57

Технология Windows Workflow Foundation

В ЭТОЙ ГЛАВЕ...

- Различные типы рабочих потоков: последовательные и конечные автоматы
- Встроенные действия
- Создание специальных действий
- Службы рабочих потоков
- Интеграция с WCF
- Хостинг рабочих потоков
- Советы по переходу на Workflow Foundation 4

В настоящей главе представлен обзор платформы рабочих потоков Windows Workflow Foundation 3.0 (WF), предлагающей модель, в которой можно определять и выполнять процессы с использованием набора строительных блоков — *действий* (activity). Технология WF поддерживает визуальный конструктор, по умолчанию развернутый в среде Visual Studio, который позволяет перетаскивать действия из панели инструментов на поверхность проектирования, создавая шаблон рабочего потока.

Этот шаблон затем может быть выполнен за счет создания экземпляра рабочего потока `WorkflowInstance` и запуска этого экземпляра. Код, выполняющий рабочий поток, известен как исполняющая среда рабочего потока — `WorkflowRuntime` — и этот объект также может содержать в себе множество служб, к которым может обращаться рабочий поток. В любое время одновременно могут выполняться несколько экземпляров рабочего потока, и исполняющая среда занимается планированием этих экземпляров, сохраняя и восстанавливая их состояние; она также может записывать поведение каждого экземпляра рабочего потока по мере его работы.

Рабочий поток конструируется из ряда действий, и эти действия запускаются в исполняющей среде. Действием может быть отправка электронной почты, обновление строки в базе данных либо выполнение транзакции в системе базы данных. Существует ряд встроенных действий, которые могут быть использованы для выполнения работ общего назначения, и, кроме того, при необходимости можно создавать собственные действия и подключать их к рабочему потоку.

Обратите внимание, что в составе Visual Studio 2010 поставляется новая версия Windows Workflow, не поддерживающая обратной совместимости. В этой главе описана предшествующая версия; однако в новых проектах рекомендуется отдавать предпочтение версии WF 4. За подробностями обращайтесь к главе 44. В конце этой главы также включено несколько рекомендаций для облегчения перехода на новую версию.

Мы начнем с канонического примера, с которым имеет дело каждый, кто впервые сталкивается с новой технологией — "Hello World", — а также опишем все, что понадобится для получения выполняющегося рабочего потока на машине разработчика.

Пример "Hello World"

Visual Studio 2010 содержит встроенную поддержку создания рабочих потоков, и когда вы откроете диалоговое окно New Project (Новый проект), то увидите список типов проектов рабочего потока, как показано на рис. 57.1.

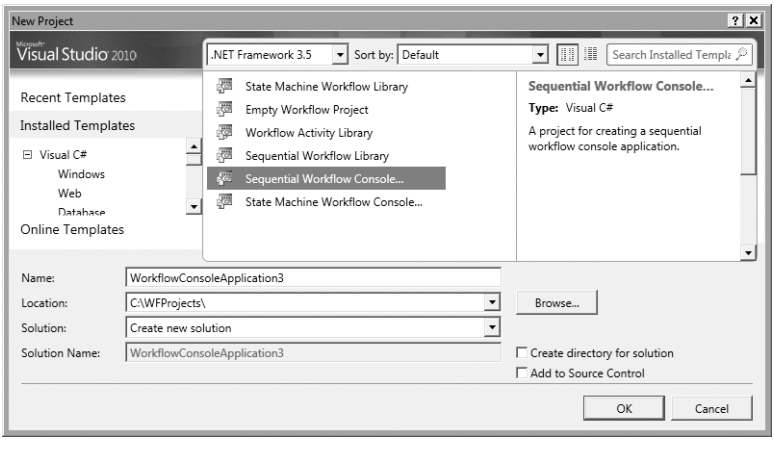

Рис. 57.1. *Новый набор проектов рабочих потоков*

Выберите из доступных шаблонов Sequential Workflow Console Application (это создаст консольное приложение, которое будет служить хостом для исполняющей среды рабочего потока) и рабочий поток по умолчанию, на который потом можно будет перетаскивать действия.

Перетащите действие `Code` из панели инструментов на поверхность конструктора — так, чтобы получить рабочий поток вроде того, что показан на рис. 57.2.

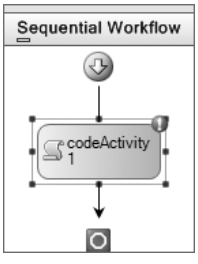

Рис. 57.2. *Рабочий поток с действием* `Code`

Восклицательный знак в верхнем правом углу действия указывает на то, что обязательное свойство действия пока еще не определено. В данном случае это свойство `ExecuteCode`, указывающее метод, который будет вызван при выполнении действия. О пометке свойства как обязательного вы узнаете в разделе, посвященном верификации действия. В результате выполнения двойного щелчка на действии кода будет создан нужный метод в классе отделенного кода, и в нем можно использовать `Console.Writeline` для вывода строки "Hello World", как показано в приведенном ниже фрагменте кода.

```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
   Console.WriteLine("Hello World");
}
```

Если теперь собрать и запустить приложение, на консоли появится вывод этого текста. Когда программа выполняется, создается экземпляр `WorkflowRuntime`, а затем конструируется и выполняется экземпляр рабочего потока. Когда активизируется код действия, он вызывает определенный нами метод, который выводит строку на консоль. В разделе "Исполняющая среда рабочего потока" детально показано, как организовать хост для исполняющей среды. Код рассмотренного примера находится в подкаталоге `01 HelloWorkflowWorld` каталога примеров для настоящей главы.

Действия

Все, что происходит в рабочем потоке — это действия. Сам рабочий поток представляет собой действие, причем действие специфического типа, которое обычно позволяет определять внутри него другие действия; это известно как составное действие (composite activity), и позднее в главе будут даны примеры других составных действий. Действие — это просто класс, который обязательно наследуется от класса `Activity`.

Класс `Activity` определяет ряд перегружаемых методов и, вероятно, наиболее важный из них — метод `Execute`, представленный в следующем фрагменте кода:

```
protected override ActivityExecutionStatus Execute
    ( ActivityExecutionContext executionContext )
{
  return ActivityExecutionStatus.Closed;
}
```

Когда исполняющая среда планирует действие к выполнению, обязательно вызывается метод `Execute`, и именно здесь имеется возможность поместить собственный код, реализующий поведение действия. В примере из предыдущего раздела, когда исполняющая среда рабочего потока вызывает `Execute` из `CodeActivity`, то реализация этого метода запустит метод, определенный в классе отделенного кода, который, в свою очередь, выведет строку на консоль.

Методу `Execute` передается параметр контекста типа `ActivityExecutionContext`, о котором вы узнаете больше на протяжении главы. Этот метод имеет тип возврата типа `ActivityExecutionStatus`, который используется исполняющей средой, чтобы определить, завершилось ли действие успешно, все еще продолжается, либо пребывает в одном из других возможных состояний, которые могут описать исполняющей среде рабочего потока, как обстоят дела с данным действием. Возврат из этого метода `ActivityExecutionStatus.Closed` говорит о том, что действие завершило свою работу и может быть закрыто.

Платформа WF предоставляет множество стандартных действий, и в следующем разделе будут приведены примеры некоторых из них вместе со сценариями, где их можно использовать. Соглашение по именованию для действий требует добавления суффикса `Activity` к имени; например, код действия, показанный на рис. 43.2, определен классом `CodeActivity`.

Все стандартные действия определены в пространстве имен `System.Workflow.Activities`, которое является частью сборки `System.Workflow.Activities.dll`. Есть две другие сборки, которые составляют WF — это `System.Workflow.ComponentModel.dll` и `System.Workflow.Runtime.dll`.

IfElseActivity

Как следует из названия, это действие работает подобно оператору `if-else` в C#.

Когда вы поместите действие `IfElseActivity` на поверхность конструктора, то увидите примерно то, что показано на рис. 57.3. Действие `IfElseActivity` является составным, потому что оно создает две ветви (которые сами по себе представляют действие определенного типа; в данном случае — `IfElseBranchActivity`). Каждая ветвь также является составным действием, унаследованным от `SequenceActivity` — класса, выполняющего последовательность действий одно за другим, сверху вниз. Как видно на рисунке, конструктор добавляет текст Drop Activities Here (Поместите сюда действия), указывая места, куда должны добавляться дочерние действия.

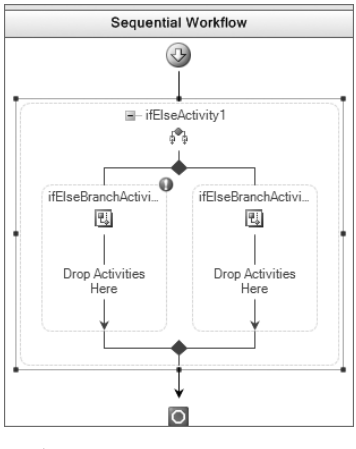

Рис. 57.3. *Действие* `IfElseActivity`

Первая ветвь, показанная на рис. 57.3, содержит восклицательный знак, указывающий на необходимость определения свойства `Condition` (условие). Условие унаследовано от `ActivityCondition` и используется для определения того, какая ветвь должна выполняться.

