Глава 3

Системы обмена сообщениями

Введение

В главе 2 были рассмотрены различные стили интеграции приложений, включая *обмен сообщениями* (*Messaging*, с. 87). Обмен сообщениями позволяет наладить асинхронное взаимодействие между слабо связанными приложениями. Передачу данных между интегрированными приложениями обеспечивает система обмена сообщениями.

Основные концепции обмена сообщениями

Как и большинство технологий, *обмен сообщениями* (*Messaging*, с. 87) характеризуется несколькими базовыми концепциями.

- **Каналы.** Приложения, объединенные с помощью технологии обмена сообщениями, передают данные по *каналам сообщений* (*Message Channel*, с. 93). Изначально система обмена сообщениями не содержит каналов; они создаются по мере определения способов взаимодействия приложений.
- **Сообщения.** *Сообщение* (*Message*, с. 98) — это наименьшая единица данных, которая может быть передана по каналу сообщений. Следовательно, для передачи данных отправитель должен разбить их на пакеты, которые затем будут упакованы в сообщения и помещены в канал. Подобным образом получатель извлекает сообщения из канала и выделяет из них полезные данные. Система обмена сообщениями гарантирует доставку сообщений путем их повторной отправки.
- **Каналы и фильтры.** В самом простом случае система обмена сообщениями доставляет сообщение непосредственно с компьютера отправителя на компьютер получателя. Однако во многих ситуациях необходима промежуточная обработка отправленного сообщения (например, преобразование формата или проверка соответствия некоторым правилам) прежде, чем оно будет принято получателем. Архитектура *каналов и фильтров* (*Pipes and Filters*, с. 102) описывает процесс организации многоэтапной обработки сообщения с использованием каналов.
- **Маршрутизация.** Для того чтобы достичь конечной точки назначения, сообщению может потребоваться пройти через несколько различных каналов. Иногда сложность маршрута сообщения приводит к тому, что отправитель не может опреде-

лить канал, в который необходимо поместить сообщение для его доставки получателю. В этом случае отправитель передает сообщение *маршрутизатору сообщений* (*Message Router*, с. 109) — программному компоненту, играющему роль фильтра в рамках архитектуры *каналы и фильтры*. Маршрутизатор определяет канал, в который необходимо поместить сообщение для его доставки конечному получателю или следующему маршрутизатору.

- **Преобразование.** Зачастую отправитель и получатель используют различные форматы для представления одних и тех же данных. Для преобразования формата отправителя в формат получателя сообщение должно пройти через промежуточный фильтр, получивший название *транслятор сообщений* (*Message Translator*, с. 115).
- **Конечные точки.** Большинство приложений не обладает встроенными средствами интеграции. Для подключения таких приложений к системе обмена сообщениями необходим промежуточный код, учитывающий особенности приложения и системы обмена сообщениями. Этот код является совокупностью *конечных точек сообщения* (*Message Endpoint*, с. 124), позволяющих приложению отправлять и принимать сообщения.

Об организации книги

Шаблоны, представленные в этой главе, позволяют получить базовое представление об интеграции приложений с помощью *обмена сообщениями* (*Messaging*, с. 87). Более подробно каждый из представленных здесь корневых шаблонов рассматривается в одной из следующих глав книги (рис. 3.1).

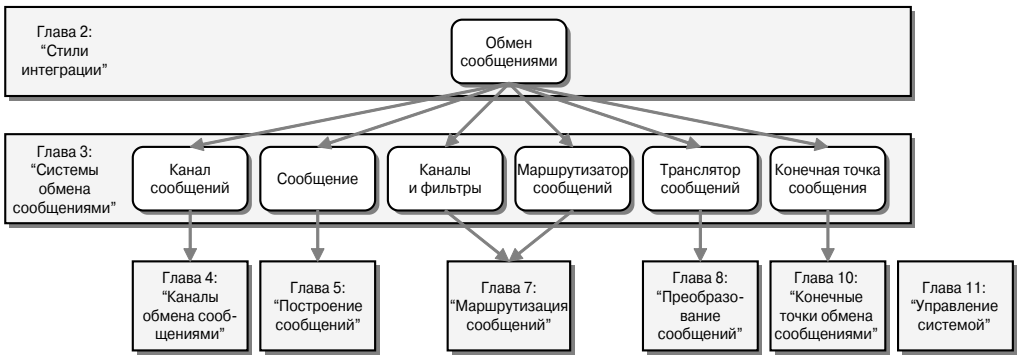


Рис. 3.1. Корневые шаблоны проектирования

Канал сообщений (Message Channel)

Компании необходимо наладить взаимодействие между двумя отдельными приложениями с помощью *обмена сообщениями* (*Messaging*, с. 87).

Как наладить взаимодействие между двумя приложениями с использованием технологии обмена сообщениями?

Наличие системы обмена сообщениями не является достаточным условием для взаимодействия подключенных к ней приложений. Другими словами, отправитель не может использовать систему обмена сообщениями для передачи информации получателю. Если бы это было так, система обмена сообщениями должна была бы обладать некими "магическими" свойствами (рис. 3.2).

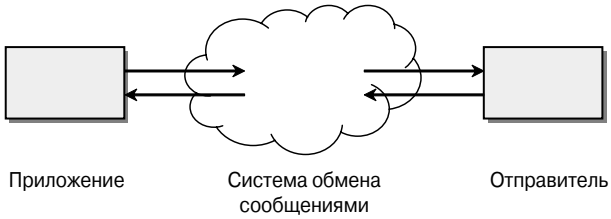


Рис. 3.2. "Магическая" система обмена сообщениями

Детерминированный обмен данными между приложениями обеспечивает набор соединений.

Соедините приложения с помощью *канала сообщений* (*Message Channel*), доставляющего помещенную в него информацию от отправителя к получателю.

Приложение, которому необходимо отправить информацию, помещает ее не просто в систему обмена сообщениями, а в конкретный *канал сообщений*. Подобным образом приложение, которому необходимо получить информацию, обращается не просто к системе обмена сообщениями, а к конкретному *каналу сообщений*.

Система обмена сообщениями подразумевает наличие отдельных *каналов сообщений* для каждого типа передаваемой информации. Таким образом, отправитель помещает информацию не в произвольный *канал сообщений*, а в *канал сообщений*, предназначенный для передачи данных конкретного типа. Аналогично получатель извлекает информацию из *канала сообщений*, предназначенного для передачи именно этого типа информации.

В действительности каналы представляют собой логические адреса в системе обмена сообщениями. Реализация каналов сообщений зависит от конкретной системы обмена сообщениями. К примеру, все *конечные точки сообщения* (*Message Endpoint*, с. 124) могут быть соединены друг с другом или подключены к центральному коммутатору, а несколько логических каналов сообщений могут быть представлены одним физическим каналом.

Как бы там ни было, логические каналы скрывают детали конкретной реализации от приложений.

Система обмена сообщениями не содержит заранее сконфигурированных каналов сообщений. Требуемые каналы сообщений определяются на этапе проектирования интеграционного решения и реализуются администратором системы обмена сообщениями. Несмотря на то что некоторые системы обмена сообщениями поддерживают создание новых каналов сообщений после начала эксплуатации интеграционного решения, это не позволяет распространить информацию о новых каналах между всеми приложениями. Следовательно, количество и назначение каналов сообщений необходимо определить до этапа развертывания интеграционного решения. (Об исключениях из этого правила рассказывается во введении к главе 4.)

О словаре обмена сообщениями

Существует несколько различных терминов, применяющихся для обозначения приложений, взаимодействующих посредством *канала сообщений*. Наиболее известными из них являются термины *отправитель* и *получатель* — приложение отправляет информацию с помощью *канала сообщений* для ее получения другим приложением. Также популярны термины *поставщик* и *потребитель*. Приложения, взаимодействующие посредством *канала "публикация–подписка"* (*Publish-Subscribe Channel*, с. 134), известны как *публикатор* и *подписчик* (следует отметить, что эти термины часто употребляются и в более широком смысле). Иногда говорят, что приложение *слушает* канал, по которому *вещает* другое приложение. Наконец, существуют такие термины, как *клиент* и *сервер*, однако они настолько устарели, что их употребление в данном контексте может считаться дурным тоном.

Иногда употребление того или иного термина может показаться некорректным. Так, в рамках архитектуры Web-служб приложение, отправляющее сообщение с запросом поставщику службы, называется *потребителем*. К счастью, применение термина *потребитель* в данном контексте ограничивается сценариями *удаленного вызова процедуры* (*Remote Procedure Invocation*, с. 85). В целом же приложение, которое отправляет или принимает сообщения, можно назвать *клиентом* системы обмена сообщениями или *конечной точкой сообщения*.

Зачастую разработчики интеграционного решения неверно интерпретируют задачу создания канала сообщений. Как правило, разработчик создает код Java, вызывающий метод JMS API `createQueue`, или код .NET, включающий в себя выражение `new MessageQueue`, однако ни тот, ни другой не выделяет ресурсы для новой очереди в системе обмена сообщениями. В действительности подобный код создает экземпляр объекта, предоставляющего доступ к ресурсу, который был заранее предусмотрен администратором системы обмена сообщениями.

Планируя каналы сообщений интеграционного решения, следует помнить, что каждый канал отнимает определенный объем оперативной или дисковой памяти. Любая реализация системы обмена сообщениями предполагает наличие жесткого лимита поддерживаемых каналов сообщений. Если разрабатываемое вами интеграционное решение требует тысяч каналов сообщений, обратите внимание на систему обмена сообщениями, обладающую незаурядной способностью к масштабированию.

Имена каналов сообщений

Поскольку каналы являются логическими адресами, они должны иметь некоторую форму записи. В большинстве случаев для обращения к каналу сообщений используется буквенно-цифровое имя, такое как `MyChannel`. Некоторые системы обмена сообщениями поддерживают иерархические имена каналов, например `MyCorp/Prod/OrderProcessing/NewOrders`.

Все каналы сообщений делятся на два типа: *каналы "точка-точка"* (*Point-to-Point Channel*, с. 131) и *каналы "публикация-подписка"* (*Publish-Subscribe Channel*, с. 134). Поскольку передача разнородной информации по одному и тому же каналу может привести к возникновению непредвиденных ситуаций, рекомендуется использовать несколько *каналов типа данных* (*Datatype Channel*, с. 139) или реализовать функциональность *избирательного потребителя* (*Selective Consumer*, с. 528). Кроме того, для каждого приложения рекомендуется предусмотреть *канал недопустимых сообщений* (*Invalid Message Channel*, с. 143), в который будут помещаться сообщения, не соответствующие определенным правилам. Приложение, не имеющее доступа к клиенту *обмена сообщениями* (*Messaging*, с. 87), можно подключить к системе обмена сообщениями с помощью *адаптеров канала* (*Channel Adapter*, с. 154). Набор каналов сообщений формирует *шину сообщений* (*Message Bus*, с. 162) для интегрируемых приложений.

