



46

Технология Message Queuing

В ЭТОЙ ГЛАВЕ...

- Краткий обзор Message Queuing
- Архитектура Message Queuing
- Административные инструменты для работы с Message Queuing
- Программирование с использованием Message Queuing
- Пример приложения заказа курсов
- Использование Message Queuing вместе с WCF

В пространстве имен `System.Messaging` определены классы, которые позволяют выполнять чтение и запись сообщений с использованием такого предлагаемого в составе операционной системы Windows средства для организации сообщений, как `Message Queuing`. Обмен сообщениями может применяться в сценарии автономной работы, в 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 также показано, как организовать обработку поступающих от сервера сообщений с помощью очередей подтверждающих (acknowledgment) и ответных (response) сообщений и использовать `Message Queuing` с привязками к очередям сообщений WCF.

Краткий обзор

Прежде чем углубляться в детали программирования с использованием `Message Queuing`, которым посвящена остальная часть главы, в данном разделе предлагается ознакомиться с основными концепциями организации очередей сообщений и сравнить их с концепциями синхронного и асинхронного программирования. При синхронном программировании, когда вызывается метод, то вызвавший его код должен ожидать, пока метод не завершит свою работу. 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 `System.Net` и `System.IO`), либо специальных потоков (см. главу 20). Как при синхронном, так и при асинхронном программировании клиент и сервер должны работать в одно и то же время.

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

Подключенное и отключенное программирование 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 послание будет прочитано. Люди, использующие эту технологию, работают в автономном (отключенном) режиме. Конечно, может случиться так, что почта не будет прочитана никогда, а просто проигнорирована. Такова природа отключенных коммуникаций. Чтобы избежать этой проблемы, можно запросить ответ или 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`.

`Message Queuing`, по сути, 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, использующим память, 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 (автономной) среде. На рис. 46.1 показан очень простой способ использования сообщений. Отправитель посылает сообщения в очередь сообщений, а получатель принимает их из этой очереди.

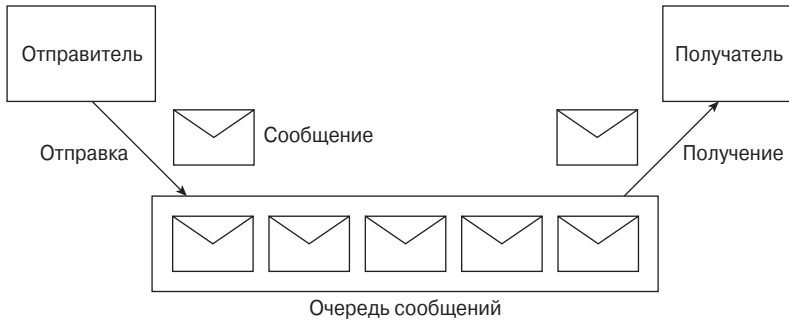


Рис. 46.1. Простой способ использования сообщений

Когда следует использовать Message Queuing

Одной 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 на клиентской системе (рис. 46.2). Как только коммивояжер возвращается в свой офис, заказ автоматически передается из очереди сообщений клиентской системы в очередь сообщений целевой системы, где и обрабатывается.

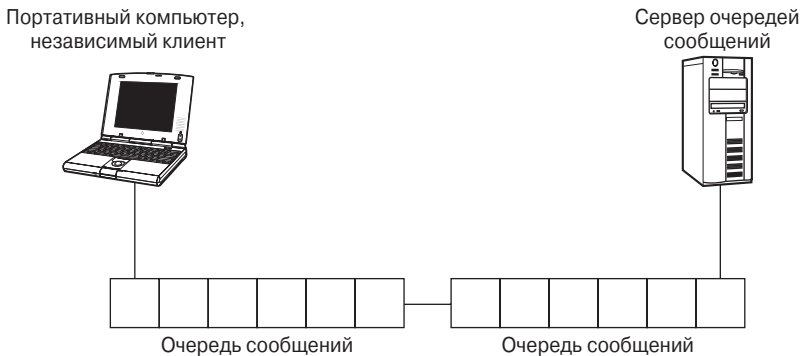


Рис. 46.2. Отключенная среда и Message Queuing

Помимо портативного компьютера, коммивояжер может использовать устройство Pocket Windows, где также доступно Message Queuing.

Технология Message Queuing может быть полезна и в подключенной среде. Представьте сайт электронной коммерции (рис. 46.3), где в определенные периоды времени сервер полностью загружен обработкой заказов, например, в ранний вечер и в выходные, при этом 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 базы данных. Таким образом, нагрузка на систему сглаживается по времени, так что сервер, обрабатывающий транзакции, может быть дешевле и не требовать модернизации.

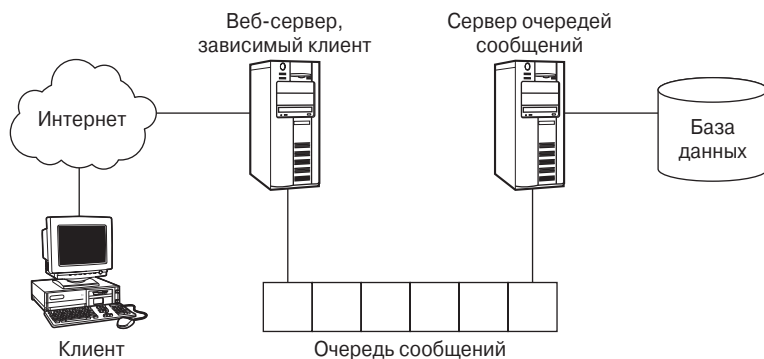


Рис. 46.3. Сайт электронной коммерции и Message Queuing

Функциональные возможности Message Queuing

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

- Сообщения могут пересылаться в автономной 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 экспресс-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 многоадресных (multicast) сообщений.
- В Message Queuing 4.0 поддерживается возможность распознавания вредоносных сообщений. Для таких сообщений может быть определена специальная очередь. Например, в случае, если после прочтения сообщения из обычной очереди, далее оно должно вставляться в базу данных, но по какой-то причине этого не происходит, это 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 преобразовать.
- 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.)