Когда выполняется действие `IfElseActivity`, оно оценивает условие первой ветви, и если оно истинно, тогда эта ветвь выполняется. Если же условие оценивается как ложное, то `IfElseActivity` проверяет следующую ветвь и т.д. — до тех пор, пока не доберется до последней ветви в действии. Следует отметить, что `IfElseActivity` может иметь любое количество ветвей, каждая со своим собственным условием. Последней ветви условие не нужно, поскольку она работает как часть `else` в операторе `if-else`. Чтобы добавить новую ветвь, необходимо отобразить контекстное меню для действия и выбрать в нем Add Branch (Добавить ветвь) — этот пункт также доступен через меню Workflow (Рабочий поток) внутри Visual Studio. По мере добавления ветвей каждая из них, за исключением последней, будет иметь обязательное условие.

В WF определено два стандартных типа условий — `CodeCondition` и `RuleCoditionReference`. Класс `CodeCondition` выполняет метод класса отделенного кода, который может вернуть `true` или `false`. Чтобы создать `CodeCondition`, отобразите таблицу свойств для `IfElseActivity` и установите условие в Code Condition, после чего введите имя кода, который должен быть выполнен, как показано на рис. 57.4.

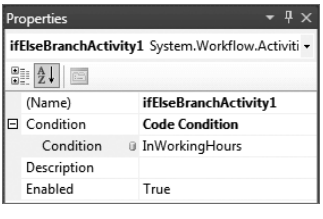

Рис. 57.4. *Установка условия*

После ввода имени метода в таблице свойств визуальный конструктор создаст метод в классе отделенного кода, как показано в следующем фрагменте:

```
private void InWorkingHours(object sender, ConditionalEventArgs e)
{
  int hour = DateTime.Now.Hour;
  e.Result = ((hour >= 9) && (hour <= 17));
}
```

В приведенном выше коде свойство `Result` переданного экземпляра `ConditionalEventArgs` устанавливается в `true`, если текущий час находится в диапазоне с 9:00 до 17:00. Условия могут быть определены в коде, как показано здесь, но есть и другой вариант — когда условие основано на правиле (rule), которое оценивается аналогичным образом. Визуальный конструктор Workflow Designer содержит редактор правил, который может быть использован для объявления условий и операторов (во многом подобно оператору `if-else` выше). Эти правила оцениваются во время выполнения на основе текущего состояния рабочего потока.

ParallelActivity

Это действие позволяет определить набор действий, выполняемых в параллельной или, скорее, в псевдопараллельной манере. Когда исполняющая среда рабочего потока планирует действие, она делает это в единственном потоке. Этот поток выполняет первое действие, затем второе и т.д., пока не будут завершены все действия (или пока действие ожидает некоторого ввода). Когда активизируется `ParallelActivity`, оно проходит по всем ветвям и планирует выполнение каждой из них по очереди. Исполняющая среда ра-

бочего потока поддерживает очередь запланированных к выполнению действий для каждого экземпляра рабочего потока и обычно выполняет их в режиме FIFO (первый пришел, первый обслужен).

Если предположить, что имеется действие `ParallelActivity`, показанное на рис. 57.5, то это запланирует выполнение `sequenceActivity1` и затем `sequenceActivity2`. Тип `SequenceActivity` работает, планируя выполнение первого действия вместе с исполняющей средой, и когда оно закончится, планирует второе действие. Этот метод "планирование/ожидание завершения" используется для прогона всех дочерних действий последовательности до тех пор, пока все они не будут выполнены, и тогда все последовательное действие может завершиться.

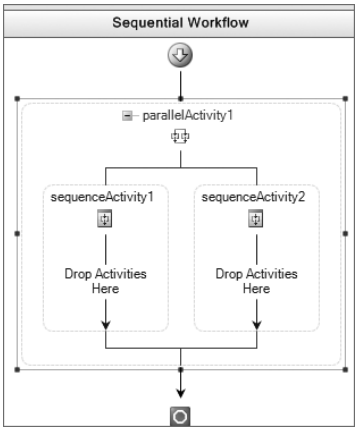

Рис. 57.5. *Действие* `ParallelActivity`

Когда говорится о том, что `SequenceActivity` планирует выполнение по одному действию за раз, подразумевается, что очередь, поддерживаемая `WorkflowRuntime`, постоянно обновляется запланированными действиями.

Предположим, что есть параллельное действие P1, которое содержит два последовательных действия S1 и S2, каждое из которых состоит из действий кода C1 и C2. Это даст очередь запланированных действий, показанную в табл. 57.1.

Таблица 57.1. Очередь запланированных действий

Очередь рабочего потока	Изначально в очереди нет никаких действий
P1	Параллельное действие запускается вместе с рабочим потоком
S1, S2	Добавляется в очередь при выполнении P1
S2, S1.C1	S1 выполняется и добавляет в очередь S1.C1
S1.C1, S2.C1	S2 выполняется и добавляет в очередь S2.C1
S2.C1, S2.C2	S1.C1 завершается, подходит очередь S1.C2
S1.C2, S2.C2	S2.C1 завершается, подходит очередь S2.C2
S2.C2	Последний элемент очереди

Здесь очередь сначала обрабатывает первый элемент (параллельное действие P1), которое добавляет два последовательных действия S1 и S2 в очередь рабочего потока. При выполнении действия S1 оно помещает свое первое действие (S1.C1) в конец

очереди, и когда это действие будет запланировано и выполнено, помещает в конец очереди свое второе действие.

Как видно в приведенном примере, выполнение `ParallelActivity` происходит не совсем параллельно; на самом деле оно поочередно выполняет компоненты двух последовательных ветвей. Отсюда можно сделать вывод, что лучше, когда отдельное действие требует минимального отрезка времени, поскольку с учетом того, что очередь каждого рабочего потока обслуживает только один поток, долго выполняющееся действие может затормозить выполнение прочих действий в очереди. Таким образом, часто возникает ситуация, когда действие может выполняться в течение произвольного отрезка времени, поэтому должен быть способ пометить действие как "долгоиграющее", чтобы другие действия получили возможность выполняться. Это можно сделать, возвратив `ActivityExecutionStatus` из метода `Execute`, которое позволяет исполняющей среде узнать о том, что вы обратитесь к ней позднее, когда данное действие завершится. Примером такого действия может служить `DelayActivity`.

CallExternalMethodActivity

Рабочий поток обычно нуждается в вызове методов извне рабочего потока, и такое действие позволяет определить интерфейс и метод для вызова в этом интерфейсе. `WorkflowRuntime` поддерживает список служб (помеченных ключом — значением `System.Type`), которые могут быть доступны через параметр `ActivityExecutionContext`, переданный методу `Execute`.

Можно определить собственные службы для добавления в коллекцию и затем обращаться к ним изнутри собственных действий. Например, можно построить уровень доступа к данным, представленный как интерфейс службы, и затем подготовить различные реализации этой службы для SQL Server и Oracle. Поскольку действия просто вызывают методы интерфейса, замена SQL Server на Oracle будет совершенно прозрачной для действий, поскольку они просто вызывают методы интерфейса и ничего не знают о конкретной системе управления базами данных.

После добавления к рабочему потоку `CallExternalMethodActivity` определяются два обязательных свойства — `InterfaceType` и `MethodName`. Тип интерфейса определяет, какая служба времени выполнения будет использована при выполнении действия, а имя метода указывает, какой метод этого интерфейса будет вызван.

Когда действие выполняется, оно ищет службу с определенным интерфейсом, запрашивая исполняющий контекст на предмет типа службы, и затем вызывает соответствующий метод этого интерфейса. Методу внутри рабочего потока можно также передать параметры; об этом речь пойдет в разделе "Привязка параметров к действиям" далее в главе.

DelayActivity

Бизнес-процессам часто бывает нужно подождать некоторое время перед тем, как завершиться (представьте себе использование рабочего потока для утверждения расходов). Рабочий поток может отправить по электронной почте письмо непосредственному начальнику, запрашивая у него утверждение расходов. Рабочий поток затем переходит в режим ожидания, в котором он либо ждет утверждения (или отказа), но было бы неплохо определить таймаут, чтобы если ответ не получен, скажем, в течение одного дня, отчет о расходах был бы автоматически перенаправлен более следующему начальнику в административной иерархии.