Пример: *торговля на бирже*

Инициатор сделки помещает сообщение с запросом в соответствующий *канал сообщений*. Приложение, обрабатывающее запросы о сделках, считывает это сообщение из того же канала. Если запрашиваемому приложению нужно узнать текущие котировки, оно помещает сообщение с запросом в другой *канал сообщений*, специально предназначенный для получения информации данного типа.

Пример: *эталонная реализация службы J2EE JMS*

Набор инструментальных средств разработки (SDK) J2EE поставляется с эталонными реализациями служб J2EE, включая службу JMS. Для запуска эталонного сервера предназначена команда `j2ee`, а для настройки каналов сообщений — средство `j2eeadmin`:

```
j2eeadmin -addJmsDestination jms/mytopic topic
j2eeadmin -addJmsDestination jms/myqueue queue
```

Как только администратор создаст каналы сообщений, к ним можно будет осуществлять доступ из клиентского кода JMS:

```
Context jndiContext = new InitialContext();
Queue myQueue = (Queue) jndiContext.lookup("jms/myqueue");
Topic myTopic = (Topic) jndiContext.lookup("jms/mytopic");
```

Запрос к службе JNDI не создает очередь или тему, поскольку они были созданы ранее с помощью команды `j2eeadmin`. Вместо этого JNDI-запрос создает экземпляр объекта `Queue`, который предоставляет доступ к очереди в системе обмена сообщениями.

***Пример:** IBM WebSphere MQ*

Следующее выражение создает точку назначения (очередь `myQueue`) в системе обмена сообщениями, основанной на использовании связующего ПО IBM WebSphere MQ:

```
DEFINE Q(myQueue)
```

Поскольку WebSphere MQ (без полнофункционального сервера WebSphere Application Server) не содержит реализации службы JNDI, мы не можем использовать ее для доступа к очереди сообщений. В этой ситуации доступ к очереди сообщений реализуется через JMS-сеанс, как показано ниже:

```
Session session = // Создание сеанса.
Queue queue = session.createQueue("myQueue");
```

***Пример:** Microsoft MSMQ*

Связующее ПО Microsoft MSMQ предоставляет несколько различных способов создания канала сообщений (очереди). В частности, для создания очереди можно воспользоваться средством Microsoft Message Queue Explorer или консолью Computer Management, как показано на рис. 3.3.

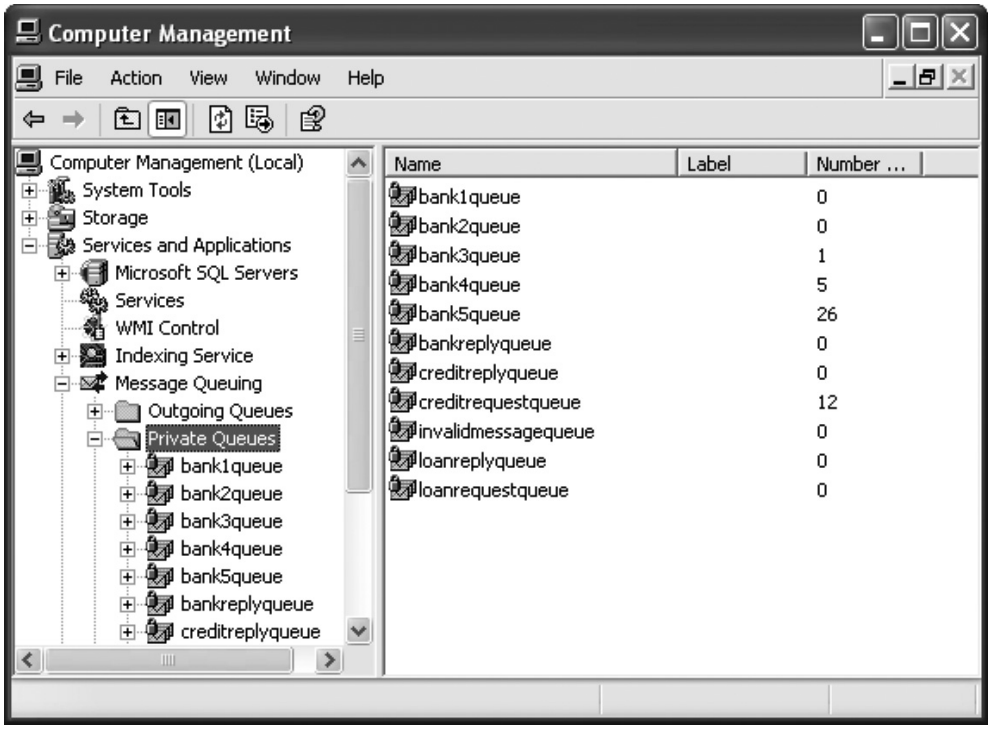


Рис. 3.3. Консоль Computer Management позволяет создавать, настраивать и удалять очереди сообщений

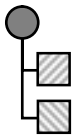
Ниже приведен пример создания очереди сообщений с помощью программного кода:

```
using System.Messaging;
...
MessageQueue.Create("MyQueue");
```

Для доступа к очереди сообщений приложение должно создать экземпляр объекта `MessageQueue`:

```
MessageQueue mq = new MessageQueue("MyQueue");
```

Сообщение (Message)



Приложения, соединенные посредством *канала сообщений* (*Message Channel*, с. 93), взаимодействуют с помощью технологии *обмена сообщениями* (*Messaging*, с. 87).

Как наладить обмен данными между двумя приложениями, соединенными с помощью *канала сообщений* (*Message Channel*)?

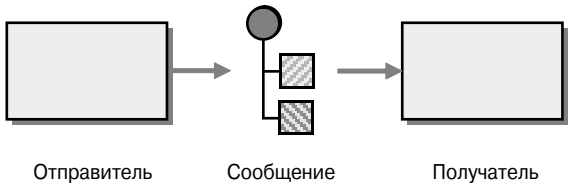
Зачастую наличие *канала сообщений* воспринимается как достаточное условие для обмена данными между двумя приложениями. Однако, в отличие от непрерывного потока информации, данные большинства приложений состоят из различных элементов, таких как записи, объекты, строки таблицы базы данных и т.п. Следовательно, канал сообщений должен обладать возможностью передачи разнородных элементов данных.

Что же представляет собой процесс "передачи" информации? К примеру, передача параметра функции осуществляется посредством передачи указателя на адрес этого параметра в памяти. Подобным образом два потока, разделяющих одну область памяти, могут обмениваться указателями на записи или объекты.

Поскольку двум различным процессам соответствуют отдельные области памяти, обмен информацией между процессами возможен только за счет копирования данных из одной области памяти в другую. Обычно для передачи данных используется поток байтов. Это означает, что первый процесс должен *упорядочить* данные в поток байтов и скопировать его во второй процесс. В свою очередь, второй процесс *разупорядочить* данные в их исходное состояние. Механизм *упорядочивания* используется технологией удаленного вызова процедуры (Remote Procedure Call — RPC) для передачи параметров удаленному процессу и возврата результата.

Таким образом, технология обмена сообщениями подразумевает передачу дискретных элементов данных путем их *упорядочения* отправителем и *разупорядочения* получателем. Чтобы обеспечить возможность передачи разнородной информации, необходимо предусмотреть общий формат данных, передаваемых по каналу сообщений.

Для передачи информации между двумя приложениями используйте формат *сообщения* (*Message*) — наименьшей единицы данных, которая может быть передана системой обмена сообщениями по *каналу сообщений* (*Message Channel,* с. 93).



Следовательно, для передачи по каналам системы обмена сообщениями данные должны быть преобразованы в одно или несколько сообщений.

Сообщение состоит из двух основных частей.

1. **Заголовок сообщения.** Информация, которая используется системой обмена сообщениями для описания передаваемых данных, включая адрес отправителя, адрес получателя и т.п.
2. **Тело сообщения.** Данные, передаваемые внутри сообщения и, как правило, игнорируемые системой обмена сообщениями.

Представленная выше концепция отнюдь не нова. Передача информации в виде дискретных сообщений применяется как традиционной почтовой службой, так и системой электронной почты. В сетях Ethernet данные передаются в виде кадров, а в Интернете — в виде пакетов TCP/IP.

С точки зрения системы обмена сообщениями все сообщения одинаковы: тело каждого из них содержит данные, которые нужно передать в соответствии с информацией, размещающейся в заголовке. Однако, с точки зрения разработчика интеграционного решения, существуют разные типы сообщений, которые отличаются способом их использования. Так, для удаленного вызова процедуры применяется *сообщение с командой* (*Command Message*, с. 169), для передачи информации — *сообщение с данными документа* (*Document Message*, с. 171), а для уведомления получателя сообщения об изменении состояния отправителя — *сообщение о событии* (*Event Message*, с. 174). Если удаленное приложение должно ответить на полученное сообщение, используйте модель *запрос-ответ* (*Request-Reply*, с. 177).

Если приложению необходимо передать информацию, размер которой превышает максимально допустимый размер сообщения, разбейте ее на меньшие части и отправьте в виде *цепочки сообщений* (*Message Sequence*, с. 192). Если передаваемая информация актуальна только в течение ограниченного промежутка времени, отметьте эту особенность, указав *срок действия сообщения* (*Message Expiration*, с. 198). Поскольку все отправители и получатели сообщений должны использовать один и тот же формат данных, задайте его с помощью *канонической модели данных* (*Canonical Data Model*, с. 367).

***Пример:** сообщение JMS*

Сообщение JMS представлено типом `Message`, имеющим несколько различных подтипов.

1. `TextMessage`. Наиболее распространенный тип сообщения. Тело сообщения представлено строкой (тип `String`), такой как простой текст или XML-документ. Для работы с телом сообщения используйте метод `textMessage.getText()`.
2. `BytesMessage`. Универсальный тип сообщения. Тело сообщения представлено байтовым массивом. Для копирования тела сообщения в указанный массив используйте метод `bytesMessage.readBytes(byteArray)`.
3. `ObjectMessage`. Тело сообщения представлено объектом Java, реализующим интерфейс `java.io.Serializable`. Для работы с телом сообщения (тип `Serializable`) используйте метод `objectMessage.getObject()`.

4. `StreamMessage`. Тело сообщения представлено потоком данных одного из примитивных типов Java. Для работы с телом сообщения используйте методы `readBoolean()`, `readChar()`, `readDouble()` и т.п.
5. `MapMessage`. Тело сообщения представлено картой типа `java.util.Map` со строковыми (`String`) ключами. Для работы с телом сообщения используйте методы `getBoolean("isEnabled")`, `getInt("numberOfItems")` и т.п.