Из-за того, что Message Queuing является частью операционной системы, установить версию Message Queuing 5.0 в системе Windows XP или Windows Server 2003 не получится. Эта версия входит в состав ОС Windows Server 2008 R2 и Windows 7.

В остальной части главы показано, как пользоваться всеми этими функциональными возможностями.

Продукты Message Queuing

Версия Message Queuing 5.0 поставляется в составе Windows 7 и Windows Server 2008 R2. В Windows 2000 входила версия Message Queuing 2.0, в которой не было поддержки ни протокола HTTP, ни многоадресных сообщений. Версия Message Queuing 3.0 поставлялась в составе Windows XP и Windows Server 2003, а версия Message Queuing 4.0 – в составе Windows Vista и Windows Server 2003.

При использовании ссылки Turn Windows Features on or off (Включение или отключение компонентов Windows), которая предлагается в Windows 7 в окне Configuring Programs and Features (Программы и компоненты), 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 MSMQ 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 многоадресной рассылки).** Позволяет отправлять сообщения группам серверов.
- **MSMQ DCOM Proxy (Прокси MSMQ DCOM).** С помощью DCOM-прокси система может подключаться к удаленному серверу, используя API-интерфейс DCOM.

После установки Message Queuing в системе должна быть обязательно запущена служба Message Queuing (рис. 46.4). 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.

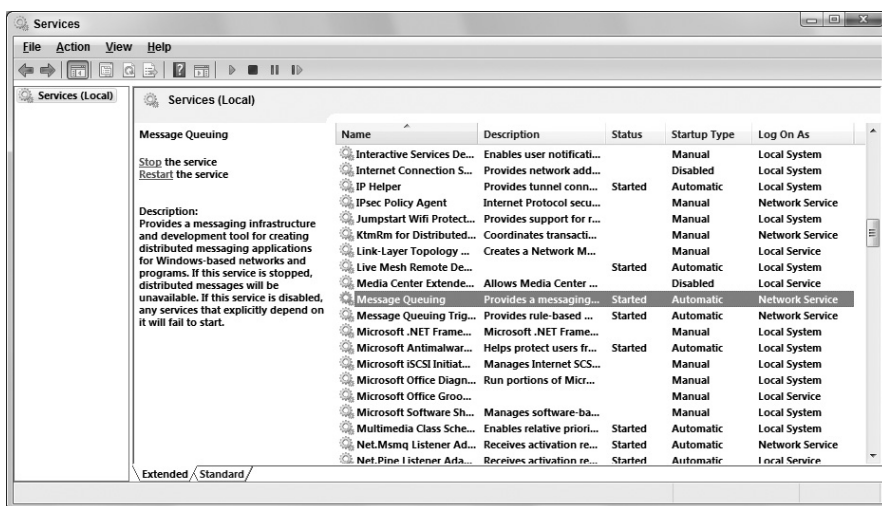


Рис. 46.4. Служба Message Queuing в запущенном состоянии

Сообщения

Сообщение посылается в очередь сообщений. Сообщение включает тело, содержащее пересылаемые данные, и метку — заголовок сообщения. В тело сообщения может быть помещена любая информация. В .NET имеется набор форматировщиков, преобразующих данные, которые помещаются в тело. Помимо метки и тела сообщение включает дополнительную информацию об отправителе, конфигурацию таймаута, идентификатор транзакции или приоритет.

Очереди сообщений содержат сообщения нескольких типов.

- *Нормальное сообщение* отправляется любым приложением.
- *Подтверждающее сообщение* (acknowledgment message) 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 то сообщение, которое имеет наивысший приоритет.

Сообщения имеют два режима доставки: *экспресс* (express) и *восстановимый* (recoverable). 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 сети.

Транзакционные сообщения — это специальная версия восстанавливаемых 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`.

Общедоступные или частные очереди обычно применяются для отправки сообщений, но существуют также и другие типы очередей.

- *Общедоступная (public) 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 общедоступной очереди можно обращаться, не зная имени компьютера, на котором она расположена. Такую очередь можно переместить с одной системы на другую и клиент этого не заметит. В среде рабочей группы (Workgroup) невозможно создавать общедоступные очереди, потому что им нужна служба Active Directory. Более подробную информацию о службе Active Directory можно найти в главе 52.
- *Частные (private) очереди* не публикуются в Active Directory. Эти очереди доступны, только когда известны их полные путевые имена. Частные очереди могут использоваться в среде Workgroup.
- *Журнальные (journal) очереди* служат для хранения копий сообщений после того, как они были получены или отправлены. Включение протоколирования для общедоступной или частной 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 мертвых писем (dead-letter)* 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 мертвых писем можно проверять для обнаружения не доставленных 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 требуется нечто большее, чем простое подтверждение о факте отправки в качестве ответа со стороны получателя. Принимающее приложение может посылать ответные сообщения обратно исходному отправителю.
- *Очередь отчетов* используется для тестовых сообщений. Очереди отчетов могут быть созданы изменением типа (или категории) общедоступной или частной 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 частными 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, в настоящем разделе следует ознакомиться с административными инструментами, которые поставляются в составе операционной системы Windows для создания и управления очередями и сообщениями.



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 системе Windows 7 оснастку Computer Management можно запустить, выбрав в меню Start (Пуск) пункт Control Panel⇒Administrative Tools⇒Computer Management (Панель управления⇒Администрирование⇒Управление компьютером). В панели древовидного представления Message Queuing находится ниже элемента Services and Applications (Службы и приложения). Выбрав Private Queues (Частные очереди) или Public Queues (Общедоступные очереди), можно создать новую очередь из меню Action (Действие), как показано на рис. 46.5. С общедоступными очередями можно работать, только если Message Queuing сконфигурирована в режиме Active Directory.

Свойства Message Queuing

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. 46.6.