Действие `DelayActivity` может формировать часть этого сценария (другую часть реализует `ListenActivity`, о котором речь пойдет ниже), и его работа заключается в ожидании в течение определенного времени, прежде чем продолжить выполнение рабочего потока. Существует также два пути определения времени ожидания: можно либо установить

в свойстве TimeoutDuration строку вроде "1.00:00:00" (1 день, ноль часов, ноль минут и ноль секунд), либо предоставить метод, который будет вызван при выполнении действия, который установит продолжительность ожидания в своем коде. Чтобы сделать это, потребуется определить значение для свойства InitializeTimeoutDuration действия задержки, что сгенерирует метод в отделенном коде, как показано в следующем фрагменте:

```
private void DefineTimeout(object sender, EventArgs e)
{
  DelayActivity delay = sender as DelayActivity;
  if (null != delay)
  {
    delay.TimeoutDuration = new TimeSpan(1, 0, 0, 0);
  }
}
```

Здесь в методе DefineTimeout осуществляется приведение sender к типу DelayActivity и установка его свойства TimoutDuration в TimeSpan. Несмотря на то что в данном примере значение жестко закодировано, более вероятно, что вы будете строить его на основе некоторых других данных — возможно, параметра, переданного в рабочий поток, или значения, прочитанного из конфигурационного файла. Параметры рабочего потока обсуждаются в разделе "Рабочие потоки" далее в главе.

ListenActivity

Распространенная программная конструкция предусматривает организацию ожидания одного из набора возможных событий; примером может служить метод WaitAny класса System.Threading.WaitHandle. Действие ListenActivity предоставляет возможность сделать это в рабочем потоке, поскольку может определить любое количество ветвей, каждая из которых основана на некотором событии.

Действие события — это такое действие, которое реализует интерфейс IEventActivity, определенный в пространстве имен System.Workflow.Activities. В настоящее время есть три таких действия, определенных как стандартные в WF, а именно: DelayActivity, HandleExternalEventActivity и WebServiceInputActivity. Ниже на рис. 57.6 показан рабочий поток, ожидающий либо внешнего ввода, либо истечения некоторого времени — это пример, подходящий для реализации утверждения расходов, о котором упоминалось ранее.

В этом примере действие CallExternalMethodActivity используется в качестве первого действия в рабочем потоке, которое вызывает метод, определенный в интерфейсе службы, который запросит у руководителя утверждения расходов. Поскольку это внешняя служба, запрос может поступить в форме сообщения электронной почты, мгновенного сообщения (instance message — IM) либо в любом другом виде, позволяющем известить руководителя о том, что ему нужно обработать запрос на расходы. Затем рабочий поток выполняет действие ListenActivity, которое ожидает ввода от этой внешней службы (утверждения или отказа), и ожидает его в течение заданного периода времени.

Действие ListenActivity ставит в очередь ожидание как первое действие в каждой ветви, и когда произойдет ожидаемое событие, все прочие ожидания прерываются, и осуществляется обработка остальной части той ветви, где произошло событие. Таким образом, в экземпляре, где отчет о расходах был утвержден, возникает событие Approved и планируется действие PayMe. Если же, однако, руководитель отклоняет запрос на расходы, генерируется событие Rejected, и в данном примере происходит переход к действию Panic.

И последнее: если не возникло ни одно из событий — ни Approved, ни Rejected, — тогда по истечении заданного времени ожидания действие DelayActivity обязательно завершается, и отчет о расходах может быть направлен другому руководителю, возможно, после выполнения его поиска вверх по иерархии в Active Directory.

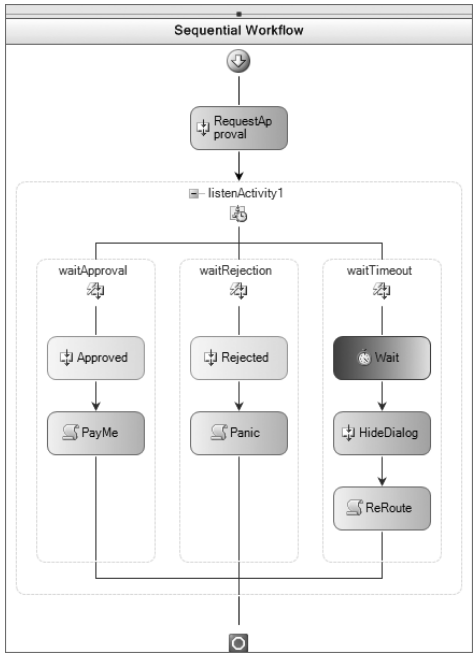

Рис. 57.6. *Действие* `ListenActivity`

В данном примере пользователю отображается диалоговое окно, когда выполняется действие `RequestApproval`, так что в случае, когда производится ожидание, также нужно закрыть это диалоговое окно, т.е. выполнить действие по имени `HideDialog` (см. рис. 57.6). В данном примере использованы некоторые концепции, которые пока еще не были раскрыты, например, как идентифицируется экземпляр рабочего потока, и как события попадают обратно в исполняющую среду рабочего потока, и в конечном итоге доставляются правильному экземпляру рабочего потока. Эти концепции рассматриваются в разделе “Рабочие потоки”.

Модель выполнения действий

До сих пор в этой главе обсуждалось только выполнение действий исполняющей средой через вызов метода `Execute`. На самом деле действие в процессе своего выполнения может проходить через ряд состояний, что отражено на рис. 57.7.

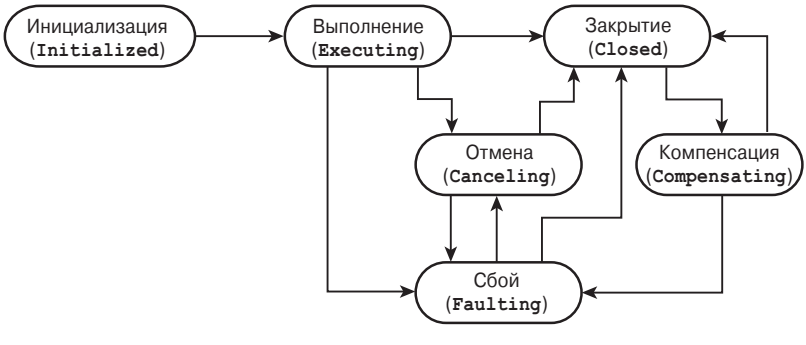

Рис. 57.7. *Состояния действия во время его выполнения*

Сначала действие инициализируется посредством WorkflowRuntime, когда исполняющая среда вызывает метод Initialize этого действия. Этому методу передается экземпляр IServiceProvider, который отображает доступные службы внутри исполняющей среды. Эти службы рассматриваются в разделе "Службы рабочих потоков" далее в главе. Большинство действий в этом методе ничего не делают, но метод осуществляет необходимые начальные установки.

Затем исполняющая среда вызывает метод Execute, и действие может вернуть любое из значений перечисления ActivityExecutionStatus. Обычно из метода будете возвращаться значение Closed, которое говорит о том, что действие завершило свою обработку. Однако если вернуть одно из других значений состояния, то исполняющая среда использует его для определения того, в каком состоянии находится действие.

Возврат из этого метода значения Executing сообщает исполняющей среде о том, что еще имеется работа, которую нужно доделать — типичным примером может быть составное действие, которое должно выполнить свои дочерние элементы. В этом случае действие может запланировать запуск каждого своего дочернего действия и ожидать завершения их работы прежде, чем известить исполняющую среду о том, что данное действие завершилось.

Пользовательские действия

До сих пор вы имели дело с действиями, определенными внутри пространства имен System.Workflow.Activities. В этом разделе будет показано, как создавать собственные действия и расширять их для удовлетворения пользовательских потребностей — как во время проектирования, так и во время выполнения.

Для начала создается действие WriteLineActivity, которое можно будет использовать для вывода строки текста на консоль. Хотя этот пример достаточно тривиален, позднее он будет расширен, чтобы продемонстрировать полный набор возможностей, доступных пользовательским действиям. При создании пользовательских действий можно просто сконструировать класс внутри проекта рабочего потока, однако предпочтительнее создавать собственные действия внутри отдельной сборки, поскольку среда времени проектирования Visual Studio (и особенно проекты рабочих потоков) загрузит действия из ваших сборок и сможет заблокировать сборку, которую вы попытаетесь обновить. По этой причине для конструирования пользовательских действий понадобится создать проект простой библиотеки классов.

Простое действие вроде WriteLineActivity будет порождено непосредственно от базового класса Activity. В следующем коде показан сконструированный класс действия и определено свойство Message, отображаемое при вызове метода Execute.

```
using System;
using System.ComponentModel;
using System.Workflow.ComponentModel;
namespace SimpleActivity
{
 /// <summary>
 /// Простое действие, которое во время выполнения отображает сообщение на консоли
 /// </summary>
 public class WriteLineActivity : Activity
 {
  /// <summary>
  /// Выполнение действия — отображение сообщения на экране
  /// </summary>
  /// <param name="executionContext"></param>
  /// <returns></returns>
  protected override ActivityExecutionStatus Execute
  (ActivityExecutionContext executionContext)
  {
```

```
        Console.WriteLine(Message);
        return ActivityExecutionStatus.Closed;
    }
    /// <summary>
    /// Методы get/set отображаемого сообщения
    /// </summary>
    [Description("The message to display")]
    [Category("Parameters")]
    public string Message
    {
        get { return _message; }
        set { _message = value; }
    }
    /// <summary>
    /// Сохранение отображаемого сообщения
    /// </summary>
    private string _message;
  }
}
```

Внутри метода `Execute` можно вывести сообщение на консоль и затем возвратить состояние `Closed`, уведомив исполняющую среду о том, что действие завершено.

