток может ожидать, пока сообщение появится в очереди, но только в течение определенного периода времени. Свойство `Current`, определенное интерфейсом `IEnumerator`, возвращает ссылку на сообщение.

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(new string[] { "System.String" });
using (MessageEnumerator messages = queue.GetMessageEnumerator())
{
    while (messages.MoveNext(TimeSpan.FromMinutes(30)))
    {
        Message message = messages.Current;
        Console.WriteLine(message.Body);
    }
}
```

## Асинхронное чтение

Метод `Receive()` класса `MessageQueue` ожидает, пока не будет прочитано сообщение из очереди. Во избежание блокировки потока в перегруженной версии этого метода можно указать время ожидания. Чтобы прочитать сообщение после истечения этого таймута, метод `Receive()` должен быть вызван заново. Вместо постоянного опроса сообщений можно применить асинхронный метод `BeginReceive()`. Прежде чем запустить асинхронное чтение методом `BeginReceive()`, следует установить обработчик события `ReceiveCompleted`. Событие `ReceiveCompleted` требует делегата `ReceiveCompletedEventHandler`. Этот делегат ссылается на метод, вызываемый после появления в очереди сообщения, которое может быть прочитано. В рассматриваемом примере метод `MessageArrived()` передается делегату `ReceivedCompletedEventHandler` (файл `WorkingWithQueues\ReceiveMessageAsync\Program.cs`):

```
var queue = new MessageQueue(@".\Private$\MyPrivateQueue");
queue.Formatter = new XmlMessageFormatter(
    new string[] { "System.String" });
queue.ReceiveCompleted += MessageArrived;
queue.BeginReceive();
// Поток не ожидает.
```

Метод обработчика `MessageArrived()` принимает два параметра. Первым параметром является источник события — `MessageQueue`. Вторым параметром типа `ReceiveCompletedEventArgs` содержит сообщение и асинхронный результат. В этом примере метод `EndReceive()` из очереди вызывается для получения результата асинхронного метода, т.е. сообщения:

```
public static void MessageArrived(object source, ReceiveCompletedEventArgs e)
{
    MessageQueue queue = (MessageQueue)source;
    Message message = queue.EndReceive(e.AsyncResult);
    Console.WriteLine(message.Body);
}
```

Если сообщение не должно удаляться из очереди, для асинхронного ввода-вывода следует применять методы `BeginPeek()` и `EndPeek()`.

## Приложение заказа курсов

Чтобы продемонстрировать использование `Message Queuing`, в этом разделе мы создадим простое приложение заказа курсов. Этот пример решения состоит из трех сборок:

- библиотека компонентов (`CourseOrder`), которая включает сущностные классы для сообщений, отправляемых и принимаемых в очереди;
- WPF-приложение (`CourseOrderSender`), отправляющее сообщения в очередь;
- WPF-приложение (`CourseOrderReceiver`), принимающее сообщения из очереди.

## Библиотека классов заказа курсов

И отправляющее, и принимающее приложения нуждаются в информации о заказе. По этой причине сущностные классы помещены в отдельную сборку. Сборка `CourseOrder` включает три сущностных класса `CourseOrder`, `Course` и `Customer` и базовый класс `BindableBase`. В этом примере приложения реализованы не все свойства, которые могут присутствовать в реальном приложении, а лишь необходимые для демонстрации концепций.

В файле `BindableBase.cs` определен класс `BindableBase`. Он реализует интерфейс `INotifyPropertyChanged` и предоставляет метод `SetProperty()`, который может быть вызван из унаследованного класса внутри установщика свойства (файл `CourseOrderApplication\CourseOrder\BindableBase.cs`):

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Wrox.ProCSharp.Messaging
{
    public abstract class BindableBase : INotifyPropertyChanged
    {
        protected void SetProperty<T>(ref T prop, T value,
            [CallerMemberName] string callerName = "")
        {
            if (!EqualityComparer<T>.Default.Equals(prop, value))
            {
                prop = value;
                OnPropertyChanged(callerName);
            }
        }

        protected virtual void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler propertyChanged = PropertyChanged;
            if (propertyChanged != null)
            {
                propertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
    }
}
```

В файле `Course.cs` определен класс `Course`. Он имеет только одно свойство, определяющее название курса (файл `CourseOrderApplication\CourseOrder\Course.cs`):

```
public class Course : BindableBase
{
    private string title;
    public string Title
    {
        get { return title; }
        set
        {
            SetProperty(ref title, value);
        }
    }
}
```

Файл `Customer.cs` включает класс `Customer`, в котором имеются свойства для названия компании и контактного имени (файл `CourseOrderApplication\CourseOrder\Customer.cs`):

```
public class Customer : BindableBase
{
    private string company;
    public string Company
    {
        get { return company; }
    }
}
```

```
        set
        {
            SetProperty(ref company, value);
        }
    }
    private string contact;
    public string Contact
    {
        get { return contact; }
        set
        {
            SetProperty(contact, value);
        }
    }
}
```

Класс `CourseOrder` в файле `CourseOrder.cs` связывает заказчика с курсом внутри заказа и определяет приоритет заказа. Кроме того, в этом классе определено имя очереди, для которого устанавливается форматное имя открытой очереди. Форматное имя используется для отправки сообщения, даже если в текущий момент очередь не доступна. Получить форматное имя можно, прочитав идентификатор очереди сообщений с помощью оснастки **Computer Management**. Если доступ к **Active Directory** для создания открытой очереди отсутствует, этот код легко изменить так, чтобы в нем применялась закрытая очередь (файл `CourseOrderApplication\CourseOrder\CourseOrder.cs`):

```
public class CourseOrder : BindableBase
{
    public const string CourseOrderQueueName =
        "FormatName:Public=D99CE5F3-4282-4a97-93EE-E9558B15EB13";

    private Customer customer;
    public Customer Customer
    {
        get { return customer; }
        set
        {
            SetProperty(ref customer, value);
        }
    }

    private Course course;
    public Course Course
    {
        get { return course; }
        set
        {
            SetProperty(ref course, value);
        }
    }
}
```

## Отправитель сообщения о заказе курса

Вторая часть решения представлена WPF-приложением `CourseOrderSender`. Это приложение отправляет заказы курсов в очередь сообщений. Должны присутствовать ссылки на сборки `System.Messaging` и `CourseOrder`.