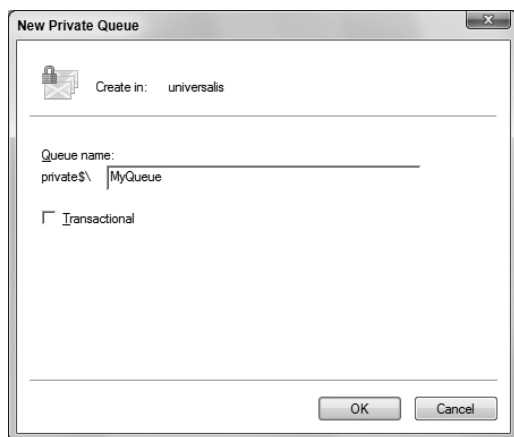


Рис. 46.5. Создание новой очереди

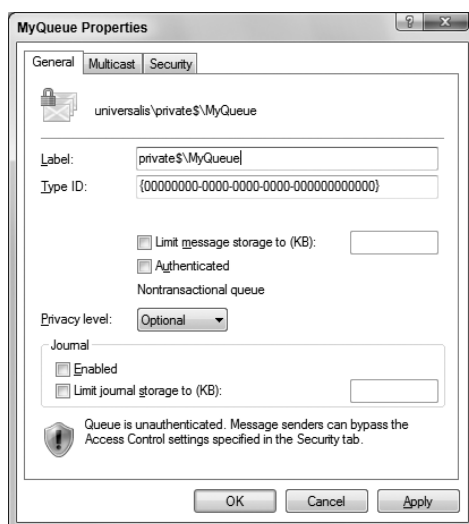


Рис. 46.6. Окно свойств очереди

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}, для отображения множественных очередей на единственную категорию типа.

Отчетные очереди, как было сказано ранее, используют специфический идентификатор типа. Type ID представляет собой универсальный уникальный идентификатор (UUID) или глобально уникальный идентификатор (GUID).



Специальные идентификаторы типа могут быть созданы утилитами `uuidgen.exe` или `guidgen.exe`. Утилита `uuidgen.exe` — это инструмент командной строки, служащий для создания уникальных идентификаторов, а `guidgen.exe` — графическая ее версия для создания UUID.

- Во избежание переполнения диска, максимальный размер всех сообщений (Limit message storage to (KB)) в очереди может ограничиваться.
- Отмеченный флажок **Authenticated** (Аутентифицированы) 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 значение по умолчанию **Optional** — принимаются те и другие.
- Целевое протоколирование может быть сконфигурировано опцией **Journal** (Журнал). С помощью этой опции обеспечивается сохранение в журнале копий принятых сообщений. Для журнальных сообщений очереди может быть указан максимальный размер занятого дискового пространства. По достижении максимального размера целевой журнал очищается.
- При 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 архитектура Message Queuing известна, 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 создаются очереди сообщений 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 частную 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.



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

Фрагмент кода `CreateMessageQueue\Program.cs`



Для создания очереди необходимы административные привилегии. Обычно нельзя рассчитывать, что пользователь приложения будет иметь их. Именно поэтому очереди обычно создаются программой установки. Далее в этой главе будет показано, как создавать очереди сообщений с помощью класса `MessageQueueInstaller`.

Нахождение очереди

Для идентификации очередей могут использоваться путевое имя и форматное имя. При поиске очереди необходимо делать различия между общедоступными и частными очередями. Общедоступные очереди публикуются в `Active Directory`. Для таких очередей не обязательно знать систему, на которой они расположены. Частные очереди могут быть найдены только в случае, если известно имя системы, на которой расположена очередь.

Общедоступные очереди в домене `Active Directory` можно искать по метке очереди, категории или форматного имени. Можно также получить все очереди, имеющиеся на машине. В классе `MessageQueue` предусмотрены статические методы для поиска: `GetPublicQueuesByLabel()`, `GetPublicQueuesByCategory()` и `GetPublicQueuesByMachine()`. Метод `GetPublicQueues()` возвращает массив всех общедоступных очередей в домене.

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

Фрагмент кода *FindQueues\Program.cs*

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 времени, а также учитывать категорию, метку или имя машины.

Частные очереди можно искать статическим методом `GetPrivateQueuesByMachine()`. Этот метод возвращает все частные 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 конструктором класса `MessageQueue`.

Путевое имя

Путь указывает имя машины и имя очереди для ее открытия. В следующем примере кода открывается очередь `MyPublicQueue` на локальном хосте. Чтобы удостовериться в существовании очереди, применяется статический метод `MessageQueue.Exists()`.

```
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");
                // Очередь .\MyPublicQueue не существует
            }
        }
    }
}
```

Фрагмент кода *OpenQueue\Program.cs*

В зависимости от типа очереди при открытии очередей должны использоваться разные идентификаторы. В табл. 46.1 показан синтаксис идентификаторов для очередей разных типов.

Таблица 46.1. Синтаксис идентификаторов для очередей разных типов

| Тип очереди | Синтаксис |
|---|---|
| Общедоступная очередь | <i>ИмяКомпьютера\ИмяОчереди</i> |
| Частная очередь | <i>ИмяКомпьютера\Private\$\ИмяОчереди</i> |
| Журнальная очередь | <i>ИмяКомпьютера\ИмяОчереди\Journal\$</i> |
| Журнальная очередь машины | <i>ИмяКомпьютера\Journal\$</i> |
| Очередь “мертвых писем” машины | <i>ИмяКомпьютера\DeadLetter\$</i> |
| Транзакционная очередь “мертвых писем” машины | <i>ИмяКомпьютера\XactDeadLetter\$</i> |

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

Форматное имя

Вместо использования 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 автономной среде, где очередь во время отправки сообщения недоступна, необходимо использовать форматное имя:

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

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

- Для доступа к частной очереди строка, передаваемая конструктору, выглядит так: `FormatName:PRIVATE=MachineGUID\QueueNumber`. Номер `QueueNumber` для частных очередей генерируется при их создании. Номера очередей можно просмотреть в каталоге `<windows>\System32\msmq\storage\lqs`.
- В `FormatName:DIRECT=Protocol:MachineAddress\QueueName` можно указать протокол, который должен использоваться для отправки сообщений. Протокол HTTP поддерживается, начиная с версии Message Queuing 3.0.
- `FormatName:DIRECT=OS:MachineName\QueueName` — другой способ задания очереди с применением форматного имени. Таким образом, протокол указывать не нужно, а только имя машины и форматное имя.