Также можно определить атрибуты на свойстве `Message`, задав для него описание и категорию; эти свойства будут присутствовать в таблице свойств внутри Visual Studio, как показано на рис. 57.8.

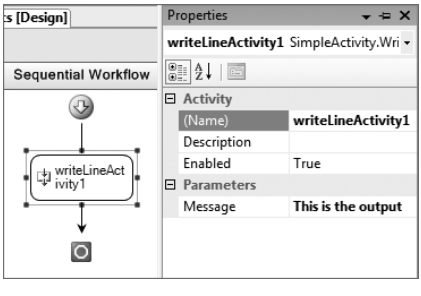

Рис. 57.8. *Атрибуты свойства* `Message`

Код для действий, созданных в этой разделе, доступен в решении `03 CustomActivities`. После компиляции этого решения можно добавить собственные действия в панель инструментов Visual Studio, выбрав пункт Choose Items (Выберите элементы) в контекстном меню панели инструментов и перейдя в папку, где находится сборка, содержащая действия. Все действия, содержащиеся в сборке, будут добавлены в панель инструментов.

В таком виде действие вполне удобно; однако есть несколько способов сделать его еще более дружественным к пользователю. Как вы видели на примере `CodeActivity` ранее в этой главе, существует несколько обязательных свойств, которые, не будучи определенными, приведут к появлению знака ошибки на поверхности визуального конструктора. Чтобы получить тот же эффект для вашего действия, нужно построить класс, унаследованный от `ActivityValidator`, и ассоциировать этот класс с действием.

Верификация действий

Когда действие помещено на поверхность визуального конструктора, Workflow Designer ищет в этом действии атрибуты, которые определяют класс, осуществляющий верификацию данного действия. Для верификации создаваемого действия необходимо проверить, установлено ли свойство `Message`.

Пользовательский верификатор передается экземпляру действия, и здесь можно определить, какие свойства являются обязательными (если таковые есть), но не были еще определены, а также добавить ошибку в `ValidationErrorCollection`, используемую конструктором. Эту коллекцию затем читает Workflow Designer, и любые найденные в коллекции ошибки вызовут добавление к действию предупреждающего знака, и необязательно свяжут каждую ошибку со свойством, требующим внимания.

```
using System;
using System.Workflow.ComponentModel.Compiler;
namespace SimpleActivity
{
 public class WriteLineValidator : ActivityValidator
 {
  public override ValidationErrorCollection Validate
     (ValidationManager manager, object obj)
  {
  if (null == manager)
     throw new ArgumentNullException("manager");
  if (null == obj)
     throw new ArgumentNullException("obj");
  ValidationErrorCollection errors = base.Validate(manager, obj);
  // Привести к WriteLineActivity
  WriteLineActivity act = obj as WriteLineActivity;
  if (null != act)
  {
     if (null != act.Parent)
     {
        // Проверить свойство Message
        if (string.IsNullOrEmpty(act.Message))
          errors.Add(ValidationError.GetNotSetValidationError("Message"));
        }
     }
     return errors;
  }
 }
}
```

Метод `Validate` вызывается конструктором, когда обновляется любая часть действия, а также когда действие помещается на поверхность конструктора. Визуальный конструктор вызывает метод `Validate` и передает действие в виде нетипизированного параметра `obj`.

В этом методе сначала проверяются переданные ему аргументы, и затем вызывается метод `Validate` базового класса, чтобы получить `ValidationErrorCollection`. Хотя это здесь и не обязательно, но если имеет место наследование от действия с множеством свойств, которые также нуждаются в верификации, то вызов метода базового класса гарантирует, что все они также будут проверены.

Приведите переданный параметр к экземпляру `WriteLineActivity` и проверьте, есть ли у действия родитель. Эта проверка необходима, потому что функция `Validate` вызывается во время компиляции действия (если это действие внутри проекта рабочего потока или библиотеки действий), а в этой точке никакого родительского действия не определено. Без этой проверки вы не сможете в действительности собрать сборку, содержащую действие и его верификатор. Этот дополнительный шаг не нужен, если типом проекта является библиотека классов.

Последний шаг состоит в проверке установки для свойства `Message` значения, отличного от пустой строки; этим занимается статический метод класса `ValidationError`, который конструирует ошибку, указывающую на то, что свойство не было определено.

Для поддержки действия `WriteLineActivity` необходимо добавить к действию атрибут `ActivityValidation`, как показано в следующем фрагменте кода:

```
[ActivityValidator(typeof(WriteLineValidator))]
public class WriteLineActivity : Activity
{
    ...
}
```

Если вы скомпилируете приложение и затем поместите `WriteLineActivity` в рабочий поток, то увидите ошибку верификации, как показано на рис. 57.9; щелчок на символе ошибки приводит к переходу на соответствующее свойство в таблице свойств.

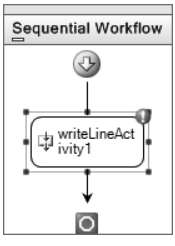

Рис. 57.9. *Ошибка верификации*

После ввода некоторого текста в свойстве `Message` ошибка верификации исчезнет, и можно будет скомпилировать и запустить приложение.

Теперь, при наличии готовой верификации действия, следующее, что потребуется изменить — это поведение визуализации действия, добавив цвет его заливки. Для этого нужно определить два класса — `ActivityDesigner` и `ActivityDesignerTheme`, как описано в следующем разделе.

Темы и конструкторы

Экранное отображение действия осуществляется с помощью класса `ActivityDesigner`, при этом также может использоваться класс `ActivityDesignerTheme`.

Класс темы служит для простого изменения поведения отображения действия внутри Workflow Designer.

```
public class WriteLineTheme : ActivityDesignerTheme
{
 /// <summary>
 /// Конструирование темы и установка ряда значений по умолчанию
 /// </summary>
 /// <param name="theme"></param>
 public WriteLineTheme(WorkflowTheme theme)
  : base(theme)
 {
   this.BackColorStart = Color.Yellow;
   this.BackColorEnd = Color.Orange;
   this.BackgroundStyle = LinearGradientMode.ForwardDiagonal;
 }
}
```

Тема унаследована от класса `ActivityDesignerTheme`, который имеет конструктор, принимающий аргумент `WorkflowTheme`. Внутри конструктора устанавливаются начальный и конечный цвета действия, а также определяется кисть линейного градиента, используемая для окраски фона.

Класс `Designer` служит для переопределения поведения визуализации действия — в данном случае никакого переопределения не требуется, так что следующего кода вполне достаточно:

```
[ActivityDesignerTheme(typeof(WriteLineTheme))]
public class WriteLineDesigner : ActivityDesigner
{
}
```

Обратите внимание, что тема ассоциирована с конструктором с помощью атрибута `ActivityDesignerTheme`.

Последний шаг состоит в том, чтобы снабдить действие атрибутом `Designer`:

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
public class WriteLineActivity : Activity
{
    ...
}
```

Когда все это будет готово, действие станет отображаться в конструкторе, как показано на рис. 57.10.

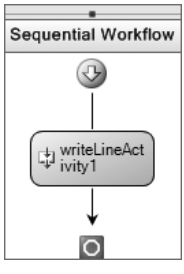

***Рис. 57.10**. Добавление конструктора и темы*

Благодаря такому добавлению конструктора и темы, действие теперь выглядит гораздо более профессионально. В теме доступно и множество других свойств, таких как перо, используемое для рисования рамки, цвет рамки и ее стиль.

Переопределяя метод `OnPaint` класса `ActivityDesigner`, вы можете получить полный контроль над отображением действия. Вам предоставляется возможность самостоятельно поупражняться в этом; создайте действие, совершенно непохожее на любые другие действия в панели инструментов.

Еще одним полезным свойством класса `ActivityDesigner`, которое можно переопределить, является `Verbs`. Оно позволяет добавить пункты в контекстное меню действия и используется конструктором `ParallelActivity` для вставки пункта Add Branch (Добавить ветвь) в контекстное меню действия, а также в меню Workflow (Рабочий поток). Также можно изменить список свойств, предоставляемых действием, переопределив метод конструктора `PreFilterProperties`, посредством которого параметры метода для действия `CallExternalMethodActivity` размещаются в таблице свойств. Если нужно внести расширение подобного рода в визуальный конструктор, понадобится запустить инструмент Lutz Roeder's Reflector (доступен по адресу `http://reflector.red-gate.com`) и загрузить сборки рабочего потока, чтобы увидеть, как в Microsoft определяют некоторые из этих расширенных свойств.

Теперь создаваемое действие почти готово, однако нужно еще определить значок, используемый при отображении свойства, а также элемент панели инструментов для ассоциирования с действием.

ActivityToolboxItem и значки

Чтобы завершить разработку пользовательского действия, нужно добавить значок и необязательно класс, унаследованный от ActivityToolboxItem, который используется при отображении действия в панели инструментов Visual Studio.

Чтобы определить значок для действия, создайте изображение размером 16×16 пикселей и включите его в проект, после чего установите действие сборки для этого изображения в Embedded Resource. Это позволит включить изображение в манифест ресурсов сборки. Можно добавить в проект папку по имени Resources, как показано на рис. 57.11.

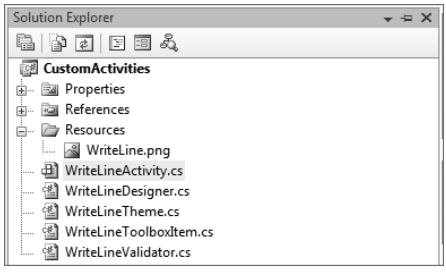

***Рис. 57.11**. Добавление в проект папки Resources*

Добавив файл изображения и установив его действие сборки в Embedded Resource, можно снабдить свое действие еще одним атрибутом, показанным в следующем фрагменте кода:

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
[ToolboxBitmap(typeof(WriteLineActivity),"Resources.WriteLine.png")]
public class WriteLineActivity : Activity
{
   ...
}
```

Атрибут ToolboxBitmap имеет ряд конструкторов, и один из них, использованный здесь, принимает тип, определенный в сборке действия, а также имя ресурса. Когда вы добавляете ресурс в папку, его имя формируется из названия пространства имен сборки и имени папки, в которой содержится изображение, так что полностью определенное имя ресурса будет выглядеть как CustomActivities.Resources.WriteLine.png. Конструктор, используемый для атрибута ToolboxBitmap, добавляет пространство имен, в котором находится тип-параметр, к строке, передаваемой в качестве второго аргумента, так что это все преобразуется в соответствующий ресурс при загрузке Visual Studio.

Последний класс, который потребуется создать, унаследован от ActivityToolboxItem. Этот класс применяется при загрузке действия в панель инструментов Visual Studio. Типичное использование этого класса состоит в изменении отображаемого имени действия в панели инструментов — все встроенные действия имеют измененные имена, в которых из типа исключено слово “Activity”. С разрабатываемым классом можно сделать то же самое, установив свойство DisplayName в "WriteLine".

```
[Serializable]
public class WriteLineToolboxItem : ActivityToolboxItem
{
/// <summary>
/// Установить отображаемое имя WriteLine, т.е. удалить из него строку 'Activity'
/// </summary>
/// <param name="t"></param>
```

```
public WriteLineToolboxItem(Type t)
    : base(t)
{
  base.DisplayName = "WriteLine";
}
/// <summary>
/// Необходимо для среды времени проектирования Visual Studio
/// </summary>
/// <param name="info"></param>
/// <param name="context"></param>
private WriteLineToolboxItem(SerializationInfo info, StreamingContext context)
{
  this.Deserialize(info, context);
}
}
```

Класс унаследован от `ActivityToolboxItem` и переопределяет конструктор, чтобы изменить отображаемое имя; кроме того, он предоставляет конструктор для сериализации, используемый панелью инструментов при загрузке элемента в эту панель. Без конструктора вы получите ошибку при попытке добавить действие в панель инструментов. Обратите также внимание, что класс помечен как `[Serializable]`.

Элемент панели инструментов добавляется к действию за счет применения атрибута `ToolboxItem`, как показано ниже:

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
[ToolboxBitmap(typeof(WriteLineActivity),"Resources.WriteLine.png")]
[ToolboxItem(typeof(WriteLineToolboxItem))]
public class WriteLineActivity : Activity
{
   ...
}
```

После внесения всех этих изменений можно скомпилировать сборку и затем создать новый проект рабочего потока. Чтобы добавить действие к панели инструментов, откройте рабочий поток и отобразите его контекстное меню, в котором выберите пункт Choose Items (Выбрать элементы).

Затем можете перейти к сборке, содержащей ваше действие, и после добавления его в панель инструментов последняя будет выглядеть примерно так, как показано на рис. 57.12. Как видите, значок несколько меньше, чем нужно, но достаточно близок к необходимому.

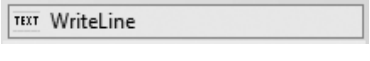

***Рис. 57.12**. Добавленный значок*

В следующем разделе мы еще вернемся к `ActivityToolboxItem`, когда будем говорить о пользовательских составных действиях, поскольку в этом классе есть еще несколько дополнительных средств, которые нужны только при добавлении составных действий на поверхность визуального конструктора.

Чтобы подытожить: вы создали специальное действие `WriteLineActivity`, добавили логику верификации, создав `WriteLineValidator`, создали визуальный конструктор и тему — в виде классов `WriteLineDesigner` и `WriteLineTheme`, создали изображение для действия и, наконец, построили класс `WriteLineToolboxItem` для изменения отображаемого имени действия.

Пользовательские составные действия

Существуют два основных типа действий, унаследованные от `Activity`, которые можно трактовать как функции, вызываемые из рабочего потока. Действия же, унаследованные от `CompositeActivity` (такие как `ParallelActivity`, `IfElseActivity` и `ListenActivity`), являются контейнерами для других действий, и их поведение времени проектирования существенно отличается от поведения простых действий в том смысле, что они представляют область в конструкторе, куда можно помещать их дочерние действия.

В настоящем разделе будет создано действие под названием `DaysOfWeekActivity`. Это действие может быть использовано для выполнения разных частей рабочего потока на основе текущей даты. Например, пусть требуется инициировать выполнение рабочего потока по другому маршруту, обрабатывая заказы, которые поступили в выходные, в отличие от тех, что поступили в течение рабочей недели. Разбирая данный пример, вы ознакомитесь с рядом усовершенствованных свойств рабочих потоков, и к концу раздела будете обладать хорошим пониманием того, как можно расширить систему за счет собственных составных действий.

Для начала создадим пользовательское действие, имеющее свойство, по умолчанию установленное в текущую дату/время, и позволим устанавливать это свойство в другие значения, которые могут поступать от других действий рабочего потока либо в виде параметра, переданного рабочему потоку при его выполнении. Это составное действие будет содержать несколько ветвей — все они определяются пользователем. Каждая из этих ветвей будет содержать перечислимую константу, определяющую, в какой день (или дни) ветвь должна выполняться. Ниже определяется действие и две ветви:

```
DaysOfWeekActivity
SequenceActivty: Monday, Tuesday, Wednesday, Thursday, Friday
  <соответствующие действия>
SequenceActivity: Saturday, Sunday
  <соответствующие действия>
```

Для данного примера понадобится перечисление, определяющее дни недели, и оно должно включать атрибут `[Flags]` (использовать встроенное перечисление `DayOfWeek`, определенное в пространстве имен `System`, нельзя, поскольку оно не имеет атрибута `[Flags]`).

```
[Flags]
[Editor(typeof(FlagsEnumEditor), typeof(UITypeEditor))]
public enum WeekdayEnum : byte
{
  None = 0x00,
  Sunday = 0x01,
  Monday = 0x02,
  Tuesday = 0x04,
  Wednesday = 0x08,
  Thursday = 0x10,
  Friday = 0x20,
  Saturday = 0x40
}
```

Также включен специальный редактор для этого типа, который позволит выбирать перечислимые значения с помощью флажков.

Определив тип перечисления, в качестве начальной заглушки можно назначить само действие. Пользовательские составные действия обычно наследуются от класса `CompositeActivity`, который помимо прочего определяет свойство `Activities`, представляющее собой коллекцию всех составляющих его действий.