Пользовательский интерфейс этого приложения показан на рис. 47.10. Поле со списком позволяет выбрать доступный курс, а в нескольких текстовых полях можно ввести соответствующий текст, прежде чем отправить заказ.

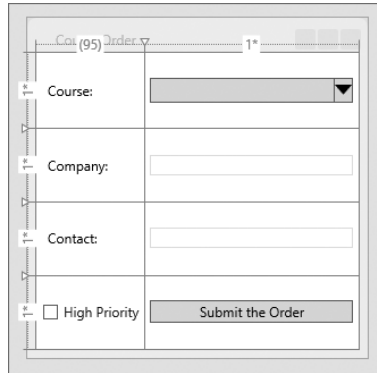


Рис. 47.10. Приложение для отправки заказа курсов

В коде XAML используется привязка данных WPF, как показано в следующем фрагменте кода. Объект `ComboBox` связан со свойством `Courses`, которое возвращает список доступных курсов (файл `CourseOrderApplication\CourseOrderSender\CourseOrderWindow.xaml`):

```
<CheckBox Grid.Row="3" Grid.Column="0"
    IsChecked="{Binding MessageConfiguration.HighPriority, Mode=OneWayToSource}">
    High Priority</CheckBox>
<ComboBox ItemsSource="{Binding Courses}" Grid.Row="0" Grid.Column="1"
    SelectedItem="{Binding CourseOrder.Course.Title, Mode=OneWayToSource}" />
<TextBox Text="{Binding CourseOrder.Customer.Company}" Grid.Row="1"
    Grid.Column="1" />
<TextBox Text="{Binding CourseOrder.Customer.Contact}" Grid.Row="2"
    Grid.Column="1" />
<Button Click="buttonSubmit_Click" Grid.Row="3" Grid.Column="1">
    Submit the Order</Button>
```

Привязанные свойства из файла отдельного кода показаны ниже. Свойство `Courses` просто возвращает коллекцию строк, которая содержит доступные курсы. Свойство `CourseOrder` типа `CourseOrder` принимает данные ввода от пользователя. Класс `CourseOrder` создает библиотеку классов заказа курсов (файл `CourseOrderApplication\CourseOrderSender\CourseOrderWindow.xaml.xs`):

```
public partial class CourseOrderWindow : Window
{
    private readonly ObservableCollection<string> courseList =
        new ObservableCollection<string>();
    private readonly CourseOrder courseOrder = new CourseOrder();
    private readonly MessageConfiguration messageConfiguration =
        new MessageConfiguration();

    public CourseOrderWindow()
    {
        InitializeComponent();
        FillCourses();
        this.DataContext = this;
    }

    public IEnumerable<string> Courses
    {
        get
        {
            return courseList;
        }
    }
}
```

```
private void FillCourses()
{
    courseList.Add("Parallel .NET Programming");
    courseList.Add("Data Access with the ADO.NET Entity Framework");
    courseList.Add("Distributed Solutions with WCF");
    courseList.Add("Windows 8 Metro Apps with XAML and C#");
}

public CourseOrder CourseOrder
{
    get
    {
        return courseOrder;
    }
}

public MessageConfiguration MessageConfiguration
{
    get
    {
        return messageConfiguration;
    }
}
```

При щелчке на кнопке Submit the Order (Отправить заказ) вызывается метод-обработчик `buttonSubmit_Click`. Этот метод создает экземпляр `MessageQueue` для открытия очереди сообщений с форматным именем. Метод `Send` передает объект `courseOrder` для записи сообщения в очередь (файл `CourseOrderApplication\CourseOrderSender\CourseOrderWindow.xaml.cs`).

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
        {
            queue.Send(courseOrder, String.Format("Course Order {{0}}",
                courseOrder.Customer.Company));
        }

        MessageBox.Show("Course Order submitted", "Course Order",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

## Отправка приоритетных и восстанавливаемых сообщений

Устанавливая свойство `Priority` класса `Message`, сообщениям можно назначать приоритеты. Если сообщения специально конфигурируются, то должен быть создан объект `Message`, конструктору которого передается тело сообщения.

В следующем примере приоритет устанавливается в `MessagePriority.High` или `MessagePriority.Normal`, в зависимости от выбора пользователя, заявленного в свойстве `MessageConfiguration.HighPriority`, которое привязано к флажку. Перечисление `MessagePriority` позволяет устанавливать значения от `Lowest` (0) до `Highest` (7). Стандартное значение, `Normal`, соответствует величине приоритета 3.

Чтобы сделать сообщение восстанавливаемым, понадобится установить свойство `Recoverable` в `true`:

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
        using (var message = new Message(courseOrder)
            {
                Recoverable = true,
                Priority = MessageConfiguration.HighPriority == true ?
                    MessagePriority.High : MessagePriority.Normal
            })
        {
            queue.Send(message, String.Format("Course Order {{{0}}}",
                courseOrder.Customer.Company));
        }

        MessageBox.Show("Course Order submitted", "Course Order",
            MessageBoxButton.OK, MessageBoxImage.Information);
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

Запустив это приложение, можно добавлять заказы курсов в очередь сообщений (рис. 47.11).