Отправка сообщения

Для отправки сообщения в очередь используется метод `Send` класса `MessageQueue`. Объект, переданный в качестве аргумента методу `Send()`, сериализуется в ассоциированную очередь. Метод `Send()` перегружен так, что можно передавать метку и объект `MessageQueueTransaction`. Транзакционное поведение Message Queuing обсуждается позже.

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

Путевое имя специфицирует точку вместо имени сервера, что означает локальную систему. Путь имена частных очередей работают только локально.

```

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);
            }
        }
    }
}

```

Фрагмент кода *SendMessage\Program.cs*

На рис. 46.7 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. 46.8, можно увидеть, что сообщение сформатировано с использованием XML. Способ форматирования сообщения — функция форматировщика, ассоциированного с очередью сообщений.

Форматировщик сообщений

Формат, в котором передаются сообщения в очередь, зависит от используемого форматировщика. Класс `MessageQueue` имеет свойство `Formatter`, через которое очереди может быть назначен объект-форматировщик.

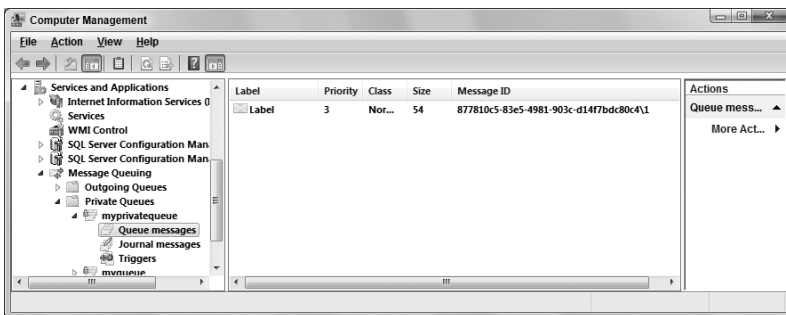


Рис. 46.7. Просмотр сообщений в очереди в оснастке Computer Management

Форматировщик по умолчанию — `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` — форматировщик по умолчанию. Он сериализует объекты, используя XML. Подробнее об XML-форматировании читайте в главе 33.
- С помощью форматировщика `BinaryMessageFormatter` сообщения сериализуются в двоичный формат. Эти сообщения короче, чем сформатированные с применением XML.
- `ActiveXMessageFormatter` — двоичный форматировщик, так что сообщения могут быть прочитаны и записаны 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. 46.8 в формате XML, на рис. 46.9 сформатировано с помощью `BinaryMessageFormatter`.

Отправка сложных сообщений

Вместо строк метода `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 системой .NET сериализуются все поля (включая приватные). Специальная сериализация может быть определена за счет реализации интерфейса `ISerializable`. Дополнительные сведения о сериализации времени выполнения .NET предлагаются в главе 29.

Сериализация XML происходит с помощью XML-форматировщика. При сериализации XML сериализуются все общедоступные поля и свойства.

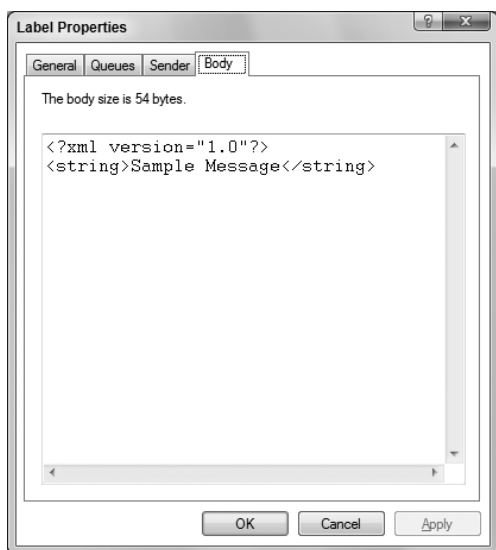


Рис. 46.8. Просмотр тела сообщения в формате XML

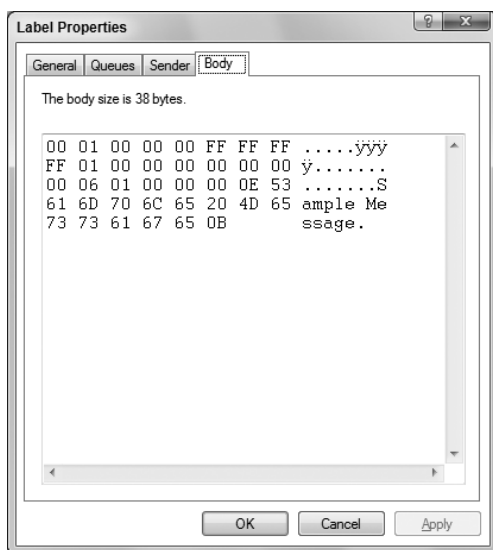


Рис. 46.9. Просмотр сообщения, сформатированного `BinaryMessageFormatter`

На сериализацию XML могут оказывать влияние атрибуты из пространства имен `System.Xml.Serialization`. Более подробно сериализация XML описана в главе 28.

Прием сообщений

Для чтения сообщений можно использовать класс `MessageQueue`. Метод `Receive()` читает единственное 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, потому что порядок сообщений в сети не гарантируется. Для получения гарантированного порядка применяйте транзакционные очереди сообщений.