```
public class DaysOfWeekActivity : CompositeActivity
{
 /// <summary>
 /// Методы get/set свойства дня недели
 /// </summary>
 [Browsable(true)]
 [Category("Behavior")]
 [Description("Привязано к свойству DateTime, укажите дату и время
              или оставьте пустым для DateTime.Now")]
 [DefaultValue(typeof(DateTime),"")]
 public DateTime Date
 {
   get { return (DateTime)base.GetValue(DaysOfWeekActivity.DateProperty); }
   set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
 }
 /// <summary>
 /// Регистрация свойства DayOfWeek
 /// </summary>
 public static DependencyProperty DateProperty =
     DependencyProperty.Register("Date", typeof(DateTime),
       typeof(DaysOfWeekActivity));
}
```

Свойство `Date` предоставляет обычные средства доступа `get` и `set`, кроме того, добавлен ряд стандартных атрибутов, чтобы они корректно отображались в панели свойств. Поэтому код выглядит несколько иначе, чем у обычного свойства .NET, поскольку эти средства `get` и `set` не используют стандартное поле для хранения значений, а вместо этого применяется то, что носит название `DependencyProperty` (свойство зависимости).

Класс `Activity` (а, следовательно, и данный класс, поскольку он обязательно порожден от `Activity`) унаследован от класса `DependencyObject`, и это определяет словарь ключевых значений `DependencyProperty`. Это косвенное получение и установка значений свойства применяется WF для поддержки привязки, т.е. связывания свойства одного действия со свойством другого. В качестве примера принято передавать параметры в коде иногда по значению, иногда по ссылке. В WF используется привязка для связывания вместе свойств, поэтому в настоящем примере можно иметь свойство `DateTime`, определенное в рабочем потоке, а действие может быть привязано к значению во время выполнения. Позднее в этой главе будет показан пример такой привязки.

Если вы соберете это действие, оно будет делать не так много — на самом деле оно даже не позволит помещать в него дочерние действия, поскольку для этого действия не определен класс `Designer`.

Добавление класса `Designer`

Как было показано в примере с `WriteLineActivity` ранее в этой главе, каждое действие может иметь ассоциированный с ним класс `Designer`, используемый для изменения поведения действия во время проектирования. В действии `WriteLineActivity` класс `Designer` был пустой, но для составного действия придется переопределить несколько методов, чтобы добавить возможность обработки специальных случаев.

```
public class DaysOfWeekDesigner : ParallelActivityDesigner
{
 public override bool CanInsertActivities
     (HitTestInfo insertLocation, ReadOnlyCollection<Activity> activities)
 {
  foreach (Activity act in activities)
  {
     if (!(act is SequenceActivity))
       return false;
  }
```

```
    return base.CanInsertActivities(insertLocation, activitiesToInsert);
  }
  protected override CompositeActivity OnCreateNewBranch()
  {
    return new SequenceActivity();
  }
}
```

Этот Designer унаследован от ParallalActivityDesigner, который предоставляет в ваше распоряжение поведение времени проектирования при добавлении дочерних действий. Вам придется переопределить CanInsertActivities для возврата false, когда любое из добавленных действий не будет являться SequenceActivity. Если все добавляемые действия будут относиться к соответствующему типу, можно будет вызвать метод базового класса, который выполнит некоторые дополнительные проверки над типами действий, допустимыми внутри пользовательского составного действия.

Также потребуется переопределить метод OnCreateNewBranch, вызываемый, когда пользователь выбирает пункт меню Add Branch. Класс Designer ассоциируется с действием за счет применения атрибута [Designer], как показано ниже:

```
[Designer(typeof(DaysOfWeekDesigner))]
public class DaysOfWeekActivity : CompositeActivity
{
}
```

Поведение времени проектирования почти готово; однако в это действие также нужно добавить класс, унаследованный от ActivityToolboxItem, который определит то, что случится, когда экземпляр этого действия будет перетаскиваться из панели инструментов. Поведение по умолчанию состоит просто в конструировании нового действия; тем не менее, в рассматриваемом примере понадобится сразу создать две ветви по умолчанию. Ниже приведен код класса элемента панели инструментов, который сделает это.

```
[Serializable]
public class DaysOfWeekToolboxItem : ActivityToolboxItem
{
  public DaysOfWeekToolboxItem(Type t)
      : base(t)
  {
    this.DisplayName = "DaysOfWeek";
  }
  private DaysOfWeekToolboxItem(SerializationInfo info, StreamingContext context)
  {
    this.Deserialize(info, context);
  }
  protected override IComponent[] CreateComponentsCore(IDesignerHost host)
  {
    CompositeActivity parent = new DaysOfWeekActivity();
    parent.Activities.Add(new SequenceActivity());
    parent.Activities.Add(new SequenceActivity());
    return new IComponent[] { parent };
  }
}
```

В этом коде видно, что отображаемое имя действия изменено, реализован конструктор сериализации и переопределен метод CreateComponentsCore.

Этот метод вызывается в конце операции перетаскивания, и именно здесь конструируется экземпляр DaysOfWeekActivity. В коде также строятся два дочерних последовательных действия, поскольку это сразу облегчит работу пользователю во время проектирования. То же самое осуществляют некоторые встроенные действия. Так, например, когда на поверхность визуального конструктора помещается действие IfElseActivity, его класс

панели инструментов также добавляет две ветви. Подобное происходит и при добавлении в рабочий поток действия `ParallelActivity`.

Конструктор сериализации и атрибут `[Serializable]` необходимы всем классам, унаследованным от `ActivityToolboxItem`.

Последнее, что потребуется сделать — ассоциировать этот класс элемента панели инструментов с действием:

```
[Designer(typeof(DaysOfWeekDesigner))]
[ToolboxItem(typeof(DaysOfWeekToolboxItem))]
public class DaysOfWeekActivity : CompositeActivity
{
}
```

Теперь пользовательский интерфейс действия почти завершен, как можно видеть на рис. 57.13.

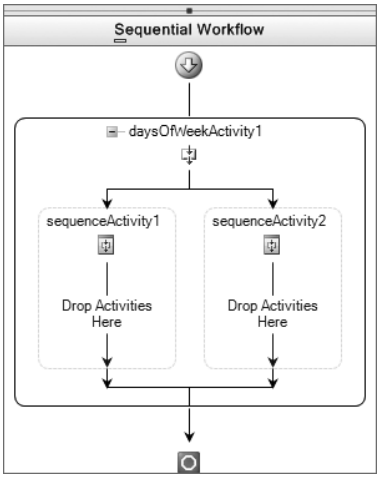

***Рис. 57.13**. Добавление `Designer` к пользовательскому действию*

Теперь нужно определить свойства каждого из последовательных действий, показанных на рис. 57.13, чтобы пользователь смог задать, в какие дни какая из них должна выполняться. Есть два способа сделать это в Windows Workflow: можно определить подкласс `SequenceActivity` и задать все это в нем или же можно воспользоваться другим средством свойств зависимости, называемым Attached Properties (присоединенные свойства).

Давайте применим второй метод, что исключит необходимость в подклассе, но вместо этого позволит эффективно расширить последовательное действие без потребности в его исходном коде.

Присоединенные свойства

При регистрации свойств зависимости можно вызвать метод `RegisterAttached` для создания присоединенного свойства. Присоединенное свойство — это такое свойство, которое определено в одном классе, но отображается в другом. В данном случае свойство определяется в `DaysOfWeekActivity`, но на самом деле это свойство отображается в пользовательском интерфейсе как присоединенное к последовательному действию.

```
public static DependencyProperty WeekdayProperty =
DependencyProperty.RegisterAttached("Weekday",
     typeof(WeekdayEnum), typeof(DaysOfWeekActivity),
     new PropertyMetadata(DependencyPropertyOptions.Metadata));
```

Последняя строка позволяет указать дополнительную информацию о свойстве; в данном примере специфицировано, что этим свойством будет `Metadata`.

Свойства `Metadata` отличаются от обычных свойств в том отношении, что они эффективно читаются только во время выполнения. Свойство `Metadata` можно воспринимать как нечто подобное объявлению констант внутри C#. Вы не можете изменять константы во время выполнения программы и точно также не можете изменять свойства `Metadata` во время выполнения рабочего потока.

В данном примере требуется определить дни активизации действия, поэтому можно установить в `Designer` это поле в "Saturday, Sunday". В коде, сгенерированном для рабочего потока, можно видеть примерно такие объявления:

```
this.sequenceActivity1.SetValue
(DaysOfWeekActivity.WeekdayProperty,
((WeekdayEnum)((WeekdayEnum.Sunday | WeekdayEnum.Saturday))));
```

В дополнение к определению свойства зависимости понадобятся методы, позволяющие получать и устанавливать это значение для произвольного действия. Оно обычно определяется как статический метод составного действия и показано в следующем коде:

```
public static void SetWeekday(Activity activity, object value)
{
  if (null == activity)
    throw new ArgumentNullException("activity");
  if (null == value)
    throw new ArgumentNullException("value");
  activity.SetValue(DaysOfWeekActivity.WeekdayProperty, value);
}
public static object GetWeekday(Activity activity)
{
  if (null == activity)
    throw new ArgumentNullException("activity");
     return activity.GetValue(DaysOfWeekActivity.WeekdayProperty);
}
```

Есть еще два изменения, которые следует внести, чтобы дополнительное свойство могло отображаться как присоединенное к `SequenceActivity`. Первое — нужно создать поставщик расширителей, который сообщит Visual Studio о необходимости включения дополнительного свойства в последовательное действие, и второе — зарегистрировать этого поставщика, что делается переопределением метода `Initialize` в Activity Designer и добавлением к нему следующего кода:

```
protected override void Initialize(Activity activity)
{
 base.Initialize(activity);
 IExtenderListService iels = base.GetService(typeof(IExtenderListService))
     as IExtenderListService;
 if (null != iels)
 {
  bool extenderExists = false;
  foreach (IExtenderProvider provider in iels.GetExtenderProviders())
  {
     if (provider.GetType() == typeof(WeekdayExtenderProvider))
     {
        extenderExists = true;
        break;
     }
  }
  if (!extenderExists)
  {
     IExtenderProviderService ieps =
```

```
        base.GetService(typeof(IExtenderProviderService))
          as IExtenderProviderService;
      if (null != ieps)
        ieps.AddExtenderProvider(new WeekdayExtenderProvider());
   }
  }
}
```

Вызовы `GetService` в приведенном коде предназначены для того, чтобы позволить пользовательскому конструктору запросить службы, предоставляемые хостом (в данном случае им является Visual Studio). Вы запрашиваете у Visual Studio `IExtenderListService`, который предоставляет способ перечислить всех доступных поставщиков расширителей, и если не будет найдено ни одного экземпляра `WeekdayExtenderProvider`, тогда запрашивается `IExtenderProviderService` и добавляется новый поставщик.

Ниже показан код поставщика расширителей.