***Пример:** сообщение .NET*

Сообщение .NET представлено типом `Message`. Доступ к телу сообщения осуществляется посредством свойства `Body` (тип `Object`), а также свойства `ByteStream` (тип `Stream`). Тип данных, хранящихся в теле сообщения (строка, дата, валюта, число и др.), определяется свойством `BodyType`.

***Пример:** сообщение SOAP*

Сообщение протокола SOAP версии 1.1 представлено XML-конвертом (корневой элемент `SOAP-ENV:Envelope`), содержащим необязательный заголовок (элемент `SOAP-ENV:Header`) и обязательное тело (элемент `SOAP-ENV:Body`). Поскольку этот XML-документ является наименьшей единицей передачи данных (как правило, для передачи SOAP-сообщений используется протокол HTTP), он является ни чем иным, как *сообщением* (*Message*).

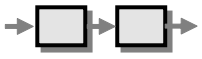
Ниже приведен типичный пример SOAP-сообщения, содержащего заголовок и тело.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
      <t:Transaction
          xmlns:t="some-URI"
          SOAP-ENV:mustUnderstand="1">
              5
      </t:Transaction>
   </SOAP-ENV:Header>
  <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
          <symbol>DEF</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP позволяет продемонстрировать рекурсивную природу сообщений. Действительно, поскольку SOAP-сообщение может быть передано с помощью системы обмена сообщениями, это означает, что сообщение системы обмена сообщениями (например, объект JMS типа `javax.jms.Message` или объект .NET `System.Messaging.Message`) будет содержать в себе SOAP-сообщение (данные XML-документа `SOAP-ENV:Envelope`). В этом случае транспортным протоколом SOAP-сообщения будет являться не HTTP, а внутренний

протокол передачи данных системы обмена сообщениями (в свою очередь, внутренний протокол системы обмена сообщениями может быть основан на использовании протокола HTTP с дополнительными средствами обеспечения надежной доставки данных). Передаче сообщения между различными системами обмена сообщениями посвящен шаблон проектирования *оболочка конверта* (*Envelope Wrapper*, с. 342).

Каналы и фильтры (Pipes and Filters)



Во многих интеграционных сценариях единственное событие может инициировать целую серию этапов обработки. Предположим, что сообщение о размещении нового заказа должно передаваться в зашифрованном виде и нести в себе аутентификационную информацию в форме цифрового сертификата. Кроме того, необходимо предусмотреть защиту от дублирования сообщений. Учитывая вышеперечисленные требования, задача первичной обработки сообщений о размещении нового заказа сводится к преобразованию последовательности зашифрованных сообщений, включающих аутентификационную информацию, с повторами в последовательность простых текстовых сообщений, не содержащих избыточных полей данных, без повторов.

Как организовать сложную обработку сообщения, руководствуясь принципами независимости и гибкости конечного решения?

Одно из возможных решений заключается в создании модуля обработки входящих сообщений, выполняющего все необходимые преобразования. Однако подобный подход не удовлетворяет требованию гибкости и усложняет тестирование. К примеру, нам может понадобиться добавить или удалить определенный этап обработки для сообщений о заказах, размещенных крупными клиентами.

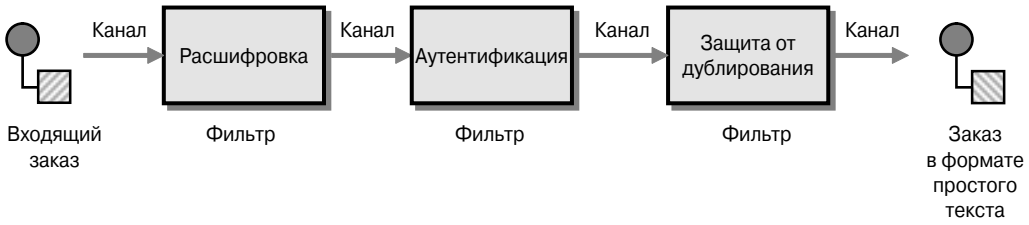
Реализация всех функций в одном компоненте снижает возможность его повторного использования. Напротив, создание небольших, узкоспециализированных компонентов существенно повышает гибкость решения. Например, обработка сообщений о проверке состояния заказа требует их расшифровки, однако не требует реализации защиты от дублирования. Следовательно, вынесение функции расшифровки в отдельный модуль позволит применять его для обработки сообщений различных типов.

Как правило, интеграционные решения используются для объединения нескольких гетерогенных систем. Это может привести к тому, что различные этапы обработки сообщения будут выполняться на разных физических устройствах. К примеру, закрытый ключ, который необходим для расшифровки входящих сообщений, по соображениям безопасности может быть доступен только для определенного компьютера. Подобным образом различные этапы обработки сообщения могут быть реализованы с помощью разных языков программирования или технологий, что будет являться препятствием для их совместного выполнения в рамках одного процесса.

Реализация каждой функции в виде отдельного компонента еще не гарантирует их независимость (к примеру, компонент расшифровки может использовать функциональность компонента аутентификации). Чтобы устранить связи между компонентами, этапы сложной обработки сообщения следует расположить в такой последовательности, которая бы исключала зависимость одного компонента от другого. Кроме того, все компоненты должны иметь универсальный внешний интерфейс, допускающий их взаимозаменяемость.

Взаимодействие между компонентами рекомендуется реализовать с помощью асинхронного обмена сообщениями. В этом случае компонент сможет отправить сообщение другому компоненту, не дожидаясь от него ответа. Использование асинхронного обмена сообщениями позволит организовать одновременную обработку нескольких сообщений, по одному на каждый компонент.

Используйте архитектуру *каналов и фильтров* (*Pipes and Filters*) для разделения сложной вычислительной задачи на последовательность простых, независимых этапов (фильтров), объединенных с помощью каналов.



Каждый фильтр имеет очень простой интерфейс: он получает сообщение по входящему каналу, обрабатывает его и публикует полученный результат в исходящем канале. Канал соединяет два фильтра и используется для передачи сообщений. Поскольку все компоненты обладают одинаковыми внешними интерфейсами, их можно комбинировать путем подключения к различным каналам. Можно добавлять новые фильтры, удалять или переупорядочивать существующие — и все это без необходимости внесения изменений в сами фильтры. Соединение между фильтром и каналом часто называют *портом*. Простой фильтр имеет один входящий и один исходящий порт.

Решение рассматриваемой нами задачи первичной обработки сообщений о размещении нового заказа с помощью архитектуры *каналов и фильтров* предполагает применение трех фильтров и двух соединяющих эти фильтры каналов. Кроме того, нам понадобится один дополнительный канал для передачи исходного сообщения компоненту расшифровки и один канал для передачи обработанного (текстового) сообщения системе управления заказами.

Архитектура *каналов и фильтров* является фундаментальной для систем обмена сообщениями: отдельные этапы обработки (фильтры) соединяются посредством каналов сообщений. На этой архитектуре основано множество шаблонов проектирования, рассматривающихся в книге (например, шаблоны маршрутизации и преобразования). К тому же архитектура *каналов и фильтров* описывает универсальный способ объединения отдельных шаблонов проектирования в более сложные решения.

Канал в архитектуре *каналы и фильтры* — это абстрактная сущность, соединяющая компоненты и одновременно отделяющая их друг от друга. Канал позволяет компоненту отправить сообщение для его последующего получения другим (неизвестным для отправителя) компонентом. Очевидно, что канал в архитектуре *каналы и фильтры* может быть представлен *каналом сообщений* (*Message Channel*, с. 93). Благодаря независимости *канала сообщений* от языка программирования, а также аппаратной или программной платформы этапы обработки (фильтры) могут быть распределены между различными компьютерами по соображениям безопасности, производительности, поддержки и др. Если же все компоненты удается разместить на одном компьютере, следует подумать о более эффективной реализации каналов, такой как очередь в памяти. Таким образом, при проектировании компонентов рекомендуется воздерживаться от конкретной реализации интерфейса канала. Более подробно этот подход описывается шаблоном проектирования *шлюз обмена сообщениями* (*Messaging Gateway*, с. 482).

Один из недостатков архитектуры *каналов и фильтров* заключается в использовании большого числа каналов. Следует помнить, что реализация канала предполагает расходование ресурсов памяти и процессора. К тому же публикация сообщения в канале требует преобразования данных из внутреннего формата отправителя в формат инфраструктуры обмена сообщениями (при получении сообщения выполняется обратная процедура). Таким образом, наличие длинной цепочки фильтров может привести к существенному снижению производительности вследствие многократного преобразования формата данных сообщения.

В чистом виде архитектура *каналов и фильтров* предполагает наличие у каждого фильтра одного входящего и одного исходящего порта. Специфика *обмена сообщениями* (*Messaging*, с. 87) требует пересмотра этого условия. В частности, компонент может извлекать сообщения из более чем одного канала и помещать сообщения в более чем один канал (примером подобного компонента является *маршрутизатор сообщений* (*Message Router*, с. 109)). Кроме того, несколько фильтров могут считывать сообщения из одного и того же *канала сообщений*. Если же сообщение предназначено единственному получателю, для его передачи применяется *канал "точка–точка"* (*Point-to-Point Channel*, с. 131).

Применение архитектуры *каналов и фильтров* позволяет тестировать отдельные компоненты с помощью *тестового сообщения* (*Test Message*, с. 577). Тестирование и отладка базовых функций решения гораздо эффективнее, нежели тестирование всего решения целиком. К примеру, для тестирования функций шифрования и расшифровки данных следует сгенерировать большое количество сообщений с произвольным содержимым, передать их для последовательной обработки компонентам шифрования и расшифровки, а затем сравнить полученный результат с исходными сообщениями. Подобным образом для тестирования функции аутентификации следует сгенерировать сообщения с заранее известными кодами аутентификации и передать их на обработку соответствующему компоненту.

Конвейерная обработка

Объединение компонентов с помощью асинхронных *каналов сообщений* (*Message Channel*, с. 93) позволяет каждому из них выполняться в собственном потоке или процессе. Это означает, что, завершив обработку сообщения и поместив его в исходящий канал, компонент может приступить к обработке нового сообщения, не дожидаясь подтверждения о получении сообщения следующим компонентом в цепочке. Подобный подход позволяет обрабатывать несколько сообщений одновременно, как показано на рис. 3.4. По сравнению с последовательной обработкой сообщений *конвейерная обработка* существенно повышает пропускную способность системы.

Параллельная обработка

Пропускная способность системы ограничена пропускной способностью ее самого медленного процесса. Увеличение пропускной способности процесса возможно за счет создания его нескольких параллельно выполняющихся экземпляров. Гарантия получения сообщения ровно одним из N доступных обработчиков обеспечивается сочетанием *канала "точка–точка"* (*Point-to-Point Channel*, с. 131) и *конкурирующих потребителей* (*Competing Consumers*, с. 515). Следует отметить, что подобный способ увеличения производительности системы может привести к нарушению порядка обработки сообщений. Если порядок сообщений имеет значение, нужно ограничиться только одним экземпляром каждого компонента или воспользоваться *преобразователем порядка* (*Resequencer*, с. 297).