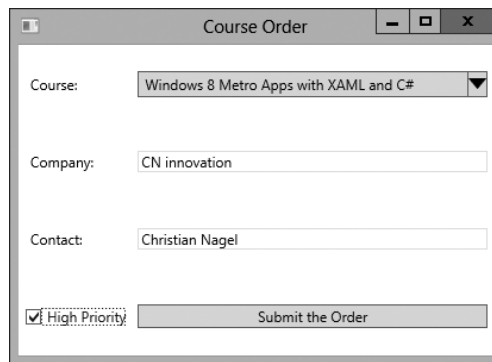


Рис. 47.11. Добавление заказа курсов

## Получатель сообщения о заказе курса

Представление визуального конструктора приложения для получения заказа курсов, которое читает сообщения из очереди, показано на рис. 47.12. В этом приложении отображаются метки каждого заказа в списке `listOrders`. Когда заказ выбран, его содержимое отображается в элементах управления в правой части окна приложения.

Как и предыдущее WPF-приложение, приложение для получения также использует привязку данных. В данном случае `ListBox` связан со свойством `OrdersList`, которое возвращает список всех заказов. Объект `Grid`, содержащий элементы управления для отображения заказа, выбранного из списка, связан со свойством `SelectedCourseInfo`.

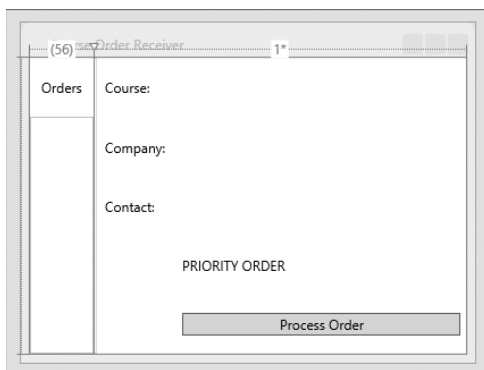


Рис. 47.12. Приложение для получения заказа курсов

Типом свойства `SelectedCourseInfo` является `CourseOrderInfo`. Этот класс создается для определения информации, необходимой дочерним элементам управления этой сетки. `CourseOrderInfo` реализует свойства `Course`, `Company`, `Contact` и т.д. (файл `CourseOrderApplication\CourseOrderReceiver\CourseOrderReceiverWindow.xaml`):

```
<Grid Grid.Column="0">
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="4*" />
  </Grid.RowDefinitions>
  <Label Grid.Row="0" Content="Orders"/>
  <ListBox x:Name="listOrders" Grid.Row="1" ItemsSource="{Binding OrdersList}"
    SelectionChanged="listOrders_SelectionChanged" />
</Grid>
<GridSplitter Grid.Column="1" HorizontalAlignment="Left" Width="3" />
<Grid Grid.Column="1" IsEnabled="True"
  DataContext="{Binding SelectedCourseInfo}" >
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Label Grid.Row="0" Grid.Column="0" Content="Course:"/>
  <Label Grid.Row="1" Grid.Column="0" Content="Company:"/>
  <Label Grid.Row="2" Grid.Column="0" Content="Contact:"/>
  <TextBlock Text="{Binding Course}" Grid.Row="0" Grid.Column="1" />
  <TextBlock Text="{Binding Company}" Grid.Row="1" Grid.Column="1" />
  <TextBlock Text="{Binding Contact}" Grid.Row="2" Grid.Column="1" />
  <TextBlock Grid.Row="3" Grid.Column="1" Text="PRIORITY ORDER"
    Visibility="{Binding HighPriority}"/>
  <Button Grid.Row="4" Grid.Column="1" Content="Process Order"
    IsEnabled="{Binding EnableProcessing}"
    Click="buttonProcessOrder_Click"/>
</Grid>
```

Код класса `CourseOrderInfo` показан ниже. Этот класс является производным от базового класса `BindableBase`, рассмотренного ранее при реализации интерфейса `INotifyPropertyChanged`, и определяет все свойства, необходимые для привяз-

ки данных из кода XAML (файл CourseOrderApplication\CourseOrderReceiver\CourseOrderInfo.cs):

```
using System.Windows;
namespace Wrox.ProCSharp.Messaging
{
    public class CourseOrderInfo : BindableBase
    {
        public CourseOrderInfo()
        {
            Clear();
        }
        private MessageInfo messageInfo;
        public MessageInfo MessageInfo
        {
            get { return messageInfo; }
            set
            {
                SetProperty(ref messageInfo, value);
            }
        }
        private string course;
        public string Course
        {
            get { return course; }
            set
            {
                SetProperty(ref course, value);
            }
        }
        private string company;
        public string Company
        {
            get { return company; }
            set
            {
                SetProperty(ref company, value);
            }
        }
        private string contact;
        public string Contact
        {
            get { return contact; }
            set
            {
                SetProperty(ref contact, value);
            }
        }
        private bool enableProcessing;
        public bool EnableProcessing
        {
            get
            {
                return enableProcessing;
            }
            set
            {
                SetProperty(ref enableProcessing, value);
            }
        }
    }
}
```

```
private Visibility highPriority;
public Visibility HighPriority
{
    get
    {
        return highPriority;
    }
    set
    {
        SetProperty(ref highPriority, value);
    }
}

public void Clear()
{
    Course = string.Empty;
    Company = string.Empty;
    Contact = string.Empty;
    EnableProcessing = false;
    HighPriority = Visibility.Hidden;
}
}
```

В конструкторе класса `Window` по имени `CourseOrderReceiverWindow` создается объект `MessageQueue`, ссылающийся на ту же очередь, что использовалась в приложении для отправки заказов. Для чтения сообщений форматировщик `XmlMessageFormatter` с читаемыми типами ассоциируется с очередью посредством свойства `Formatter`.