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

Сообщение читается методом `Receive()` и затем тело сообщения выводится на консоль:

```
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);
        }
    }
}
```

Фрагмент кода *SendMessage\Program.cs*

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

Перечисление сообщений

Вместо чтения сообщений друг за другом с помощью метода `Receive()` можно применить перечислитель для прохождения сразу по всем сообщениям в очереди. Класс `MessageQueue` реализует интерфейс `IEnumerable`, и потому может быть использован в операторе `foreach`. Здесь сообщения не удаляются из очереди, вы только заглядываете в каждое сообщение для извлечения 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`. Класс `MessageEnumerator` реализует интерфейс `IEnumerator`, но имеет несколько дополнитель-

ных свойств. С интерфейсом `IEnumerator` сообщения не удаляются из очереди. Метод `RemoveCurrent()` класса `MessageEnumerator` удаляет сообщение из текущей позиции курсора перечислителя. В этом примере используется метод `GetMessageEnumerator()` класса `MessageQueue` 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);
    }
}
```

Чтение асинхронным образом

Метод `Receive()` класса `MessageQueue` ожидает, пока не будет прочитано сообщение из очереди. Чтобы избежать 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`, который ссылается на метод, вызываемый после появления сообщения в очереди, которое может быть прочитано. В этом примере метод `MessageArrived` передается делегату `ReceivedCompletedEventHandler`:

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

Фрагмент кода *ReceiveMessageAsync\Program.cs*

Метод-обработчик `MessageArrived` принимает два параметра. Первый параметр — источник события, `MessageQueue`.

Второй параметр 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, методы `BeginPeek()` и `EndPeek()` могут применяться для асинхронного ввода-вывода.

Приложение заказа курсов

Чтобы продемонстрировать использование Message Queuing, в этом разделе мы создадим простое приложение заказа курсов. Этот пример приложения 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 помещаются в отдельную сборку. Сборка CourseOrder включает три сущностных класса: CourseOrder, Course и Customer. В этом примере приложения реализованы не все свойства, которые могут присутствовать в реальном приложении, а лишь столько, сколько достаточно для демонстрации концепций.

В файле Course.cs определен класс Course. Этот класс имеет лишь одно свойство для названия курса:

```
namespace Wrox.ProCSharp.Messaging
{
    public class Course
    {
        public string Title { get; set; }
    }
}
```

Фрагмент кода *CourseOrder\Course.cs*

Файл Customer.cs включает класс Customer, в котором имеются свойства для компании и контактного имени:

```
namespace Wrox.ProCSharp.Messaging
{
    public class Customer
    {
        public string Company { get; set; }
        public string Contact { get; set; }
    }
}
```

Фрагмент кода *CourseOrder\Customer.cs*

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 общедоступной очереди. Форматное имя используется для отправки сообщения, даже если в текущий момент добраться до очереди невозможно. Получить форматное имя можно, прочитав идентификатор очереди сообщений с помощью оснастки Computer Management. Если доступ к Active Directory для создания общедоступной очереди не требуется, этот код легко изменить так, чтобы в нем применялась и частная очередь:

```
namespace Wrox.ProCSharp.Messaging
{
    public class CourseOrder
    {
        public const string CourseOrderQueueName =
            "FormatName:Public=D99CE5F3-4282-4a97-93EE-E9558B15EB13";
    }
}
```

```

    public Customer Customer { get; set; }
    public Course Course { get; set; }
}
}

```

Фрагмент кода *CourseOrder\CourseOrder.cs*

Отправитель сообщения о заказе курса

Вторая часть решения представлена приложением Windows по имени *CourseOrderSender*. Это приложение отправляет заказы курсов в очередь сообщений. Должны присутствовать ссылки на пространства имен *System.Messaging* и *CourseOrder*.

Пользовательский интерфейс этого приложения показан на рис. 46.10. Элементы комбинированного списка *comboBoxCourses* включают несколько курсов, таких как “Advanced .NET Programming”, “Programming with LINQ” и “Distributed Application Development using WCF”.

В результате щелчка на кнопке *Submit the Order* (Отправить заказ) вызывается метод-обработчик *buttonSubmit_Click()*. Этот метод создает объект *CourseOrder* и заполняет его содержимым элементов управления *TextBox* и *ComboBox*. Затем создается экземпляр *MessageQueue* для открытия общедоступной очереди с форматным именем. С помощью метода *Send()* объект *CourseOrder* передается для сериализации форматировщиком по умолчанию *XmlMessageFormatter* и записи в очередь.



Рис. 46.10. Приложение для отправки заказа курсов

```

private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var order = new CourseOrder();
        order.Course = new Course()
        {
            Title = comboBoxCourses.SelectedItem.ToString()
        };

        order.Customer = new Customer()
        {
            Company = textCompany.Text,
            Contact = textContact.Text
        };

        using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
        {
            queue.Send(order, String.Format("Course Order {{0}}", order.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);
    }
}

```

Фрагмент кода *CourseOrderSender\CourseOrderWindow.xaml.cs*

Отправка приоритетных и восстанавливаемых сообщений

Сообщения могут быть снабжены приоритетом за счет установки свойства `Priority` класса `Message`. Если сообщения специально сконфигурированы, то должен быть создан объект `Message`, и тело сообщения передано его конструктору.

В этом примере приоритет устанавливается в `MessagePriority.High`, если отмечен флажок `checkBoxPriority`. Перечисление `MessagePriority` позволяет устанавливать значения от `Lowest (0)` до `Highest (7)`. Значение по умолчанию, `Normal`, соответствует величине приоритета 3. Чтобы сделать сообщение восстанавливаемым, понадобится установить свойство `Recoverable` в `true`:

```
private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var order = new CourseOrder
        {
            Course = new Course
            {
                Title = comboBoxCourses.Text
            },
            Customer = new Customer
            {
                Company = textCompany.Text,
                Contact = textContact.Text
            }
        };
        using (var queue = new MessageQueue(CourseOrder.CourseOrderQueueName))
        using (var message = new Message(order))
        {
            if (checkBoxPriority.IsChecked == true)
            {
                message.Priority = MessagePriority.High;
            }
            message.Recoverable = true;
            queue.Send(message, String.Format("Course Order {{{0}}}", order.Customer.Company));
        }
        MessageBox.Show("Course Order submitted");
    }
    catch (MessageQueueException ex)
    {
        MessageBox.Show(ex.Message, "Course Order Error",
            MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```

Фрагмент кода *CourseOrderSender\CourseOrderWindow.xaml.cs*

Запустив это приложение, можно добавлять заказы курсов в очередь сообщений (рис. 46.11).