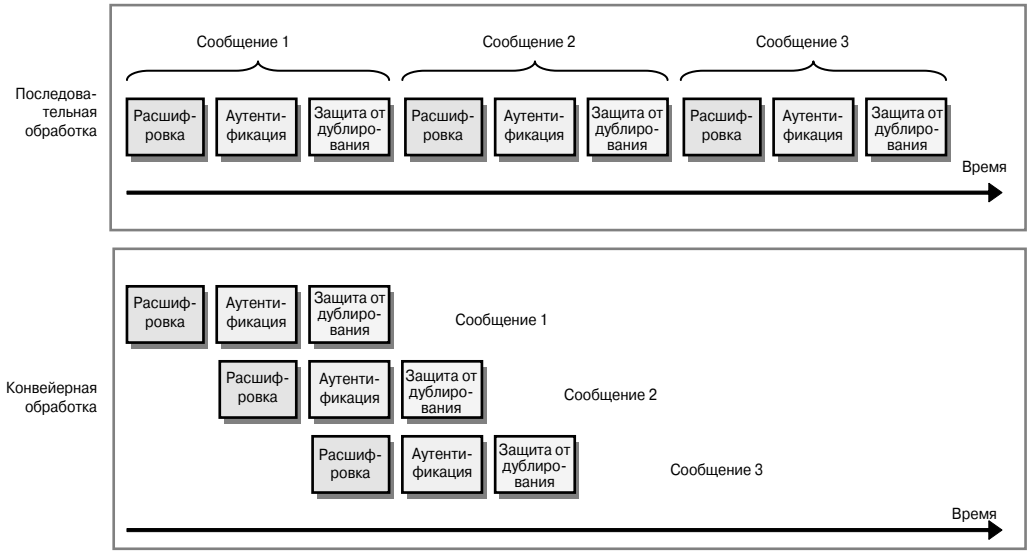


Рис. 3.4. Конвейерная обработка сообщений с помощью архитектуры каналов и фильтров (Pipes and Filters)

Предположим, что на расшифровку сообщения уходит гораздо больше времени, чем на его аутентификацию. Чтобы ускорить обработку сообщения, создадим два дополнительных экземпляра компонента расшифровки, как показано на рис. 3.5.

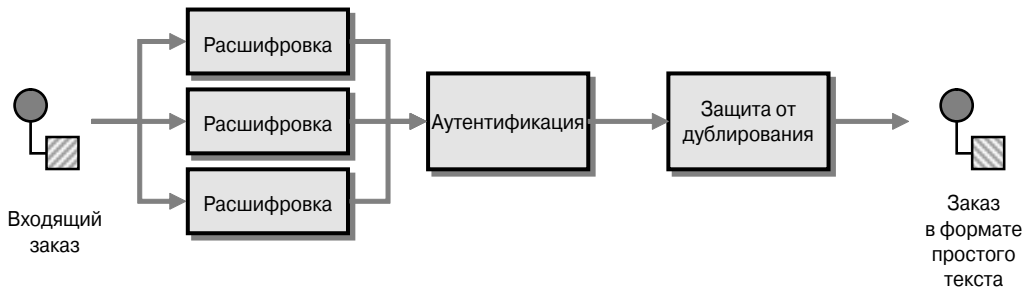


Рис. 3.5. Повышение пропускной способности системы за счет параллельной обработки сообщений

Использование параллельно выполняющихся экземпляров фильтра наиболее эффективно, если фильтр не сохраняет свое состояние, т.е. возвращается к исходному состоянию после обработки сообщения. Примером фильтра, сохраняющего состояние, является компонент, реализующий защиту от дублирования сообщений.

История архитектуры каналов и фильтров

Простота и эффективность *каналов и фильтров* (*Pipes and Filters*) завоевала этой архитектуре огромную популярность, а несложная семантика позволила описать ее формальными методами.

В 1974 году Кан определил вычислительную сеть Кана как набор параллельных процессов, объединенных неограниченными каналами FIFO (First-In, First-Out — первым прибыл, первым обслужен) [19]. В [11] есть глава, в которой описываются различные архитектурные стили, включая *каналы и фильтры*. Монро дает подробное толкование взаимоотношения между архитектурными стилями и шаблонами проектирования в [26]. Статья Режин Менье "The Pipes and Filters Architecture" ("Архитектура каналов и фильтров"), опубликованная в [31], послужила основой для шаблона проектирования *каналы и фильтры* (*Pipes and Filters*), включенного в [33]. Практически все реализации *каналов и фильтров* соответствуют представленной в [33] схеме "Scenario IV" ("Сценарий IV"), которая подразумевает наличие активных фильтров, независимо друг от друга считывающих, обрабатывающих и помещающих элементы в каналы. Шаблон, описанный в [33], предполагает, что каждый элемент проходит одинаковые этапы обработки при передаче от фильтра к фильтру. Это условие неизбежно нарушается, когда речь заходит о сценариях интеграции. В большинстве случаев маршрут сообщения определяется динамически на основе содержимого сообщения или административным путем. В действительности маршрутизация сообщений является настолько обыденной практикой в интеграции корпоративных приложений, что для ее описания предусмотрен отдельный шаблон проектирования *маршрутизатор сообщений* (*Message Router*, с. 109).

Словарь

Обсуждая архитектуру *каналов и фильтров* (*Pipes and Filters*), следует обратить внимание на корректность употребления термина *фильтр*. Позднее нами будут рассмотрены такие шаблоны проектирования, как *фильтр сообщений* (*Message Filter*, с. 253) и *фильтр содержимого* (*Content Filter*, с. 354). Следует отметить, что это далеко не единственные фильтры среди представленных в данной книге шаблонов. Дело в том, что *фильтр* в терминах архитектуры *фильтров и каналов* совсем не обязательно должен реализовывать функцию отбора (полей или сообщений). Чтобы избежать недоразумения, мы могли бы переименовать архитектуру *каналов и фильтров*, что, однако же, способно привести к еще большей путанице. Употребляя термин *фильтр*, мы постараемся делать это так, чтобы читателю было понятно, о чем идет речь — о фильтре архитектуры *каналов и фильтров* или о шаблонах проектирования *фильтр сообщений*/*фильтр содержимого*. Если контекст, в котором будет упоминаться термин *фильтр*, не позволит сделать однозначного вывода, для обозначения фильтра архитектуры *каналов и фильтров* будет применяться термин *компонент*.

Архитектура *каналов и фильтров* имеет сходство с концепцией CSP (Communicating Sequential Processes — взаимодействие последовательных процессов), впервые представленной Хоаром в 1978 году [6]. Хоар предлагает использовать простую модель для описания проблем синхронизации в системах параллельной обработки. Механизм, лежащий в основе CSP, применяется для синхронизации двух процессов посредством системы ввода-вывода. Взаимодействие между процессами осуществляется при условии готовности процесса А к передаче информации процессу Б, а процесса Б — к принятию информации от процесса А. Если это условие выполняется только наполовину, один из процессов ставится в очередь ожидания готовности другого процесса. В отличие от интеграционных решений концепция CSP предполагает более сильное связывание процессов и

отсутствие поддержки очередей "каналов". Тем не менее знакомство с ней очень полезно при изучении архитектуры *каналов и фильтров*.

***Пример:** простой фильтр MSMQ (C#)*

Ниже приведен код базового класса фильтра с одним входящим и одним исходящим портом. Функциональность класса `Processor` предельно проста: вывести тело полученного сообщения на консоль и отправить его через исходящий порт. Для реализации более полезной функциональности (например, преобразования формата содержимого сообщения или его перенаправления в другой порт) следует создать подкласс класса `Processor`, переопределяющий метод `ProcessMessage`.

Обратите внимание, что экземпляр класса `Processor` получает ссылки на входящий и исходящий каналы на этапе своего создания. Таким образом, класс `Processor` не привязан к каким-либо конкретным каналам или фильтрам, что позволяет создавать несколько экземпляров фильтров и объединять их в произвольном порядке.

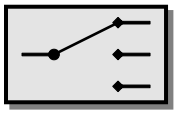
```
using System;
using System.Messaging;

namespace PipesAndFilters
{
    public class Processor
    {
        protected MessageQueue inputQueue;
        protected MessageQueue outputQueue;

        public Processor (MessageQueue inputQueue,
                          MessageQueue outputQueue)
        {
            this.inputQueue = inputQueue;
            this.outputQueue = outputQueue;
        }

        public void Process()
        {
            inputQueue.ReceiveCompleted += new
              ReceiveCompletedEventHandler(OnReceiveCompleted);
            inputQueue.BeginReceive();
        }

        private void OnReceiveCompleted(Object source,
                       ReceiveCompletedEventArgs asyncResult)
        {
            MessageQueue mq = (MessageQueue)source;

            Message inputMessage =
              mq.EndReceive(asyncResult.AsyncResult);
            inputMessage.Formatter = new XmlMessageFormatter
              (new String[] {"System.String,mscorlib"});
            Message outputMessage = ProcessMessage(inputMessage);

            outputQueue.Send(outputMessage);
```

```
            mq.BeginReceive();
        }

        protected virtual Message ProcessMessage(Message m)
        {
            Console.WriteLine("Received Message: " + m.Body);
            return (m);
        }
    }
}
```

Фильтр `Processor` является примером реализации *событийно управляемого потребителя* (*Event-Driven Consumer*, с. 511). Метод `Process` регистрируется для получения входящих сообщений и уведомляет систему обмена сообщениями о необходимости вызова метода `OnReceiveCompleted` при получении каждого сообщения. Метод `OnReceiveCompleted` извлекает данные из объекта события и вызывает виртуальный метод `ProcessMessage`.

Представленный выше фильтр не является транзакционным. Если во время обработки сообщения (до его помещения в исходящий канал) произойдет ошибка, сообщение будет потеряно. Решение данной проблемы описывается шаблоном проектирования *транзакционный клиент* (*Transactional Client*, с. 498).

Маршрутизатор сообщений (Message Router)



Несколько этапов обработки в цепочке *каналов и фильтров* (*Pipes and Filters*, с. 102) соединены с помощью *каналов сообщений* (*Message Channel*, с. 93).

Как реализовать возможность передачи сообщения различным фильтрам в зависимости от набора условий?