Для отображения доступных сообщений в списке создается новая задача, которая читает сообщения в фоновом режиме. Основным методом выполнения задачи является `PeekMessages` (файл `CourseOrderApplication\CourseOrderReceiver\CourseOrderReceiverWindow.xaml.cs`):

```
using System;
using System.Collections.ObjectModel;
using System.Messaging;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Threading;

namespace Wrox.ProCSharp.Messaging
{
    public partial class CourseOrderReceiverWindow : Window
    {
        private MessageQueue ordersQueue;
        private ObservableCollection<MessageInfo> ordersList =
            new ObservableCollection<MessageInfo>();
        private object syncOrdersList = new object();

        public ObservableCollection<MessageInfo> OrdersList
        {
            get
            {
                return ordersList;
            }
        }

        protected override void OnClosed(EventArgs e)
        {
            base.OnClosed(e);
        }
    }
}
```

```

        if (ordersQueue != null)
            ordersQueue.Dispose();
    }

    public CourseOrderReceiverWindow()
    {
        InitializeComponent();
        this.DataContext = this;
        BindingOperations.EnableCollectionSynchronization(ordersList,
            syncOrdersList);

        ordersQueue = new MessageQueue(CourseOrder.CourseOrderQueueName);
        ordersQueue.Formatter = new XmlMessageFormatter(
            new Type[]
            {
                typeof(CourseOrder),
                typeof(Customer),
                typeof(Course)
            });
        // Запуск задачи, заполняющей ListBox заказами.
        Task.Factory.StartNew(PeekMessages);
    }

```

**На заметку!** Более подробно о задачах рассказывалось в главе 21.

В главном методе задачи, `PeekMessages()`, для отображения всех сообщений используется перечислитель очереди сообщений. Внутри цикла `while` перечислитель `messagesEnumerator` проверяет наличие нового сообщения в очереди. Если в очереди нет сообщений, задача в течение 3 часов ожидает поступления следующего сообщения, а по истечении этого времени завершается:

```

private void PeekMessages()
{
    try
    {
        using (MessageEnumerator messagesEnumerator =
            ordersQueue.GetMessageEnumerator2())
        {
            while (messagesEnumerator.MoveNext(TimeSpan.FromHours(3)))
            {
                var messageInfo = new MessageInfo
                {
                    Id = messagesEnumerator.Current.Id,
                    Label = messagesEnumerator.Current.Label
                };
                ordersList.Add(messageInfo);
            }
        }
        MessageBox.Show("No orders in the last 3 hours. Exiting thread",
            "Course Order Receiver", MessageBoxButton.OK,
            MessageBoxImage.Information);
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButton.OK,
            MessageBoxImage.Error);
    }
}

```

Элемент управления `ListBox` содержит элементы класса `MessageInfo`. Этот класс служит для отображения меток сообщений в поле со списком, оставляя скрытым идентификатор каждого сообщения. Идентификатор сообщения может быть использован для последующего чтения сообщения (файл `CourseOrderApplication\CourseOrderReceiver\MessageInfo.cs`):

```
private class MessageInfo
{
    public string Label { get; set; }
    public string Id { get; set; }

    public override string ToString()
    {
        return Label;
    }
}
```

Элемент управления `ListBox` имеет событие `SelectedIndexChanged`, ассоциированное с методом `listOrders_SelectionChanged()`. Этот метод получает объект `LabelIdMapping` из текущего выбора и использует идентификатор для еще одного обращения к сообщению методом `PeekById()`. Затем содержимое сообщения отображается в элементе управления `TextBox`. Поскольку по умолчанию приоритет сообщения не читается, для получения значения `Priority` должно быть установлено соответствующее свойство `MessageReadPropertyFilter` (файл `CourseOrderApplication\CourseOrderReceiver\CourseOrderReceiverWindow.xaml.cs`):

```
private void listOrders_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    var messageInfo = (sender as ListBox).SelectedItem as MessageInfo;
    if (messageInfo == null)
        return;

    ordersQueue.MessageReadPropertyFilter.Priority = true;
    Message message = ordersQueue.PeekById(messageInfo.Id);

    var order = message.Body as CourseOrder;
    if (order != null)
    {
        selectedCourseInfo.MessageInfo = messageInfo;
        selectedCourseInfo.Course = order.Course.Title;
        selectedCourseInfo.Company = order.Customer.Company;
        selectedCourseInfo.Contact = order.Customer.Contact;
        selectedCourseInfo.EnableProcessing = true;

        if (message.Priority > MessagePriority.Normal)
        {
            selectedCourseInfo.HighPriority = Visibility.Visible;
        }
        else
        {
            selectedCourseInfo.HighPriority = Visibility.Hidden;
        }
    }
    else
    {
        MessageBox.Show("The selected item is not a course order",
            "Course Order Receiver", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}
```

Щелчок на кнопке Process Order (Обработать заказ) приводит к вызову метода обработчика OnProcessOrder(). Здесь опять производится обращение к текущему выбранному сообщению в поле со списком, и сообщение удаляется из очереди с помощью метода ReceiveById():

```
private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    Message message = ordersQueue.ReceiveById(
        SelectedCourseInfo.MessageInfo.Id);

    ordersList.Remove(SelectedCourseInfo.MessageInfo);

    listOrders.SelectedIndex = -1;
    selectedCourseInfo.Clear();
    MessageBox.Show("Course order processed", "Course Order Receiver",
        MessageBoxButton.OK, MessageBoxImage.Information);
}
```

На рис. 47.13 показано работающее приложение для приема заказов, отображающее четыре заказа из очереди, один из которых в данный момент выбран.