Получатель сообщений о заказе курсов

Представление визуального конструктора приложения `Course Order Receiver` (Получатель заказов курсов), которое читает сообщения из очереди, показано на рис. 46.12. В этом приложении отображаются метки каждого заказа в списке `listOrders`. Когда заказ выбран, его содержимое отображается в элементах управления в правой части окна приложения.

В конструкторе класса `Window` по имени `CourseOrderReceiverWindow` создается объект `MessageQueue`, ссылающийся на ту же очередь, что была использована в приложении, отправляющем заказы. Для чтения сообщений форматировщик `XmlMessageFormatter` с читаемыми типами ассоциируется с очередью через свойство `Formatter`.

Рис. 46.11. Добавление заказа курсов

Рис. 46.12. Приложение Course Order Receiver

Для отображения доступных сообщений в списке создается новая задача, которая читает сообщения в фоновом режиме. Главным методом этой задачи является `PeekMessages`.



Более подробно о задачах рассказывалось в главе 20.

```
using System;
using System.Messaging;
using System.Threading;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;
namespace Wrox.ProCSharp.Messaging
{
    public partial class CourseOrderReceiverWindow: Window
    {
        private MessageQueue orderQueue;
        public CourseOrderReceiverWindow()
        {
            InitializeComponent();
            string queueName = CourseOrder.CourseOrderQueueName;
            orderQueue = new MessageQueue(queueName);
            orderQueue.Formatter = new XmlMessageFormatter(
                new Type[]
                {
                    typeof(CourseOrder),
                    typeof(Customer),
                    typeof(Course)
                }
            );
            // Запуск задачи, заполняющей ListBox заказами
            Thread t1 = new Task(PeekMessages);
            t1.Start();
        }
    }
}
```

Фрагмент кода `CourseOrderReceiver\CourseOrderReceiverWindow.xaml.cs`

Главный метод задачи — `PeekMessages()` — использует перечислитель очереди сообщений для отображения всех сообщений. Внутри цикла `while` перечислитель `messagesEnumerator` проверяет наличие нового сообщения в очереди. Если в очереди

нет сообщений, организуется ожидание в течение трех часов появления нового сообщения, и работа завершается.

Поток не может напрямую писать текст в окно списка, чтобы отображать в нем каждое сообщение из очереди, а потому должен переадресовать вызов потоку, создавшему окно списка. Поскольку элементы управления WPF привязаны к одному потоку, доступ к их методам и свойствам разрешен только потоку-создателю. Метод `Dispatcher.Invoke()` переадресует запрос потоку-создателю.

```
private void PeekMessages()
private void PeekMessages()
{
    using (MessageEnumerator messagesEnumerator =
        orderQueue.GetMessageEnumerator2())
    {
        while (messagesEnumerator.MoveNext(TimeSpan.FromHours(3)))
        {
            var labelId = new LabelIdMapping()
            {
                Id = messagesEnumerator.Current.Id,
                Label = messagesEnumerator.Current.Label
            };
            Dispatcher.Invoke(DispatcherPriority.Normal,
                new Action<LabelIdMapping>(AddListItem),
                labelId);
        }
    }
    MessageBox.Show("No orders in the last 3 hours. Exiting thread",
        // Заказы в последние 3 часа не поступали. Завершение потока.
        "Course Order Receiver", MessageBoxButton.OK,
        MessageBoxImage.Information);
}

private void AddListItem(LabelIdMapping labelIdMapping)
{
    listOrders.Items.Add(labelIdMapping);
}
```

Элемент управления `ListBox` содержит элементы класса `LabelIdMapping`. Этот класс служит для отображения меток сообщений в окне списка, оставляя скрытым идентификатор каждого сообщения. Идентификатор сообщения может быть использован для последующего чтения сообщения.

```
private class LabelIdMapping
{
    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 `Priority` должно быть установлено свойство `MessageReadPropertyFilter`.

```
private void listOrders_SelectionChanged(object sender,
    RoutedEventArgs e)
{
    LabelIdMapping labelId = listOrders.SelectedItem as LabelIdMapping;

    if (labelId == null)
        return;

    orderQueue.MessageReadPropertyFilter.Priority = true;
    Message message = orderQueue.PeekById(labelId.Id);
    CourseOrder order = message.Body as CourseOrder;

    if (order != null)
    {
        textCourse.Text = order.Course.Title;
        textCompany.Text = order.Customer.Company;
        textContact.Text = order.Customer.Contact;
        buttonProcessOrder.IsEnabled = true;
        if (message.Priority > MessagePriority.Normal)
        {
            labelPriority.Visibility = Visibility.Visible;
        }
        else
        {
            labelPriority.Visibility = Visibility.Hidden;
        }
    }
    else
    {
        MessageBox.Show("The selected item is not a course order",
            // Выбранный элемент не является заказом курса
            "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 окне списка, и сообщение удаляется из очереди с помощью метода `ReceiveById()`.

```
private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    LabelIdMapping labelId = listOrders.SelectedItem as LabelIdMapping;
    Message message = orderQueue.ReceiveById(labelId.Id);
    listOrders.Items.Remove(labelId);
    listOrders.SelectedIndex = -1;
    buttonProcessOrder.IsEnabled = false;
    textCompany.Text = string.Empty;
    textContact.Text = string.Empty;
    textCourse.Text = string.Empty;
    MessageBox.Show("Course order processed", "Course Order Receiver",
        // Заказ курса обработан
        MessageBoxButton.OK, MessageBoxImage.Information);
}
}
```

На рис. 46.13 показано работающее приложение для приема заказов, в котором отображены три заказа из очереди, из которых один в данный момент выбран.

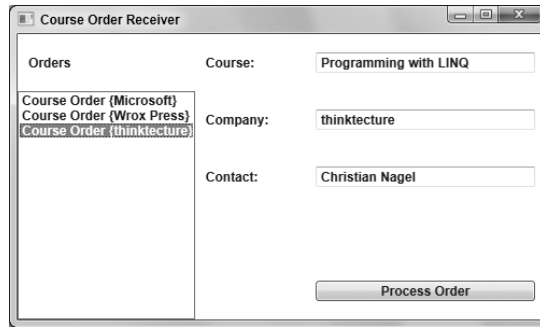


Рис. 46.13. Приложение для приема заказов

Получение результатов

В текущей версии примера приложение, отправляющее заказы, никак не может узнать, было ли принято сообщение к исполнению. Для извлечения результатов от получателя можно использовать очереди подтверждающих или ответных сообщений.