Архитектура *каналов и фильтров* предусматривает непосредственное соединение фильтров друг с другом с помощью фиксированных каналов. Подобный подход вполне оправдан, поскольку в большинстве случаев шаблон *каналы и фильтры* применяется для обработки больших наборов однотипных данных [33]. К примеру, при компиляции программного кода каждая его строка проходит три последовательных этапа обработки: лексический анализ, синтаксический анализ и семантический анализ. Сообщения, применяющиеся в интеграционных решениях, как правило, не связаны с каким-либо набором данных. В результате каждое сообщение может потребовать свою цепочку вычислений.

Канал сообщений позволяет провести четкую границу между отправителем и получателем *сообщения* (*Message*, с. 98). В частности, из этого следует, что несколько приложений могут публиковать *сообщения* в одном и том же *канале сообщений*. В результате *канал сообщений* может содержать сообщения, способ обработки которых будет зависеть от их типа, источника или какого-либо другого критерия. Несмотря на то что для передачи сообщения каждого типа можно задействовать отдельный *канал сообщений* (более подробно эта концепция описывается шаблоном проектирования *канал типа данных* (*Datatype Channel*, с. 139)), это приведет к необходимости классификации сообщений отправителями, а также к существенному росту числа *каналов сообщений*. К тому же способ обработки сообщения не всегда зависит от его источника. Представим ситуацию, в которой получатель сообщения определяется динамически на основе общего числа сообщений, переданных по конкретному каналу. Поскольку это число не может знать ни один из отправителей, он не сможет выбрать корректный канал для размещения сообщения.

Канал сообщений обеспечивает простую форму маршрутизации сообщений. Публикуя *сообщение* в *канале сообщений*, приложение теряет контроль за его доставкой. Отныне маршрут *сообщения* зависит от компонента, находящегося на другом конце *канала сообщений*. К сожалению, этот тип "маршрутизации" не учитывает свойства самого сообщения и напоминает использование символа "|" для обработки текстовых файлов в системах UNIX. Подобно *каналу сообщений* символ "|" позволяет создать цепочку *каналов и фильтров*, однако на время ее существования все строки текстового файла будут проходить одни и те же этапы обработки.

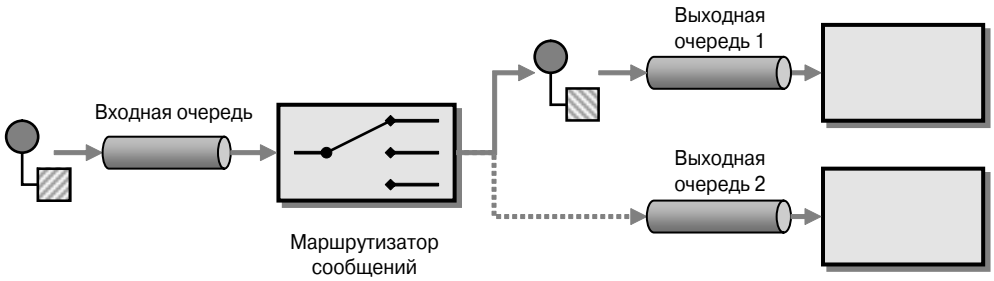
Решение о необходимости обработки сообщения, поступившего по общему *каналу сообщений*, можно возложить на получателя. Однако как только сообщение будет извлечено из канала, оно перестанет быть доступным для проверки другим компонентам. Некоторые системы обмена сообщениями позволяют просмотреть свойства сообщения без его извлечения из канала. Тем не менее это решение не является универсальным и к тому же "привязывает" компонент к определенному типу сообщения. Встраивание логики выбо-

ра сообщения в компонент затрудняет его повторное использование и сводит на нет основное преимущество модели *каналов и фильтров* — возможность переупорядочивания компонентов.

Большинство альтернативных подходов предполагает внесение изменений в компоненты. Однако компоненты интеграционных решений — это, как правило, крупные приложения, модифицировать которые зачастую не представляется возможным. Следовательно, изменение отправителей или получателей сообщений для удовлетворения требований системы обмена сообщениями не только экономически невыгодно, но и в большинстве случаев попросту невозможно.

Как уже упоминалось, одним из преимуществ архитектуры *каналов и фильтров* является возможность переупорядочивания компонентов. В частности, это позволяет добавлять дополнительные этапы (например, этап, реализующий логику маршрутизации) в цепочку фильтров, не затрагивая существующие компоненты.

Добавьте специальный фильтр — *маршрутизатор сообщений* (*Message Router*), — который будет извлекать *сообщение* (*Message*, с. 98) из одного *канала сообщений* (*Message Channel*, с. 93) и размещать его в другом *канале сообщений* на основе заданного условия.



Маршрутизатор сообщений не соответствует базовому определению архитектуры *каналов и фильтров* в том смысле, что он имеет несколько исходящих каналов (т.е. более одного исходящего порта). Вместе с тем особенность этой архитектуры позволяет скрыть существование *маршрутизатора сообщений* от соседних с ним компонентов — как и прежде, они извлекают сообщения из входящего канала, обрабатывают их и помещают в исходящий канал. Ключевое свойство *маршрутизатора сообщений* состоит в том, что он не изменяет содержимого сообщения, заботясь только о его корректной доставке.

Использование *маршрутизатора сообщений* позволяет сосредоточить всю логику принятия решения о доставке сообщения в одном компоненте. Отныне определение новых типов сообщений, добавление новых этапов обработки или изменение правил маршрутизации затронет только один компонент, а именно *маршрутизатор сообщений*. Кроме того, прохождение всех входящих сообщений через *маршрутизатор сообщений* делает возможной их обработку в корректном порядке.

К сожалению, *маршрутизатору сообщений* свойственны определенные недостатки. В частности, он должен обладать информацией обо всех доступных каналах сообщений. Если эти сведения часто меняются, *маршрутизатор сообщений* может превратиться в "узкое место" интеграционного решения. В этом случае рекомендуется переложить от-

ветственность за выбор сообщения на получателя, воспользовавшись *каналом "публикация–подписка"* (*Publish-Subscribe Channel*, с. 134) и набором *фильтров сообщений* (*Message Filter*, с. 253). Метод, подразумевающий использование *маршрутизатора сообщений*, получил название *предиктивная маршрутизация*, а метод, возлагающий ответственность за выбор сообщения на получателя, — *реактивная фильтрация* (более подробно сравнение двух подходов приводится в описании шаблона *фильтр сообщений* в главе 7).

Поскольку использование *маршрутизатора сообщений* подразумевает добавление нового этапа обработки, это приводит к снижению производительности интеграционного решения. Чтобы минимизировать негативный эффект предиктивной маршрутизации, можно запустить несколько параллельных процессов *маршрутизатора сообщений* или установить дополнительное оборудование. Это позволит сохранить пропускную способность системы (количество сообщений, обработанных за единицу времени) на прежнем уровне, однако практически неизбежно увеличит ее задержку (время, требующееся для передачи одного сообщения).

Применение *маршрутизаторов сообщений* способно превратить преимущество слабосвязанной системы в ее недостаток. Зачастую слабая связь между компонентами решения мешает отслеживанию потоков сообщений. Использование *маршрутизаторов сообщений* только усугубляет положение. Невозможность определения потока сообщений затрудняет тестирование, отладку и поддержку интеграционного решения. Для устранения этой проблемы можно воспользоваться *журналом доставки сообщения* (*Message History*, с. 561), в который заносится информация о компонентах, через которые прошло сообщение. Альтернативный подход заключается в составлении списка всех каналов, к которым подключены компоненты системы, что позволит создать граф допустимых потоков сообщений интеграционного решения. Многие средства интеграции корпоративных приложений (EAI) хранят подобную информацию в центральном репозитории, делая ее легко доступной для анализа.

Типы маршрутизаторов сообщений

Маршрутизатор сообщений (*Message Router*) может использовать несколько различных критериев при определении исходящего канала для отправки сообщения. Наиболее тривиальный пример *маршрутизатора сообщений* — это так называемый фиксированный маршрутизатор. Фиксированный маршрутизатор имеет один входящий и один исходящий канал. Функциональность фиксированного маршрутизатора предельно проста — извлечь сообщение из входящего канала и поместить его в исходящий канал. Зачем же нужен такой примитивный маршрутизатор? Во-первых, фиксированный маршрутизатор может использоваться как временное решение до реализации более интеллектуальной логики маршрутизации. Во-вторых, фиксированный маршрутизатор может применяться для перенаправления сообщений между несколькими интеграционными решениями. Зачастую вместе с фиксированным маршрутизатором используются *транслятор сообщений* (*Message Translator*, с. 115) и *адаптер канала* (*Channel Adapter*, с. 154).

Многие *маршрутизаторы сообщений* принимают решение о точке назначения сообщения на основании его свойств, например типа сообщения или значения определенных полей. Подобные маршрутизаторы получили название *маршрутизаторы на основе содержимого* (*Content-Based Router*, с. 247). Более подробно *маршрутизатор на основе содержимого* описывается соответствующим шаблоном проектирования.

Оставшиеся маршрутизаторы сообщений принимают решение о точке назначения сообщения на основании условий среды. Подобные маршрутизаторы получили название *маршрутизаторы на основе контекста*. Как правило, маршрутизаторы на основе контекста применяются для балансировки нагрузки, тестирования или обеспечения восстановления при отказе. К примеру, если компонент выйдет из строя, маршрутизатор на основе контекста сможет перенаправить сообщения на обработку другим компонентом с аналогичной функциональностью. Некоторые маршрутизаторы на основе контекста разделяют поток сообщений между несколькими каналами для достижения эффекта балансировки нагрузки. Следует отметить, что возможность балансировки нагрузки можно реализовать и без использования *маршрутизатора сообщений*. В частности, примером балансировки нагрузки является извлечение сообщений несколькими *конкурирующими потребителями* (*Competing Consumers*, с. 515) из одного и того же *канала сообщений* (*Message Channel*, с. 93) для параллельной обработки. Несмотря на это применение *маршрутизатора сообщений* позволит реализовать более интеллектуальную логику маршрутизации по сравнению с простой логикой алгоритма кругового обслуживания, реализуемой *каналом сообщений*.

Многие *маршрутизаторы сообщений* являются маршрутизаторами *без поддержки состояния*. Другими словами, принятие решения о передаче сообщения принимается без учета ранее переданных сообщений. Маршрутизаторы, которые учитывают обработанные ранее сообщения при принятии решения о точке назначения текущего сообщения, называются маршрутизаторами *с поддержкой состояния*. Примером маршрутизатора с поддержкой состояния является компонент, реализующий защиту от дублирования сообщений из рассмотренного ранее примера использования архитектуры *каналов и фильтров* (*Pipes and Filters*, с. 102).