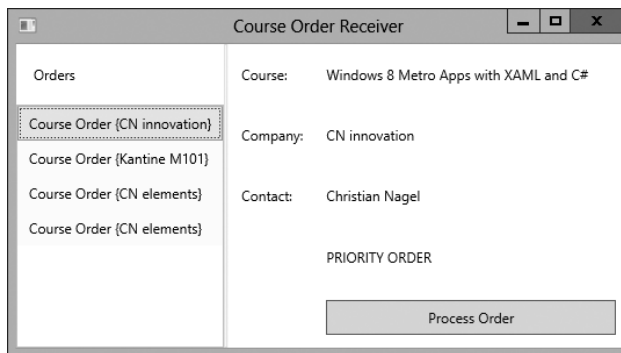


Рис. 47.13. Приложение для приема заказов в работе

## Получение результатов

В представленной версии примера приложение, отправляющее заказы, никогда не узнает, было ли сообщение принято для обработки. Для получения результатов от получателя можно использовать очереди подтверждающих или ответных сообщений.

## Очереди подтверждений

С помощью очереди подтверждений приложение-отправитель может получать информацию о состоянии сообщения. С помощью подтверждений можно определять, нужен ли ответ в ситуации, когда все завершилось успешно или что-то пошло не так. Например, подтверждения могут отправляться в том случае, когда сообщение достигло целевой очереди либо было прочитано, или же когда оно не достигло целевой очереди либо не было прочитано в течение заданного периода времени.

В рассматриваемом примере свойство AdministrationQueue класса Message указывает на очередь CourseOrderAck. Эта очередь должна создаваться подобно нормальной очереди. Однако она действует в противоположном направлении: подтверждения получает исходный отправитель. Для получения подтверждения при прочтении сообщения свойство AcknowledgeType устанавливается в AcknowledgeTypes.FullReceive.

```
var message = new Message(order);
message.AdministrationQueue = new MessageQueue(@".\CourseOrderAck");
message.AcknowledgeType = AcknowledgeTypes.FullReceive;
queue.Send(message, String.Format("Course Order {{0}}",
    order.Customer.Company));
string id = message.Id;
```

*Корреляционный идентификатор* определяет то, какому отправленному сообщению принадлежит каждое подтверждающее сообщение. Каждое отправленное сообщение имеет свой идентификатор, и подтверждающее сообщение, отправленное в ответ на него, содержит идентификатор этого исходного сообщения в виде корреляционного идентификатора. Сообщение из очереди подтверждений может быть прочитано методом `MessageQueue.ReceiveByCorrelationId()` для получения ассоциированного подтверждения.

Вместо подтверждений для писем, которые не прибыли по назначению, можно использовать очередь пропавших писем. Установив свойство `UseDeadLetterQueue` класса `Message` в `true`, можно скопировать сообщение в очередь пропавших писем, если оно не появилось в целевой очереди до истечения заданного таймута.

Таймаут устанавливается с помощью свойств `TimeToReachQueue` и `TimeToBeReceived` класса `Message`.

## Очереди ответов

Если от принимающего приложения необходимо получить больше информации, чем позволяет механизм подтверждений, можно воспользоваться очередью ответов. Очередь ответов подобна обычной очереди, но исходный отправитель применяет ее в качестве получателя, а исходный получатель — в качестве отправителя.

Отправитель должен указать очередь ответов в свойстве `ResponseQueue` класса `Message`. В приведенном ниже примере кода показано, каким образом получатель применяет очередь ответов для возврата ответного сообщения. В этом ответном сообщении `responseMessage` свойство `CorrelationId` устанавливается в идентификатор исходного сообщения. Таким образом, клиентское приложение узнает о том, к какому исходному сообщению относится ответ. Это подобно очередям подтверждений. Ответное сообщение отправляется методом `Send()` объекта `MessageQueue`, возвращенного свойством `ResponseQueue`.

```
public void ReceiveMessage(Message message)
{
    var responseMessage = new Message("response")
    {
        CorrelationId = message.Id
    }
    message.ResponseQueue.Send(responseMessage);
}
```

## Транзакционные очереди

При использовании восстанавливаемых сообщений нет никакой гарантии их доставки в том порядке, в каком они были отправлены, а также в том, что доставка будет однократной. Сбои в сети могут привести к многократной доставке одних и тех же сообщений; это случается и тогда, когда отправитель и получатель имеют несколько установленных протоколов, используемых `Message Queuing`.

Транзакционные очереди должны применяться, когда необходимо гарантировать следующее:

- сообщения должны поступать в том порядке, в каком были отправлены;
- сообщения должны поступать только однократно.

В случае транзакционных очередей одна транзакция не в состоянии охватить и отправку, и прием сообщений. Природа Message Queuing такова, что между отправкой и получением сообщения может пройти довольно длительное время. В отличие от этого, транзакции должны быть кратковременными. В Message Queuing первая транзакция используется для отправки сообщения в очередь, вторая — для передачи его по сети и третья — для получения сообщения.

В следующем примере демонстрируется создание транзакционной очереди сообщений, а также отправка сообщения с применением транзакции.

Транзакционная очередь сообщений создается за счет передачи методу `MessageQueue.Create()` во втором параметре значения `true`.