Очереди подтверждений

Через очередь подтверждений приложение-отправитель может получать информацию о состоянии сообщения. С помощью подтверждений можно определять, нужен ли ответ в ситуации, когда все прошло хорошо или что-то не так. Например, подтверждения могут отправляться о том, что сообщение достигло целевой очереди либо было прочитано, или же о том, что 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. Она работает в противоположном направлении: подтверждения получает исходный отправитель. Свойство `AcknowledgeType` устанавливается в `AcknowledgeTypes.FullReceive` для получения подтверждения о прочтении сообщения.

```
Message message = new Message(order);

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

Корреляционный идентификатор (correlation ID) служит для определения того, к какому исходному сообщению относится подтверждение. 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 применения подтверждений может применяться очередь “мертвых писем”, куда попадают сообщения, которые не появились в месте назначения. Установив свойство `UseDeadLetterQueue` класса `Message` в `true`, можно скопировать сообщение в очередь “мертвых писем”, если оно не появилось в целевой очереди до истечения заданного тайм-аута.

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

Очереди ответов

Если необходимо получить от принимающего приложения 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 назначить очередь ответов через свойство `ResponseQueue` класса `Message`. В приведенном ниже примере кода показано, каким образом получатель применяет очередь ответов для возврата ответного сообщения. В этом ответном сообщении 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)
{
    Message responseMessage = new Message("response");
    responseMessage.CorrelationId = message.Id;
    message.ResponseQueue.Send(responseMessage);
}
```

Транзакционные очереди

В случае восстанавливаемых сообщений нет никакой гарантии их доставки в том порядке, в котором 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 Queueing`.

Транзакционные очереди должны применяться, когда необходимы следующие гарантии:

- сообщения должны появляться в том порядке, в котором были отправлены;
- сообщения должны появляться по одному разу.

В случае транзакционных очередей одна транзакция не охватывает отправку и прием сообщений. Природа `Message Queueing` такова, что между отправкой и получением сообщения может пройти довольно длительное время. В отличие от этого, транзакции должны быть кратковременными. В `Message Queueing` 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 использованием транзакции.

Транзакционная очередь сообщений создается передачей `true` во втором параметре методу `MessageQueue.Create()`.

Если в очередь необходимо записывать сразу несколько сообщений в пределах 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 возможности 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. На рис. 46.14 показана упрощенная архитектура. Для отправки сообщения в очередь клиентское приложение вызывает метод прокси-объекта WCF. Сообщение создается посредством прокси. Разработчику клиента не зачем 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. Слушатель канала MSMQ на стороне службы читает сообщения из очереди, преобразует их в вызовы методов и обращается к соответствующим методам службы.

Давайте преобразуем приложение заказа курсов для использования Message Queuing с точки зрения WCF. В этом решении модифицируются три созданных ранее проекта и добавляется одна сборка, включающая контракт службы WCF.

- Библиотека компонентов (CourseOrder) включает сущностные классы для сообщений, передаваемых по сети. Эти сущностные классы модифицируются для соответствия контракту данных и сериализации с помощью WCF.
- Добавлена новая библиотека (CourseOrderService), которая определяет контракт, предоставляемый службой.

- WPF-приложение отправителя (CourseOrderSender) модифицируется, чтобы вместо отправки сообщений вызывались методы прокси WCF.
- WPF-приложение получателя (CourseOrderReceiver) модифицируется для использования службы WCF, реализующей контракт.

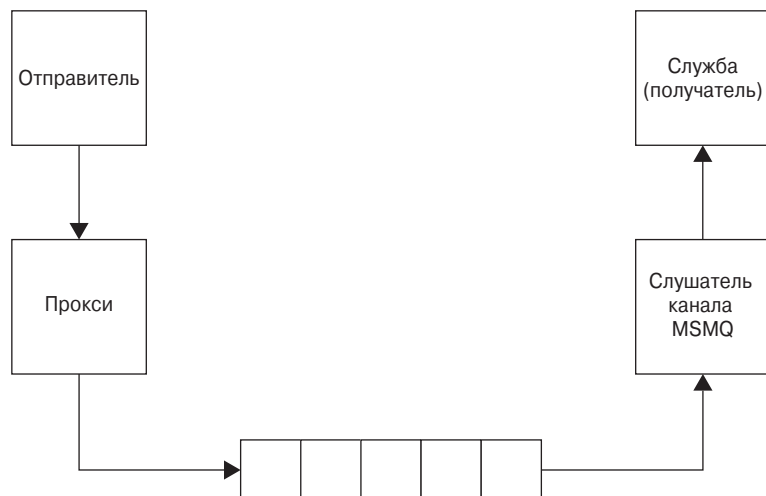


Рис. 46.14. Архитектура Message Queuing с WCF

Классы сущностей с контрактом данных

В библиотеке CourseOrder классы Course, Customer и CourseOrder модифицируются для 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.

```

using System.Runtime.Serialization;
namespace Wrox.ProCSharp.Messaging
{
    [DataContract]
    public class Course
    {
        [DataMember]
        public string Title { get; set; }
    }
}
  
```

Фрагмент кода CourseOrder\Course.cs

Класс Customer также требует атрибутов контракта данных:


```

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

Фрагмент кода CourseOrder\Course.cs

В классе `CourseOrder` не только добавляются атрибуты контракта данных, но и переопределяется метод `ToString()`, чтобы предоставить строковое представление этих объектов по умолчанию:

```

 [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);
    }
}


```

Фрагмент кода `CourseOrder\CourseOrder.cs`

Контракт службы WCF

Для предоставления службы через контракт службы WCF добавим библиотеку служб 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 должен ожидать немедленного результата. В атрибуте `[OperationContract]` устанавливается свойство `IsOneWay`. Вызывающий эту операцию код не ожидает ответа от службы.