В большинстве случаев логика маршрутизации жестко закодирована в *маршрутизаторе сообщений*. Тем не менее некоторые маршрутизаторы могут принимать решение о перенаправлении сообщения на основе информации, полученной по *шине управления* (*Control Bus*, с. 552). Этот подход позволяет изменять критерий маршрутизации, не внося изменений в маршрутизатор и не затрагивая текущий поток сообщений. К примеру, *шина управления* может передать значение некоторой глобальной переменной всем *маршрутизаторам сообщений* в системе для перевода последней из тестового режима функционирования в рабочий. *Динамический маршрутизатор* (*Dynamic Router*, с. 259) обладает способностью к изменению логики маршрутизации на основании управляющих сообщений, полученных от конечных точек системы.

Более подробно различные типы *маршрутизаторов сообщений* рассматриваются в главе 7.

***Пример:** коммерческие средства EAI*

Концепция *маршрутизатора сообщений* (*Message Router*) является основой для *брокера сообщений* (*Message Broker*, с. 334) — средства, реализованного практически во всех EAI-пакетах. *Брокер сообщений* принимает входящие сообщения, тестирует их на предмет наличия ошибок, преобразовывает и помещает в требуемый исходящий канал. Подобная архитектура позволяет максимально изолировать приложения друг от друга, что чрезвычайно важно для их интеграции. *Брокер сообщений* реализует всю логику, требующуюся для маршрутизации сообщений между приложениями, и, по сути, является "интеграционным" аналогом *посредника* (*Mediator*) [12].

***Пример:** простой маршрутизатор MSMQ (C#)*

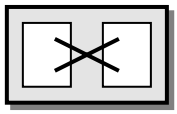
Следующий код демонстрирует пример реализации простого маршрутизатора, перенаправляющего все входящие сообщения в один из двух каналов на основе заданного условия.

```
class SimpleRouter
{
    protected MessageQueue inQueue;
    protected MessageQueue outQueue1;
    protected MessageQueue outQueue2;

    public SimpleRouter(MessageQueue inQueue,
     MessageQueue outQueue1, MessageQueue outQueue2)
    {
        this.inQueue = inQueue;
        this.outQueue1 = outQueue1;
        this.outQueue2 = outQueue2;

        inQueue.ReceiveCompleted += new
ReceiveCompletedEventHandler(OnMessage);
        inQueue.BeginReceive();
    }

    private void OnMessage(Object source,
      ReceiveCompletedEventArgs asyncResult)
    {
        MessageQueue mq = (MessageQueue)source;
        Message message = mq.EndReceive(asyncResult.AsyncResult);

        if (IsConditionFulfilled())
            outQueue1.Send(message);
        else
            outQueue2.Send(message);
        mq.BeginReceive();
    }

    protected bool toggle = false;

    protected bool IsConditionFulfilled ()
    {
        toggle = !toggle;
        return toggle;
    }
}
```

Подобно простому фильтру, представленному при описании архитектуры *каналов и фильтров* (*Pipes and Filters*, с. 102), класс `SimpleRouter` реализует *событийно управляемый потребитель* (*Event-Driven Consumer*, с. 511) сообщений с использованием делегатов. Конструктор регистрирует метод `OnMessage` в качестве обработчика сообщений, поступающих по каналу `inQueue`. Среда .NET будет вызывать метод `OnMessage` для обработки каждого сообщения, извлеченного из канала `inQueue`. Метод `OnMessage` определяет маршрут сообщения с помощью метода `IsConditionFulfilled`. В рассмотренном выше примере метод `IsConditionFulfilled` реализует простую логику маршрутизации,

поочередно помещая сообщения в каналы `outQueue1` и `outQueue2`. Маршрутизатор `SimpleRouter` не является транзакционным. Если во время обработки сообщения (до его помещения в исходящий канал) произойдет ошибка, сообщение будет потеряно. Решение данной проблемы описывается шаблоном проектирования *транзакционный клиент* (*Transactional Client*, с. 498).

Транслятор сообщений (Message Translator)



Ранее мы рассмотрели шаблоны, касающиеся создания сообщений и их доставки получателю. Большинство интеграционных решений объединяет разнородные приложения — унаследованные, коммерческие и созданные на заказ. Как правило, каждое из этих приложений использует собственную модель данных. К примеру, система бухгалтерского учета может оперировать такими данными о заказчике, как номер налогоплательщика, а система управления взаимоотношениями с клиентами (CRM) — телефонным номером и адресом проживания. Зачастую модель данных приложения является основой для схемы базы данных, формата файлов и интерфейса API — точек "соприкосновения" приложения с интеграционным решением.

Вдобавок к внутренним моделям данных приложений интеграционное решение должно поддерживать стандартные форматы данных, применяющиеся для взаимодействия с внешними приложениями. Стандартные протоколы и форматы данных создаются различными консорциумами и комитетами по стандартизации, такими как RosettaNet, ebXML, OAGIS и др. В большинстве случаев "официальные" форматы данных используются для взаимодействия интеграционного решения с приложениями бизнес-партнеров и внешних организаций.

Как организовать обмен сообщениями между приложениями, использующими различные форматы данных?

Если бы все интегрируемые приложения использовали одинаковые форматы данных, задача преобразования сообщений решилась бы сама собой. К сожалению, это очень сложно реализовать по целому ряду причин (см. шаблон проектирования *общая база данных* (*Shared Database*, с. 83)). Во-первых, изменение внутреннего формата данных требует внесения большого объема изменений в бизнес-логику приложения, что экономически нецелесообразно для большинства унаследованных приложений (вспомним попытку изменения единственного поля в рамках "проблемы 2000").

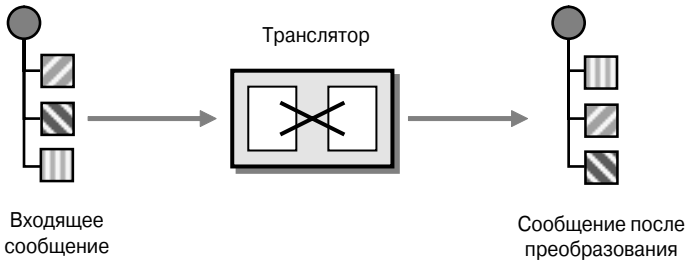
Во-вторых, использование одинаковых имен полей и, возможно, типов данных может сочетаться с различным представлением информации, к примеру XML-документ и файл данных COBOL.

В-третьих, приведение формата данных одного приложения к формату данных другого приложения приводит к их сильному связыванию. Как известно, одним из ключевых принципов проектирования интеграционных решений является слабая связь между объединяемыми приложениями (см. шаблон проектирования *каноническая модель данных* (*Canonical Data Model*, с. 367)). Идентичность форматов данных приложений нарушает этот принцип. Отныне замена приложения или внесение в него изменений (сценарий, присущий интеграционным решениям) будет иметь существенные последствия для остальных приложений.

Функция преобразования формата данных может быть встроена в *конечную точку сообщения* (*Message Endpoint*, с. 124). В результате все приложения будут отправлять и принимать сообщения общего формата данных. Однако этот подход предполагает наличие

доступа к исходному коду конечной точки, что возможно далеко не всегда. К тому же встраивание кода преобразования в конечную точку снижает возможность его повторного использования.

Используйте для преобразования формата данных специальный фильтр — *транслятор сообщений* (*Message Translator*), — расположив его между другими фильтрами или приложениями.



Транслятор сообщений (*Message Translator*) — эквивалент шаблона проектирования *адаптер* (*Adapter*), описанного в [12]. *Адаптер* преобразует интерфейс компонента в другой интерфейс, который может быть использован в отличном контексте.

Уровни преобразования

Преобразование сообщений может осуществляться на нескольких различных уровнях. К примеру, элементы данных приложений могут иметь одни и те же имя и тип, однако отличаться представлением (XML-документ, файл с разделителями-запятыми и т.д.). С другой стороны, элементы данных могут иметь одинаковые форматы (например, XML), однако отличаться именами. Обобщим различные типы преобразования сообщений, разделив их на несколько уровней (табл. 3.1).

Таблица 3.1. Уровни преобразования сообщений

Уровень	Объект преобразования	Пример	Средства/методы
Структуры данных (приложения)	Сущность, ассоциация, кардинальность	Приведение отношения “множество-множество” к агрегации	Шаблоны структурного преобразования, заказной код
Типа данных	Имя поля, тип данных, область определения значения, ограничение, значение кода	Преобразование почтового индекса из числа в строку; объединение полей `First Name` и `Last Name` в поле `Name`; замена названия штата США двузначным кодом	Визуальные редакторы преобразования EAI, XSL, обращение к базе данных, заказной код

Окончание табл. 3.1

Уровень	Объект преобразования	Пример	Средства/методы
Представления данных	Формат данных (XML, пары "имя–значение", поля данных фиксированной длины, форматы EAI-вендоров и др.), набор символов (ASCII, Unicode, EBCDIC), шифрование/сжатие	Анализ представления данных с последующим преобразованием в другой формат; шифрование/расшифровка данных	Анализаторы XML, анализаторы/генераторы EAI, заказные API
Транспортный	Коммуникационные протоколы: сокеты TCP/IP, HTTP, SOAP, JMS, TIBCO RendezVous	Передача данных с использованием одного из коммуникационных протоколов без изменения содержимого сообщения	Адаптер канала (Channel Adapter, с. 154), адаптеры EAI

Транспортный уровень обеспечивает передачу данных между различными системами. На этом уровне осуществляется надежный обмен данными между сетевыми сегментами, включающий исправление ошибок. Некоторые EAI-вендоры разрабатывают собственные транспортные протоколы (например, TIBCO RendezVous), в то время как остальные технологии интеграции основываются на использовании протоколов TCP/IP (например, SOAP). Преобразование сообщений между различными транспортными уровням можно реализовать с помощью *адаптера канала* (*Channel Adapter*, с. 154).

Уровень *представления данных* часто называют *уровнем синтаксиса*. Как следует из его названия, этот уровень определяет представление передаваемых данных. Поскольку на транспортном уровне осуществляется передача потока символов или байтов, сложные структуры данных должны быть преобразованы в строку с помощью одного из распространенных форматов — XML, полей фиксированной длины (например, записей EDI) и др. Во многих случаях данные сжимаются и (или) шифруются, в результате чего к ним добавляется дополнительная информация, такая как контрольная сумма или цифровой сертификат. Объединение систем, использующих различные представления данных, зачастую предполагает расшифровку, распаковку и анализ данных с последующим созданием нового представления, сжатием и шифрованием (при необходимости).

Уровень *типа данных* определяет типы данных, на которых основана модель приложения. На этом уровне принимаются такие решения, как способ представления полей даты (строка или специальная структура), формат почтового индекса и т.п. Большинство подобных решений фиксируется в так называемом *словаре данных*. Иногда вопросы, касающиеся типов данных, выходят за пределы выбора между строковым и числовым представлениями значения. Предположим, что информация о продажах товаров сгруппирована по регионам. Приложение, установленное в отделе продаж, разделяет страну на четыре региона — Запад, Центр, Юг и Восток, — для представления которых используются буквы "З", "Ц", "Ю" и "В" соответственно. В то же время приложение, установленное в отделе маркетинга, выделяет три дополнительных региона — Тихоокеанский, Северо-Восточный и Юго-Восточный, — используя для представления регионов двузначные числа. Какое число следует поставить в соответствие букве "В"?