Если в очередь необходимо записывать сразу несколько сообщений в рамках одной транзакции, то для этого придется создать экземпляр объекта `MessageQueueTransaction` и вызвать его метод `Begin()`. По завершении отправки всех сообщений, относящихся к транзакции, следует вызвать метод `Commit()` того же объекта `MessageQueueTransaction`. Для отмены транзакции (не оставляя никаких сообщений в очереди) должен быть вызван метод `Abort()`, что и делается в блоке `catch`.

```
using System;
using System.Messaging;

namespace Wrox.ProCSharp.Messaging
{
    class Program
    {
        static void Main()
        {
            if (!MessageQueue.Exists(@".\MyTransactionalQueue"))
            {
                MessageQueue.Create(@".\MyTransactionalQueue", true);
            }

            var queue = new MessageQueue(@".\MyTransactionalQueue");
            var transaction = new MessageQueueTransaction();

            try
            {
                transaction.Begin();
                queue.Send("a", transaction);
                queue.Send("b", transaction);
                queue.Send("c", transaction);
                transaction.Commit();
            }
            catch
            {
                transaction.Abort();
            }
        }
    }
}
```

## Использование Message Queuing вместе с WCF

В главе 43 описана архитектура и основные функциональные средства технологии WCF. В WCF можно сконфигурировать привязку Message Queuing, использующую архитектуру подсистемы очередей сообщений Windows Message Queuing. При этом WCF предоставляет уровень абстракции над Message Queuing. Упрощенная архитектура показана на рис. 47.14. Для отправки сообщения в очередь клиентское приложение вызывает метод прокси-объекта WCF. Сообщение создается посредством прокси. Разработчику клиента незачем знать о том, что сообщение отправляется в очередь. Он просто вызывает методы прокси-объекта.

Прокси абстрагирует работу с классами из пространства имен `System.Messaging` и отправляет сообщение в очередь. Прослушиватель канала MSMQ на стороне службы читает сообщения из очереди, преобразует их в вызовы методов и обращается к соответствующим методам службы.

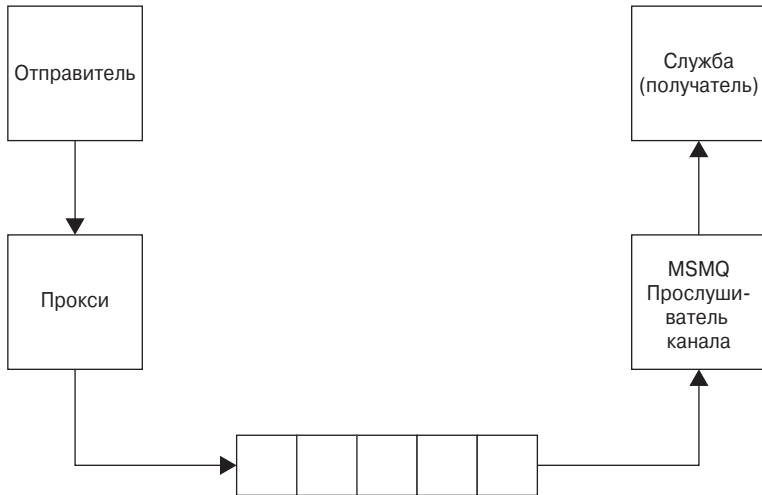


Рис. 47.14. Архитектура Message Queuing вместе с WCF

В следующем примере мы преобразуем приложение заказа курсов для использования Message Queuing с точки зрения WCF. В этом решении модифицированы три созданных ранее проекта, и добавлена одна сборка, включающая контракт службы WCF.

- Библиотека компонентов (`CourseOrder`) включает сущностные классы для сообщений, передаваемых по сети. Эти сущностные классы изменяются для соответствия контракту данных и сериализации с помощью WCF.
- Добавлена новая библиотека (`CourseOrderService`), которая определяет контракт, предоставляемый службой.
- WPF-приложение отправителя (`CourseOrderSender`) изменено так, чтобы вместо отправки сообщений вызывались методы прокси WCF.
- WPF-приложение получателя (`CourseOrderReceiver`) изменено для использования службы WCF, реализующей контракт.

## Классы сущностей с контрактом данных

В библиотеке `CourseOrder` классы `Course`, `Customer` и `CourseOrder` модифицированы с целью применения контракта данных с атрибутами `[DataContract]` и `[DataMember]`. Для использования этих атрибутов необходимо сослаться на сборку `System.Runtime.Serialization` и импортировать пространство имен `System.Runtime.Serialization` (файл `CourseOrderApplicationWCF\CourseOrder\Course.cs`):

```

using System.Runtime.Serialization;
namespace Wrox.ProCSharp.Messaging
{
    [DataContract]
    public class Course
    {

```

```

        [DataMember]
        public string Title { get; set; }
    }
}

```

Класс `Customer` требует также атрибутов контракта данных (файл `CourseOrderApplicationWCF\CourseOrder\Customer.cs`):

```

[DataContract]
public class Customer
{
    [DataMember]
    public string Company { get; set; }

    [DataMember]
    public string Contact { get; set; }
}

```

В классе `CourseOrder` не только добавляются атрибуты контракта данных, но и переопределяется метод `ToString()`, чтобы предоставить стандартное строковое представление этих объектов (файл `CourseOrderApplicationWCF\CourseOrder\CourseOrder.cs`):

```

[DataContract]
public class CourseOrder
{
    [DataMember]
    public Customer Customer { get; set; }

    [DataMember]
    public Course Course { get; set; }

    public override string ToString()
    {
        return String.Format("Course Order {{0}}", Customer.Company);
    }
}