```

 using System.ServiceModel;
namespace Wrox.ProCSharp.Messaging
{
    [ServiceContract]
    [DeliveryRequirements(
        QueuedDeliveryRequirements=QueuedDeliveryRequirementsMode.Required)]
    public interface ICourseOrderService
    {
        [OperationContract(IsOneWay = true)]
        void AddCourseOrder(CourseOrder courseOrder);
    }
}

```

Фрагмент кода `CourseOrderServiceContract\ICourseOrderService.cs`



Для получения ответа на стороне клиента можно использовать очереди подтверждений и ответов.


WCF-приложение получателя сообщений

WPF-приложение `CourseOrderReceiver` теперь модифицировано 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, свойство `UseSynchronizationContext` устанавливается с атрибутом `[ServiceBehavior]`. Это средство исполняющей системы WCF для передачи вызова метода в поток, определенный контекстом синхронизации приложения WPF.

```

 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)
        {
            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; }
    }
}

```

Фрагмент кода `CourseOrderReceiver\CourseOrderService.cs`



О контексте синхронизации более подробно рассказывалось в главе 20.

В конструкторе класса `CourseReceiverWindow` создается и открывается для запуска слушателя экземпляр объекта `ServiceHost`. Привязка слушателя 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.



Лямбда-выражения подробно рассматривались в главе 8.

Здесь используется класс коллекции `ObservableCollection<T>` из пространства имен `System.Collections.ObjectModel`. В нем реализован интерфейс `INotifyCollectionChanged` и потому элементы управления WPF, привязанные к коллекции, информированы о динамических изменениях в списке.

```

 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();
            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);
            }
            this.DataContext = courseOrders;
        }
    }
}
```

Фрагмент кода *CourseOrderReceiver\CourseOrderReceiverWindow.xaml.cs*

Элементы WPF в коде XAML теперь используют привязку данных. Элемент `ListBox` привязан к контексту данных, а простые элементы управления — к свойствам текущего элемента в контексте данных:

```
 <ListBox Grid.Row="1" x:Name="listOrders" ItemsSource="{Binding}"
    IsSynchronizedWithCurrentItem="true" />
<!-- ... -->
<TextBox x:Name="textCourse" Grid.Row="0" Grid.Column="1"
    Text="{Binding Path=Course.Title}" />
<TextBox x:Name="textCompany" Grid.Row="1" Grid.Column="1"
    Text="{Binding Path=Customer.Company}" />
<TextBox x:Name="textContact" Grid.Row="2" Grid.Column="1"
    Text="{Binding Path=Customer.Contact}" />
```

Фрагмент кода *CourseOrderReceiver\CourseOrderReceiverWindow.xaml*

В конфигурационном файле приложения определена привязка `netMsmqBinding`. Для надежного обмена сообщениями требуются транзакционные очереди. Чтобы получать и отправлять сообщения в нетранзакционные очереди, свойство `exactlyOnce` должно быть установлено в `false`.



Привязка `netMsmqBinding` будет использоваться в приложениях отправителя и получателя, если они оба являются приложениями WCF. Если одно из них использует API-интерфейс `System.Messaging` для отправки и получения сообщений или же является старым приложением COM, можно применять привязку `msmqIntegrationBinding`.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <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:

```

❶ private void buttonProcessOrder_Click(object sender, RoutedEventArgs e)
{
    CourseOrder courseOrder = listOrders.SelectedItem as CourseOrder;
    courseOrders.Remove(courseOrder);
    listOrders.SelectedIndex = -1;
    buttonProcessOrder.IsEnabled = false;
    MessageBox.Show("Course order processed", "Course Order Receiver",
        // Заказ купца обработан
        MessageBoxButton.OK, MessageBoxImage.Information);
}

```

Фрагмент кода `CourseOrderReceiver\CourseOrderReceiverWindow.xaml.cs`

WCF-приложение отправителя сообщений

Приложение-отправитель также модифицировано для использования класса прокси WCF.

Для контракта службы включена ссылка на сборку `CourseOrderServiceContract`, а сборка `System.ServiceModel` необходима для применения классов WCF.

В обработчике событий Click элемента управления `buttonSubmit` класс `ChannelFactory` возвращает прокси. Прокси посылает сообщение в очередь, вызывая метод `AddCourseOrder()`.

```

❶ private void buttonSubmit_Click(object sender, RoutedEventArgs e)
{
    try
    {
        var order = new CourseOrder
        {
            Course = new Course()
            {
                Title = comboCourses.Text
            },
            Customer = new Customer()
            {
                Company = textCompany.Text,
                Contact = textContact.Text
            }
        };
    }
}

```

```

var factory = new ChannelFactory<ICourseOrderService>("queueEndpoint");
ICourseOrderService proxy = factory.CreateChannel();
proxy.AddCourseOrder(order);
factory.Close();
MessageBox.Show("Course order submitted", "Course Order",
    // Заказ на курс отправлен
    MessageBoxButtons.OK, MessageBoxImage.Information);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message, "Course Order Error",
        // Возникла ошибка с заказом
        MessageBoxButtons.OK, MessageBoxImage.Error);
}
}

```

Фрагмент кода *CourseOrderSender\CourseOrderWindow.xaml.cs*

В конфигурационном файле приложения определена клиентская часть соединения WCF. В ней снова используется `netMsmqBinding`.



```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <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" />
    </client>
  </system.serviceModel>
</configuration>

```

Фрагмент кода *CourseOrderSender\app.config*

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 с WCF.

Однако для создания очередей сообщений и удаления сообщений все-таки нужен класс `MessageQueue`. WCF в данном случае — просто абстракция для процесса отправки и получения сообщений.



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 Message Queuing

Очереди сообщений создаются с помощью 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 компонента позволит 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.

Установщики более подробно рассматриваются в главе 17.

Резюме

В этой главе было показано, как применять технологию Message Queuing. Это очень важная технология, которая позволяет налаживать не только асинхронные, но и разбеденные 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 по 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 позволяет применять предлагаемые в WCF концепции для отправки сообщений за счет вызова методов прокси-класса и получения сообщений за счет реализации службы.

Следующая глава посвящена службам каталогов.