Уровень *структуры данных* описывает данные с точки зрения модели домена приложения (именно поэтому его вторым названием является *уровень приложения*). Этот уровень определяет логические сущности, которыми оперирует приложение (например, “заказчик”, “адрес” или “счет”), а также взаимоотношения между ними (может ли один заказчик иметь несколько счетов (адресов), могут ли несколько заказчиков иметь одинаковые адреса (счета), частью какой сущности (счета или заказчика) является адрес и т.д.). Зачастую на уровне структуры данных используются диаграммы сущностей и связей, а также классовые диаграммы.

Уровни связывания

Многие компромиссные решения, принимаемые при интеграции приложений, связаны с необходимостью уменьшения зависимости между объединяемыми системами. Слабое связывание интегрированных приложений создает основу для эффективного управления изменениями. Благодаря *каналам сообщений* (*Message Channel*, с. 93) приложения могут не знать расположение друг друга. *Маршрутизатор сообщений* (*Message Router*, с. 109) избавляет приложения от необходимости иметь общий *канал сообщений*. Несмотря на это приложения могут все еще сильно зависеть друг от друга по причине связанности внутренних форматов данных. Именно этот уровень связывания и призван устранить *транслятор сообщений* (*Message Translator*, с. 115).

Цепочечные преобразования

Многие сценарии бизнес-интеграции предполагают преобразование данных на нескольких различных уровнях. Предположим, что запись заказа покупок EDI 850, представленную файлом с фиксированным форматом, следует преобразовать в XML-документ и отправить по протоколу HTTP в систему управления заказами, использующую отличное определение объекта `Order`. Необходимые при этом преобразования охватывают все четыре уровня: преобразование способа транспортировки данных с передачи файла на передачу по протоколу HTTP; преобразование представления данных с файла фиксированного формата в документ XML; а также преобразование типа и структуры данных в соответствии с определением объекта `Order` системы управления заказами. Преимущество рассмотренной ранее модели многоуровневого преобразования состоит в независимости каждого уровня от остальных уровней, как показано на рис. 3.6.

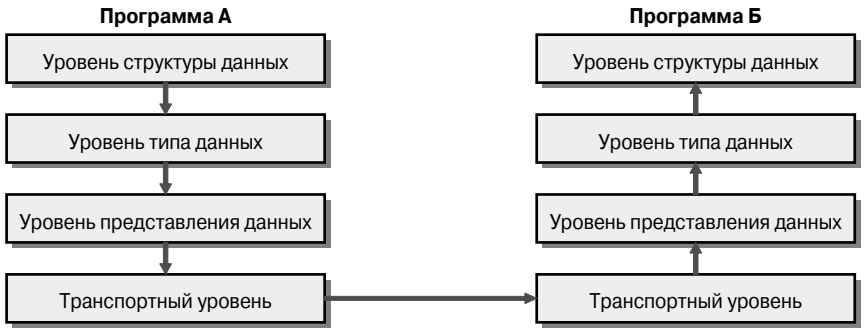


Рис. 3.6. Взаимоотношения между уровнями преобразования

На рис. 3.7 показан результат объединения нескольких *трансляторов сообщений* (*Message Translator*) с помощью архитектуры *каналов и фильтров* (*Pipes and Filters*, с. 102).

Создание одного *транслятора сообщений* для каждого уровня преобразования позволяет использовать этот компонент в других сценариях интеграции. К примеру, *адаптер канала* (*Channel Adapter*, с. 154) и *транслятор сообщений* EDI-в-XML представляют собой универсальное сочетание компонентов, которое можно применять при обработке любых входящих документов EDI.

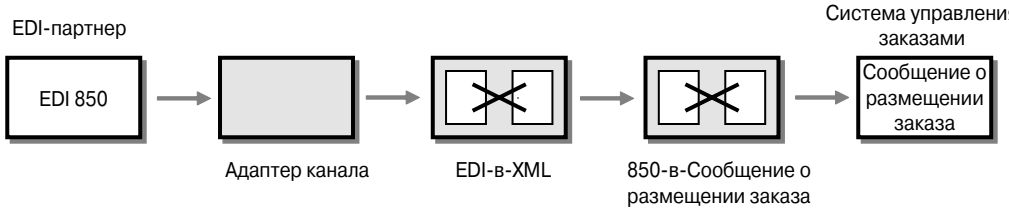


Рис. 3.7. Соединение в цепочку нескольких трансляторов сообщений (*Message Translator*)

Объединение нескольких *трансляторов сообщений* в цепочку позволяет при необходимости вносить изменения в ее звенья без затрагивания оставшихся компонентов. К примеру, замена функции преобразования в файл фиксированного формата функцией преобразования в файл с разделителями-запятыми сводится к замене компонента на соответствующем уровне преобразования.

Существует несколько различных разновидностей *трансляторов сообщений*. *Оболочка конверта* (*Envelope Wrapper*, с. 342) предусматривает помещение данных сообщения в "конверт" для последующей передачи через систему обмена сообщениями. *Расширитель содержимого* (*Content Enricher*, с. 348) добавляет информацию в сообщение, а *фильтр содержимого* (*Content Filter*, с. 354) удаляет ее. Шаблон проектирования *квитанция* (*Claim Check*, с. 358) предполагает удаление информации для ее последующего извлечения. *Нормализатор* (*Normalizer*, с. 364) преобразует несколько различных форматов сообщений в общий формат. Наконец, *каноническая модель данных* (*Canonical Data Model*, с. 367) демонстрирует пример совместного использования нескольких *трансляторов сообщений* для устранения различий между внутренними форматами данных объединяемых приложений. Каждый из перечисленных шаблонов проектирования реализует сложные структурные преобразования, такие как отображение взаимоотношения "множество–множество" на взаимоотношение "один–к–одному".

***Пример:** структурные преобразования XSL*

Консорциум W3C определил стандартный язык преобразования XML-документов — Extensible Stylesheet Language (расширяемый язык стилей, XSL). Частью XSL является основанный на правилах язык XSL Transformation (преобразование XSL, XSLT), использующийся для преобразования формата документа XML. Рассмотрим конкретный пример, демонстрирующий использование языков XSL и XSLT (более подробно язык XSLT описывается в [40, 49]).

Рассмотрим задачу передачи входящего XML-документа в систему бухгалтерского учета. При отсутствии отличий на уровне представления данных (XML) остается устранить рассогласование имен полей, типов данных и их структуры. Исходный XML-документ показан ниже.

```
<data>
    <customer>
        <firstname>Joe</firstname>
        <lastname>Doe</lastname>
        <address type="primary">
            <ref id="55355"/>
        </address>
        <address type="secondary">
            <ref id="77889"/>
        </address>
    </customer>
    <address id="55355">
        <street>123 Main</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94123</postalcode>
        <country>USA</country>
        <phone type="cell">
            <area>415</area>
            <prefix>555</prefix>
            <number>1234</number>
        </phone>
        <phone type="home">
            <area>415</area>
            <prefix>555</prefix>
            <number>5678</number>
        </phone>
    </address>
    <address id="77889">
        <company>ThoughtWorks</company>
        <street>410 Townsend</street>
        <city>San Francisco</city>
        <state>CA</state>
        <postalcode>94107</postalcode>
        <country>USA</country>
    </address>
</data>
```

XML-документ содержит сведения о заказчике. Каждый заказчик может иметь несколько адресов, каждый из которых, в свою очередь, может содержать несколько телефонных номеров. Поскольку адреса представлены как независимые сущности, несколько заказчиков могут иметь один адрес.

Предположим, что система бухгалтерского учета ожидает получить XML-документ следующего вида (использование немецкого языка в именах тэгов не является чем-то сверхъестественным в области интеграции корпоративных приложений).

```
<Kunde>
    <Name>Joe Doe</Name>
    <Adresse>
        <Strasse>123 Main</Strasse>
        <Ort>San Francisco</Ort>
        <Telefon>415-555-1234</Telefon>
    </Adresse>
</Kunde>
```

Приведенный выше XML-документ имеет более простую структуру, чем исходный. Кроме отличий в именах тэгов, некоторые поля исходного документа объединяются в единственное поле в целевом документе. Поскольку результирующая структура не позволяет заказчику иметь несколько адресов или телефонных номеров, необходимо выбрать соответствующие данные из исходного документа на основании существующих бизнес-правил. Следующая XSLT-программа выполняет преобразование формата исходного XML-документа в требуемый формат.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:key name="addrlookup" match="/data/address" use="@id"/>
    <xsl:template match="data">
        <xsl:apply-templates select="customer"/>
    </xsl:template>
    <xsl:template match="customer">
        <Kunde>
            <Name>
                <xsl:value-of select="concat(firstname, ' ',
                  lastname)"/>
            </Name>
            <Adresse>
                <xsl:variable name="id"
                  select="./address[@type='primary']/ref/@id"/>
                <xsl:call-template name="getaddr">
                    <xsl:with-param name="addr"
                      select="key('addrlookup', $id)"/>
                </xsl:call-template>
            </Adresse>
        </Kunde>
    </xsl:template>
    <xsl:template name="getaddr">
        <xsl:param name="addr"/>
        <Strasse>
            <xsl:value-of select="$addr/street"/>
        </Strasse>
        <Ort>
            <xsl:value-of select="$addr/city"/>
        </Ort>
        <Telefon>
```

```
            <xsl:choose>
                <xsl:when test="$addr/phone[@type='cell']">
                    <xsl:apply-templates
                      select="$addr/phone[@type='cell']"
                      mode="getphone"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:apply-templates
                      select="$addr/phone[@type='home']"
                      mode="getphone"/>
                </xsl:otherwise>
            </xsl:choose>
        </Telefon>
    </xsl:template>
    <xsl:template match="phone" mode="getphone">
        <xsl:value-of select="concat(area, '-',
          prefix, '-', number)"/>
    </xsl:template>
    <xsl:template match="*"/>
</xsl:stylesheet>
```

Приведенный выше код XSL основан на сопоставлении фрагментов текста. Коротко говоря, инструкции внутри элемента `<xsl:template>` выполняются всякий раз, когда в XML-документе встречается фрагмент, определенный атрибутом `match`. К примеру, строка