```

## Контракт службы WCF

Для предоставления службы посредством контракта службы WCF добавим библиотеку службы WCF по имени `CourseOrderServiceContract`. Контракт определяется интерфейсом `ICourseOrderService`. Этот контракт нуждается в атрибуте `[ServiceContract]`. Если использование этого интерфейса необходимо ограничить только очередями сообщений, можно применить атрибут `[DeliveryRequirements]` и присвоить значение свойству `QueuedDeliveryRequirements`. Возможными значениями перечисления `QueuedDeliveryRequirementsMode` являются `Required`, `Allowed` и `NotAllowed`. Метод `AddCourseOrder` предоставляется службой. Методы, используемые `Message Queuing`, могут иметь только входные параметры. Поскольку отправитель и получатель могут выполняться независимо друг от друга, отправитель не может рассчитывать на немедленное получение результата. В атрибуте `[OperationContract]` устанавливается свойство `IsOneWay`. Клиент, вызвавший эту операцию, не ожидает ответа от службы (файл `CourseOrderApplicationWCF\CourseOrderServiceContract\ICourseOrderService.cs`):

```

using System.ServiceModel;

namespace Wrox.ProCSharp.Messaging
{
    [ServiceContract]
    [DeliveryRequirements(
        QueuedDeliveryRequirements=QueuedDeliveryRequirementsMode.Required)]

```

```

public interface ICourseOrderService
{
    [OperationContract(IsOneWay = true)]
    void AddCourseOrder(CourseOrder courseOrder);
}

```

**На заметку!** Для получения ответа на стороне клиента можно применять очереди подтверждений и ответов.

## Приложение для получения сообщений, использующее WCF

Изменим WPF-приложение `CourseOrderReceiver` для реализации службы WCF и приема сообщений. Ему необходимы ссылки на сборку `System.ServiceModel` и сборку контракта WCF `CourseOrderServiceContract`.

Класс `CourseOrderService` реализует интерфейс `ICourseOrderService`. В этой реализации инициируется событие `CourseOrderAdded`. WPF-приложение регистрирует это событие для получения объектов `CourseOrder`.

Поскольку элементы управления WPF привязаны к единственному потоку, с помощью атрибута `[ServiceBehavior]` устанавливается свойство `UseSynchronizationContext`. Передача вызова метода в поток, определенный контекстом синхронизации WPF-приложения, является функцией исполняющей среды WCF (файл `CourseOrderApplicationWCF\CourseOrderReceiver\CourseOrderService.cs`):

```

using System.ServiceModel;
namespace Wrox.ProCSharp.Messaging
{
    [ServiceBehavior(UseSynchronizationContext=true)]
    public class CourseOrderService: ICourseOrderService
    {
        public static event EventHandler<CourseOrderEventArgs>
            CourseOrderAdded;

        public void AddCourseOrder(CourseOrder courseOrder)
        {
            var courseOrderAdded = CourseOrderAdded;
            if (courseOrderAdded != null)
            {
                courseOrderAdded(this, new CourseOrderEventArgs(courseOrder));
            }
        }
    }

    public class CourseOrderEventArgs : EventArgs
    {
        public CourseOrderEventArgs(CourseOrder courseOrder)
        {
            this.CourseOrder = courseOrder;
        }
        public CourseOrder CourseOrder { get; private set; }
    }
}

```

**На заметку!** Контекст синхронизации описан в главе 21.

В конструкторе класса `CourseReceiverWindow` создается и открывается для запуска прослушателя экземпляр объекта `ServiceHost`. Привязка прослушателя осуществляется в конфигурационном файле приложения.

В конструкторе выполняется подписка на события `CourseOrderAdded` и `CourseOrderService`. Поскольку единственное, что здесь происходит — это добавление объекта `CourseOrder` в коллекцию, воспользуемся простым лямбда-выражением.

**На заметку!** Лямбда-выражения подробно рассматривались в главе 8.

Здесь используется класс коллекции `ObservableCollection<T>` из пространства имен `System.Collections.ObjectModel`. Вне реализован интерфейс `INotifyCollectionChanged` и потому элементы управления WPF, привязанные к коллекции, информированы о динамических изменениях в списке (файл `CourseOrderApplicationWCF\CourseOrderReceiver\CourseOrderReceiverWindow.xaml.cs`):

```
using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.Windows;

namespace Wrox.ProCSharp.Messaging
{
    public partial class CourseOrderReceiverWindow : Window
    {
        private ObservableCollection<CourseOrder> courseOrders =
            new ObservableCollection<CourseOrder>();
        public CourseOrderReceiverWindow()
        {
            InitializeComponent();
            this.DataContext = courseOrders;
            CourseOrderService.CourseOrderAdded += (sender, e) =>
            {
                courseOrders.Add(e.CourseOrder);
                buttonProcessOrder.IsEnabled = true;
            };
            var host = new ServiceHost(typeof(CourseOrderService));
            try
            {
                host.Open();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

Теперь привязка данных элементов WPF в коде XAML изменена так, чтобы использовалась новая коллекция. Элемент `ListBox` привязан к контексту данных, а простые элементы управления — к свойствам текущего элемента в контексте данных (файл `CourseOrderApplicationWCF\CourseOrderReceiver\CourseOrderReceiverWindow.xaml`):

```
<ListBox x:Name="listOrders" Grid.Row="1" ItemsSource="{Binding}"
        IsSynchronizedWithCurrentItem="True" />
<!-- ... -->

<TextBlock Text="{Binding Course.Title}" Grid.Row="0" Grid.Column="1" />
<TextBlock Text="{Binding Customer.Company}" Grid.Row="1" Grid.Column="1" />
<TextBlock Text="{Binding Customer.Contact}" Grid.Row="2" Grid.Column="1" />
```

В конфигурационном файле приложения определена привязка `netMsmqBinding`. Для надежного обмена сообщениями требуются транзакционные очереди. Чтобы можно было принимать и отправлять сообщения в транзакционные очереди, свойство `exactlyOnce` должно быть установлено в `false` (файл `CourseOrderApplicationWCF\CourseOrderReceiver\App.config`):

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/>
  </startup>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="NonTransactionalQueueBinding" exactlyOnce="false">
          <security mode="None" />
        </binding>
      </netMsmqBinding>
    </bindings>
    <services>
      <service name="Wrox.ProCSharp.Messaging.CourseOrderService">
        <endpoint address="net.msmq://localhost/private/courseorder"
          binding="netMsmqBinding"
          bindingConfiguration="NonTransactionalQueueBinding"
          name="OrderQueueEP"
          contract="Wrox.ProCSharp.Messaging.ICourseOrderService" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Обработчик события Click кнопки `buttonProcessOrder` удаляет выбранный заказ курсов из класса коллекции (файл `CourseOrderApplicationWCF\CourseOrderReceiver\CourseOrderReceiverWindow.xaml.cs`):

```
private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    var courseOrder = listOrders.SelectedItem as CourseOrder;
    courseOrders.Remove(courseOrder);
    listOrders.SelectedIndex = -1;
    buttonProcessOrder.IsEnabled = false;

    MessageBox.Show("Course order processed", "Course Order Receiver",
        MessageBoxButton.OK, MessageBoxImage.Information);
}
```

## Приложение для отправки сообщений, использующее WCF

Приложение-отправитель также модифицировано для использования класса прокси WCF. Для контракта службы включена ссылка на сборку `CourseOrderServiceContract`, а сборка `System.ServiceModel` необходима для работы с классами WCF.

В обработчике событий Click элемента управления `buttonSubmit` класс `ChannelFactory` возвращает прокси. Прокси отправляет сообщение в очередь, вызывая метод `AddCourseOrder()` (файл `CourseOrderApplicationWCF\CourseOrderSender\CourseOrderWindow.xaml.cs`):

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
```

```
var factory = new ChannelFactory<ICourseOrderService>("queueEndpoint");
ICourseOrderService proxy = factory.CreateChannel();
proxy.AddCourseOrder(CourseOrder);
factory.Close();

MessageBox.Show("Course Order submitted", "Course Order",
    MessageBoxButtons.OK, MessageBoxIcon.Information);
}
catch (MessageQueueException ex)
{
    MessageBox.Show(ex.Message, "Course Order Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
}
```

В конфигурационном файле приложения определена клиентская часть соединения WCF. Как и ранее, используется привязка `netMsmqBinding` (файл `CourseOrderApplicationWCF\CourseOrderSender\App.config`):

```
<?xml version="1.0"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/>
  </startup>
  <system.serviceModel>
    <bindings>
      <netMsmqBinding>
        <binding name="nonTransactionalQueueBinding" exactlyOnce="false">
          <security mode="None" />
        </binding>
      </netMsmqBinding>
    </bindings>
    <client>
      <endpoint address="net.msmq://localhost/private/courseorder"
        binding="netMsmqBinding"
        bindingConfiguration="nonTransactionalQueueBinding"
        contract="Wrox.ProCSharp.Messaging.ICourseOrderService"
        name="queueEndpoint"
        kind="" endpointConfiguration="" />
    </client>
  </system.serviceModel>
</configuration>
```

Если теперь запустить приложение, оно будет работать так же, как и раньше. Больше нет необходимости применять классы из пространства имен `System.Messaging` для отправки и получения сообщений. Вместо этого приложение разрабатывается таким же образом, как и при использовании каналов TCP или HTTP вместе с WCF.

Тем не менее, для создания очередей сообщений и удаления сообщений все-таки нужен класс `MessageQueue`. В данном случае WCF — просто абстракция для процесса отправки и получения сообщений.

**На заметку!** Если необходимо обеспечить взаимодействие приложения `System.Messaging` с приложением WCF, вместо привязки `netMsmqBinding` понадобится применить `msmqIntegrationBinding`. Эта привязка использует формат сообщения, используемый в COM и `System.Messaging`.

## Установка очередей сообщений

Очереди сообщений могут быть созданы вызовом метода `MessageQueue.Create()`. Однако пользователь, запускающий приложение, как правило, не имеет административных привилегий, необходимых для создания очередей сообщений.

Обычно очереди сообщений создаются с помощью программы установки, предусматривающей использование класса `MessageQueueInstaller`. Если класс установщика является частью приложения, то утилита командной строки `installutil.exe` (или Windows Installation Package) вызывает метод `Install()` установщика.

В Visual Studio имеется специальная поддержка применения `MessageQueueInstaller` с приложениями Windows Forms. После перетаскивания компонента `MessageQueue` из панели инструментов на форму смарт-тег этого компонента позволяет добавить установщик через пункт меню `Add Installer` (Добавить установщик). Объект `MessageQueueInstaller` может быть сконфигурирован в редакторе свойств для определения транзакционных очередей, журнальных очередей, типа форматировщика, базового приоритета и т.д.

**На заметку!** Установщики более подробно рассматриваются в главе 18.

## Резюме

В этой главе было показано, как применять технологию `Message Queuing`. Это очень важная технология, которая позволяет налаживать не только асинхронные, но и отключенные коммуникации. Отправитель и получатель могут запускаться в разное время, что делает `Message Queuing` удобным вариантом для создания интеллектуальных клиентов, а также полезным средством для распределения нагрузки на сервер во времени.

Наиболее важными классами в `Message Queuing` являются `Message` и `MessageQueue`. Класс `MessageQueue` позволяет отправлять, принимать и считывать сообщения, а класс `Message` — определять подлежащее отправке содержимое.

В WCF предлагается абстракция для `Message Queuing`. Это дает возможность применять предлагаемые в WCF концепции для отправки сообщений за счет вызова методов прокси-класса и получения сообщений путем реализации службы.