```
<xsl:template match="customer">
```

указывает на необходимость выполнения последующих инструкций при обнаружении в исходном XML-документе тэга `<customer>`. В данном случае это приводит к формированию элемента `<Name>`, состоящего из имени и фамилии заказчика, а также элемента `<Adresse>`. Для извлечения из исходного XML-документа корректного адреса и телефонного номера код XSL вызывает процедуру `getaddr`. По умолчанию процедура `getaddr` извлекает номер мобильного телефона, а если он не определен — номер домашнего телефона.

***Пример:** визуальные средства преобразования*

Большинство разработчиков средств интеграции включают в состав своих продуктов визуальный редактор преобразований, отображающий на экране структуру исходного и целевого форматов документа. Пользователю предоставляется возможность ассоциировать требуемые элементы путем их соединения линиями. Подобный подход к выполнению преобразований гораздо проще, нежели код XSL. Некоторые вендоры, такие как Contivo, специализируются исключительно на создании средств преобразования.

На рис. 3.8 показано окно редактора преобразования Microsoft BizTalk Mapper, интегрированного в Visual Studio.

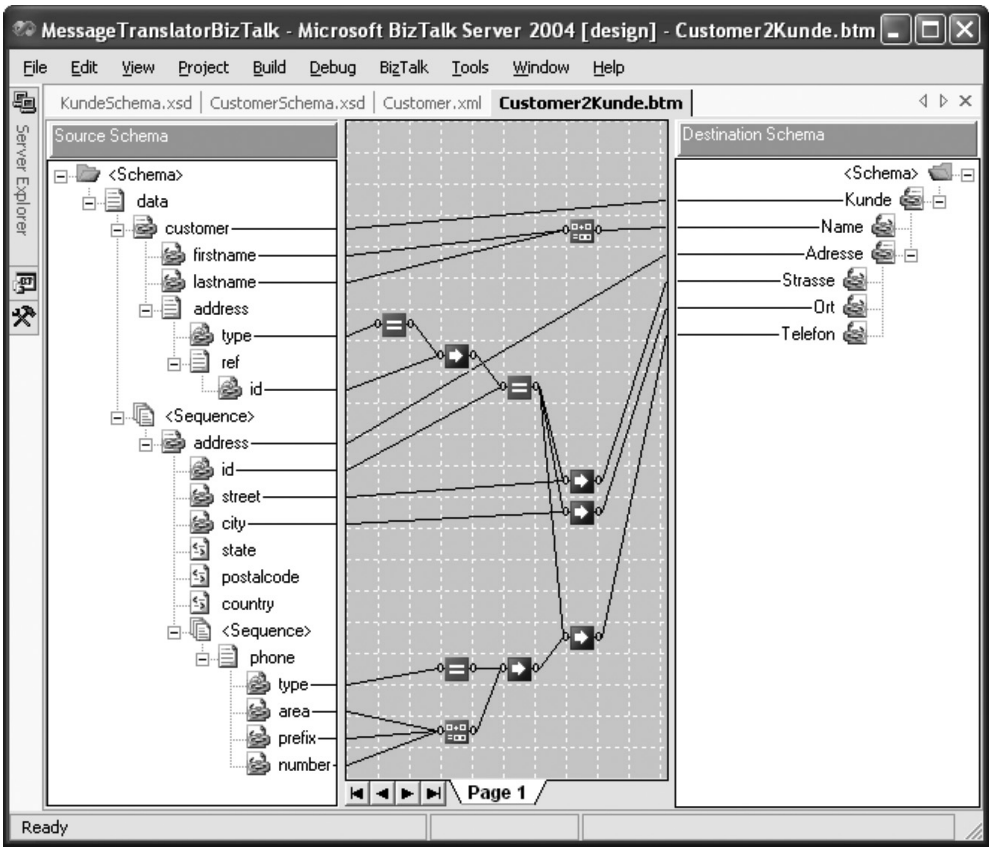
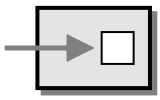


Рис. 3.8. Создание преобразования с помощью перетаскивания

Безусловно, преобразование элементов гораздо удобнее наблюдать на диаграмме, чем в коде XSL. С другой стороны, детали реализации (например, логика выбора телефонного номера) скрыты за так называемыми функциональными пиктограммами.

Возможность создания преобразования путем перетаскивания существенно упрощает разработку *транслятора сообщений* (*Message Translator*). Тем не менее как только речь заходит об отладке кода или создании сложного интеграционного решения, визуальное представление преобразования становится скорее недостатком, чем достоинством. Именно поэтому многие визуальные средства преобразования позволяют переключаться между визуальным представлением и кодом XSL.

Конечная точка сообщения (Message Endpoint)



Приложения обмениваются *сообщениями* (*Message*, с. 98) по *каналам сообщений* (*Message Channel*, с. 93).

Как подключить приложение к каналу системы обмена сообщениями?

Приложение и система обмена сообщениями представляют собой две отдельные программные сущности. Приложение обеспечивает функциональность для пользователей, в то время как система обмена сообщениями управляет каналами, применяющимися для передачи сообщений. Даже будучи встроенной в приложение, система обмена сообщениями предоставляет обособленные, специализированные функции подобно СУБД или Web-серверу. В связи с этим возникает задача подключения приложения к системе обмена сообщениями (рис. 3.9).

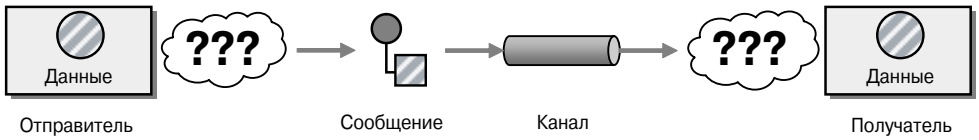
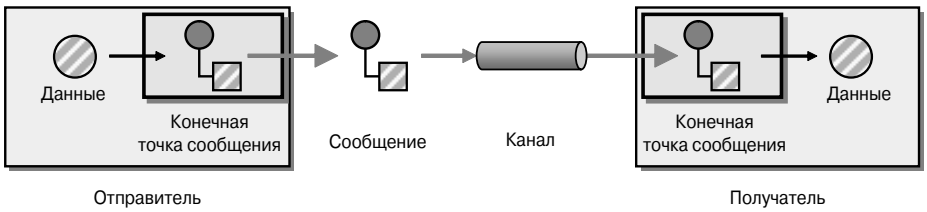


Рис. 3.9. Приложения не подключены к системе обмена сообщениями

Система обмена сообщениями представляет собой определенный тип сервера, принимающего запросы и отвечающего на них. Подобно базе данных система обмена сообщениями принимает и доставляет сообщения. Таким образом, система обмена сообщениями является сервером сообщений.

Клиентом сервера сообщений является приложение, взаимодействующее с другими приложениями с помощью обмена сообщениями. Подобно серверу баз данных сервер сообщений предоставляет клиентский API, с помощью которого приложение может взаимодействовать с сервером. Поскольку клиентский API сервера сообщений отражает специфику конкретной системы обмена сообщениями, приложение должно включать в себя код подключения к системе обмена сообщениями.

Подключите приложение к каналу обмена сообщениями с помощью *конечной точки сообщения* (*Message Endpoint*) — клиента системы обмена сообщениями, позволяющего приложению отправлять и принимать *сообщения* (*Message*, с. 98).



Код *конечной точки сообщения* (*Message Endpoint*) создается с учетом особенностей конкретного приложения и клиентского API системы обмена сообщениями. Оставшаяся часть приложения не обладает сведениями о формате сообщений, каналах и других подробностях взаимодействия посредством обмена сообщениями. Конечная точка сообщения принимает от приложения команду или данные, формирует из них сообщение и отправляет его в требуемый канал сообщений. Кроме того, конечная точка получает сообщение от системы обмена сообщениями и передает его содержимое приложению в понятном для него формате.

Конечная точка сообщения инкапсулирует систему обмена сообщениями от приложения. Таким образом, внесение изменений в приложение или в клиентский API системы обмена сообщениями приведет к необходимости правки кода только конечной точки сообщения.

Конечная точка сообщения может использоваться для отправки или получения сообщений, однако она не может совмещать в себе обе функции. Кроме того, конечная точка сообщения создается для обслуживания конкретного канала сообщений. Таким образом, для взаимодействия по нескольким каналам сообщений приложение должно располагать соответствующим числом конечных точек сообщения. Приложение может использовать несколько конечных точек сообщения и при обслуживании одного канала, как правило, для поддержки нескольких параллельных потоков сообщений.

Конечная точка сообщения является частным случаем *адаптера канала* (*Channel Adapter*, с. 154), интегрированного в приложение и созданного для его подключения к конкретной системе обмена сообщениями.

Конечная точка сообщения должна проектироваться как *шлюз обмена сообщениями* (*Messaging Gateway*, с. 482), что позволит скрыть систему обмена сообщениями от приложения. Она может включать в себя *преобразователь обмена сообщениями* (*Messaging Mapper*, с. 491) для передачи данных между объектами домена и сообщениями. Конечная точка сообщения может быть структурирована как *активатор службы* (*Service Activator*, с. 545), обеспечивая асинхронный доступ к синхронной службе или вызову функции. Наконец, она может выступать в роли *транзакционного клиента* (*Transactional Client*, с. 498) по отношению к системе обмена сообщениями.

Различные типы конечных точек реализуют разные подходы к получению сообщений. Получатель сообщений может быть как *опрашивающим потребителем* (*Polling Consumer*, с. 507), так и *событийно управляемым потребителем* (*Event-Driven Consumer*, с. 511). Несколько потребителей могут извлекать сообщения из одного и того же канала с помощью *диспетчера сообщений* (*Message Dispatcher*, с. 521) или в соответствии с моделью *конкурирующих потребителей* (*Competing Consumers*, с. 515). Для реализации логики фильтра сообщений получатель может применить шаблон *избирательного потребителя* (*Selective Consumer*, с. 528). С помощью шаблона *постоянный подписчик* (*Durable Subscriber*, с. 535) получатель гарантированно не пропустит ни одного сообщения, а с помощью шаблона *идемпотентный получатель* (*Idempotent Receiver*, с. 541) обеспечит защиту от дублирования сообщений.

Пример: *классы* `MessageProducer` *и* `MessageConsumer` *(JMS)*

Два основных типа конечных точек сообщений в JMS представлены классами `MessageProducer` (используется для отправки сообщений) и `MessageConsumer` (используется для получения сообщений). *Конечная точка сообщения* (*Message Endpoint*) использует экземпляр одного из этих классов для отправки или получения сообщений по соответствующему каналу.

Пример: *класс* `MessageQueue` *(.NET)*

В .NET *конечная точка сообщения* (*Message Endpoint*) и *канал сообщений* (*Message Channel*, с. 93) представлены одним и тем же классом — `MessageQueue`. Конечная точка сообщения использует экземпляр класса `MessageQueue` для отправки или получения сообщений по соответствующему каналу.