```
[ProvideProperty("Weekday", typeof(SequenceActivity))]
public class WeekdayExtenderProvider : IExtenderProvider
{
 bool IExtenderProvider.CanExtend(object extendee)
 {
  bool canExtend = false;
  if ((this != extendee) && (extendee is SequenceActivity))
  {
     Activity parent = ((Activity)extendee).Parent;
     if (null != parent)
       canExtend = parent is DaysOfWeekActivity;
  }
  return canExtend;
 }
 public WeekdayEnum GetWeekday(Activity activity)
 {
  WeekdayEnum weekday = WeekdayEnum.None;
  Activity parent = activity.Parent;
  if ((null != parent) && (parent is DaysOfWeekActivity))
     weekday = (WeekdayEnum)DaysOfWeekActivity.GetWeekday(activity);
  return weekday;
 }
 public void SetWeekday(Activity activity, WeekdayEnum weekday)
 {
  Activity parent = activity.Parent;
  if ((null != parent) && (parent is DaysOfWeekActivity))
     DaysOfWeekActivity.SetWeekday(activity, weekday);
 }
}
```

Поставщик расширителей снабжен свойствами, которые он предоставляет, и для каждого из этих свойств он должен обеспечить общедоступные методы `Get<Property>` и `Set<Property>`. Имена этих методов должны соответствовать именам свойства с соответствующим префиксом `Get` или `Set`.

После внесения показанных выше изменений в `Designer` и добавления поставщика расширений, когда вы щелкнете на последовательном действии в визуальном конструкторе, в Visual Studio отобразятся свойства, показанные на рис. 57.14.

Поставщики расширителей используются в .NET и для других целей. Одним из распространенных применений является добавление всплывающих подсказок (tooltip) к элементам управления в проекте Windows Forms. Когда на форму добавляется элемент подсказки, это регистрирует расширитель и добавляет свойство `Tooltip` к каждому элементу управления формы.

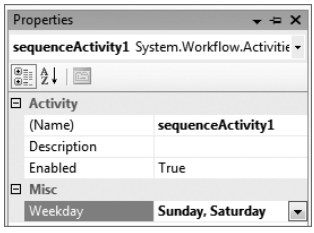

***Рис. 57.14**. Присоединенные свойства*

Рабочие потоки

До этого момента в настоящей главе внимание было сосредоточено на действиях, но не говорилось о рабочих потоках. Рабочий поток (workflow) — это просто список действий, и на самом деле рабочий поток сам по себе является еще одним типом действий. Применение такой модели упрощает механизм исполняющей среды, поскольку ему достаточно знать, как выполнять только один тип объектов — любой, унаследованный от класса `Activity`.

Каждый экземпляр рабочего потока уникально идентифицирован свойством `InstanceId`, т.е. `Guid`, который может быть назначен исполняющей средой, или же этот `Guid` может быть представлен исполняющей среде кодом — обычно это делается для корреляции работающего экземпляра рабочего потока с некоторым внешним источником данных, находящимся вне рабочего потока, таким как строка базы данных. Обратиться к определенному экземпляру рабочего потока можно с помощью метода `GetWorkflow(Guid)` класса `WorkflowRuntime`.

Существуют два типа рабочих потоков, доступных в WF — последовательный и конечный автомат.

Последовательные рабочие потоки

Корневым действием последовательного рабочего потока является `SequentialWorkflow Activity`. Этот класс унаследован от показанного ранее класса `SequenceActivity`, и он определяет два события, к которым при необходимости можно присоединить обработчики — `Initialized` и `Completed`.

Последовательный рабочий поток начинает работу, выполняя находящееся в нем первое дочернее действие, и обычно продолжается до тех пор, пока не выполнит все остальные дочерние действия. Существует пара ситуаций, когда рабочий поток не выполняет всех своих действий. Одна из них — это когда возникает исключение в процессе выполнения рабочего потока, а другая — когда внутри потока встречается `TerminateActivity`.

Рабочий поток может и не выполняться постоянно; например, когда встречается `DelayActivity`, то поток входит в состояние ожидания, и тогда он может быть удален из памяти, если определена служба постоянства потоков. Постоянство потоков описано далее в разделе “Служба постоянства”.

Рабочие потоки типа конечных автоматов

Поток типа конечного автомата используется в случае, когда имеется процесс, который может пребывать в одном из нескольких состояний, и переходы из одного состояния в другое могут выполняться посредством передачи данных в рабочий поток.

Примером может служить рабочий поток, используемый для управления зданием. В этом случае можно моделировать класс двери, которая может быть закрыта или открыта, и класс замка, который может быть замкнут или незамкнут. Изначально, когда вы загружаете

систему (или здание), то начинаете с известного состояния; для простоты предположим, что все двери закрыты и заперты на замок, так что состояние данной двери — *закрыта и заперта*.

Когда сотрудник вводит свой код доступа на внешней двери, он тем самым посылает событие рабочему потоку, включающее такие подробности, как введенный код и, возможно, идентификатор пользователя. Затем может понадобиться обратиться к базе данных для извлечения сведений о том, например, имеет ли данное лицо право открывать выбранную дверь в это время дня, и если имеет, то рабочий поток должен изменить состояние двери с начального на состояние *открыта и не заперта*.

Из этого состояния есть два потенциальных пути — сотрудник открывает дверь (вы знаете это, поскольку дверь снабжена сенсором открытия/закрытия), или же он решит не входить, поскольку забыл что-то у себя в машине, и тогда после небольшой паузы вы запираете дверь. Таким образом, дверь может перейти в свое состояние *закрыта и заперта* или же в состояние *открыта и не заперта*.

Теперь предположим, что сотрудник вошел в здание и закрыл за собой дверь; опять-таки, вы должны выполнить переход из состояния *открыта и не заперта* в состояние *закрыта и не заперта*, и после паузы должен произойти переход в состояние *закрыта и заперта*. Вы также можете выдать сигнал тревоги, если дверь осталась *открытой и не запертой* в течение длительного периода времени.

Смоделировать эту ситуацию в среде Windows Workflow необыкновенно просто. Вам нужно определить состояния, которые может принимать система, и затем определить события, которые могут переводить ее из одного состояния в другое. В табл. 57.2 описаны состояния системы и подробности переходов, которые возможны из каждого состояния, а также ввод (как внешний, так и внутренний), изменяющий эти состояния.

Таблица 57.2. Состояния и переходы системы

Состояние	Переход
Закрыта и заперта	Это начальное состояние системы. В ответ на вставку пользователем карточки (и успешной проверки права доступа) состояние изменяется в “закрыта и не заперта”, т.е. замок двери отпирается электроникой.
Закрыта и не заперта	Когда дверь находится в этом состоянии, может произойти одно из двух событий: • пользователь открывает дверь — выполняется переход в состояние “открыта и не заперта”; • истекает время таймера и дверь возвращается в состояние “закрыта и заперта”.
Открыта и не заперта	Из этого состояния рабочий поток может перейти только в состояние “закрыта и не заперта”.
Пожарная тревога	Это состояние — финальное для рабочего потока, и в него можно перейти из любого другого состояния.

Одно из других средств, которые можно решить добавить в систему — это способность реагировать на сигнал пожарной тревоги. Когда сигнал тревоги будет выключен, нужно отпереть все двери, чтобы все могли покинуть здание, а пожарная команда — проникнуть в него. Вы можете смоделировать это как финальное состояние рабочего потока управляющего дверью, поскольку из этого состояния вся система должна быть сброшена, как только пожарная тревога будет отменена.

Рабочий поток на рис. 57.15 определен как конечный автомат. На нем показаны состояния, которые рабочий поток может принимать, а линиями обозначены возможные переходы из одного состояния в другое.

Начальное состояние рабочего потока смоделировано действием `ClosedLocked`. Оно состоит из некоторого кода инициализации (запирающего дверь) и затем действия на основе события, которое ожидает внешнего события — в данном случае ввода сотрудником собственного кода доступа в здание. Каждое из показанных действий внутри фигуры состояния состоит из последовательных рабочих потоков — таким образом, был определен рабочий поток инициализации системы (`CLInitialize`) и рабочий поток, отвечающий на внешние события, которые возникают при вводе сотрудником своего PIN-кода (`RequestEntry`). Взгляните на определение рабочего потока `RequestEntry` на рис. 57.16.

Все состояния состоят из ряда подпотоков, каждый из которых имеет управляемое событием действие в начале, а затем некоторое количество других действий, которые формируют обрабатывающий код внутри состояния. На рис. 57.16 имеется действие `HandleExternalEventActivity` в начале, которое ожидает ввода PIN. Затем выполняется его проверка, и если код действителен, рабочий поток переходит в состояние `ClosedUnlocked`.

Состояние `ClosedUnlocked` состоит из двух рабочих потоков — один реагирует на событие открытия двери, которое переводит рабочий поток в состояние `OpenUnlocked`, а другой, который содержит действие задержки, используется для перехода в состояние `ClosedLocked`. Действие, управляемое состоянием, работает в манере, аналогичной действию `ListenActivity`, показанному ранее в этой главе — это состояние состоит из ряда управляемых событиями рабочих потоков, и любое возникающее событие обрабатывается только одним рабочим потоком.

Для поддержки рабочего потока вы должны быть готовы инициировать события в системе, чтобы вызвать изменения состояния системы. Это делается за счет использования

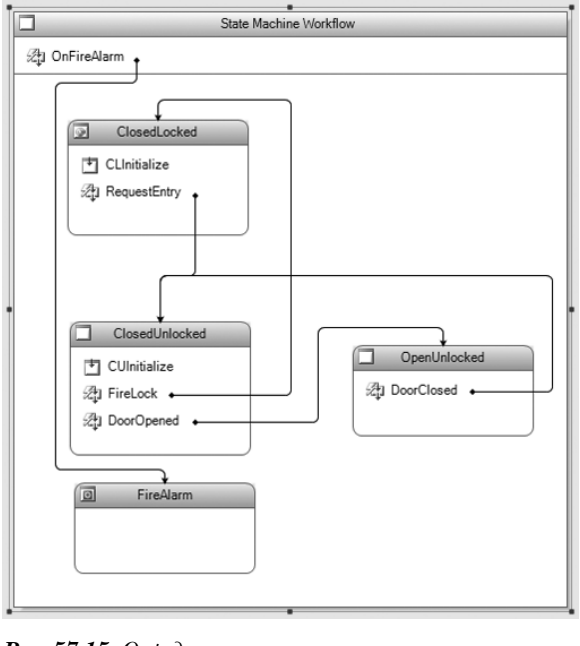

***Рис. 57.15**. Определение конечного автомата*

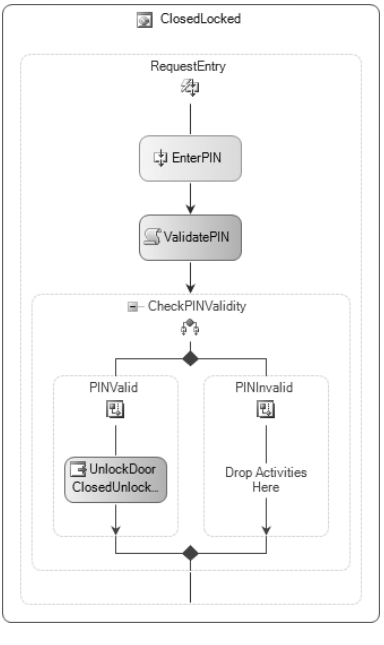

***Рис. 57.16**. Определение рабочего потока `RequestEntry`*

интерфейса и реализации этого интерфейса, и такая пара объектов называется *внешней службой* (external service). Ниже в этой главе будет описан интерфейс, используемый в данном конечном автомате.

Передача параметров рабочему потоку

Типичный рабочий поток для своего выполнения требует некоторых данных. Это может быть идентификатор заказа для потока обработки заказов, идентификатор учетной записи покупателя для рабочего потока обработки платежей либо любые другие необходимые данные.

Механизм передачи параметров для рабочих потоков несколько отличается от того же механизма в стандартных классах .NET, где обычно параметры передаются при вызове метода. Для рабочего потока параметры передаются за счет сохранения их в словаре в виде пар "имя-значение", и когда вы конструируете рабочий поток, то проходите по этому словарю.

Когда WF планирует на выполнение рабочий поток, эти пары "имя-значение" используются для установки значений общедоступных свойств экземпляра рабочего потока. Каждое имя параметра проверяется по общедоступным свойствам рабочего потока и, если обнаружено соответствие, вызывается средство доступа `set` этого свойства и ему передается значение параметра. Если вы добавляете в словарь пару "имя-значение", причем имя не соответствует свойству рабочего потока, то при попытке сконструировать такой рабочий поток будет сгенерировано исключение.

В качестве примера рассмотрим следующий рабочий поток, определяющий целочисленное свойство `OrderID`:

```
public class OrderProcessingWorkflow: SequentialWorkflowActivity
{
public int OrderID
{
  get { return _orderID; }
  set { _orderID = value; }
}
private int _orderID;
}
```

Следующий фрагмент показывает, как можно передать параметр — идентификатор заказа экземпляру этого рабочего потока:

```
WorkflowRuntime runtime = new WorkflowRuntime ();
Dictionary<string,object> parms = new Dictionary<string,object>();
parms.Add("OrderID", 12345) ;
WorkflowInstance instance =
   runtime.CreateWorkflow(typeof(OrderProcessingWorkflow), parms);
instance.Start();
... остальной код
```

В приведенном примере конструируется словарь `Dictionary<string, object>`, содержащий параметры, которые должны быть переданы рабочему потоку и затем использованы при его конструировании. В коде присутствуют классы `WorkflowRuntime` и `WorkflowInstance`, которые еще не были описаны; о них речь пойдет в разделе "Хостинг рабочих потоков" далее в главе.

Возврат результатов из рабочего потока

Другим распространенным требованием, предъявляемым к рабочему потоку, является возврат выходных параметров — возможно, тех, что будут как-то использованы, а затем записаны в базу данных или другое постоянное хранилище.

Поскольку рабочий поток выполняется исполняющей средой рабочего потока, вы не можете просто так вызвать рабочий поток, используя стандартный вызов метода — вы должны создать экземпляр рабочего потока, запустить его и затем ожидать завершения этого экземпляра. Когда рабочий поток завершается, исполняющая среда рабочего потока генерирует событие `WorkflowCompleted`. Оно сопровождается контекстной информацией о рабочем потоке, который только что завершился, и содержит выходные данные этого потока.

Поэтому чтобы получить выходные параметры рабочего потока, необходимо предусмотреть обработчик события `WorkflowCompleted`, и этот обработчик может извлекать выходные параметры из рабочего потока. В следующем коде показан пример того, как это можно сделать.

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
  AutoResetEvent waitHandle = new AutoResetEvent(false);
  workflowRuntime.WorkflowCompleted +=
     delegate(object sender, WorkflowCompletedEventArgs e)
     {
       waitHandle.Set();
       foreach (KeyValuePair<string, object> parm in e.OutputParameters)
       {
         Console.WriteLine("{0} = {1}", parm.Key, parm.Value);
       }
     };
  WorkflowInstance instance = workflowRuntime.CreateWorkflow(typeof(Workflow1));
  instance.Start();
  waitHandle.WaitOne();
}
```

Здесь к событию `WorkflowCompleted` присоединен делегат. Внутри него осуществляется проход по коллекции `OutputParameters` экземпляра класса `WorkflowCompletedEventArgs`, переданного делегату, и все выходные параметры отображаются на консоли. Эта коллекция содержит все общедоступные свойства рабочего потока. Нет никаких пометок специфических выходных параметров рабочего потока.

Привязка параметров к действиям

Теперь, когда известно, как передавать параметры в рабочий поток, также понадобится посмотреть, как связать эти параметры с действиями. Это делается с помощью механизма, называемого привязкой (binding). В ранее определенном действии `DaysOfWeekActivity` имеется свойство `Date`, которое, как упоминалось ранее, может быть жестко закодировано либо привязано к другому значению внутри рабочего потока. Привязываемое свойство отображается в таблице свойств, в среде Visual Studio, как показано на рис. 57.17. Наличие небольшого значка справа от имени свойства `Date` говорит о том, что это привязываемое свойство.

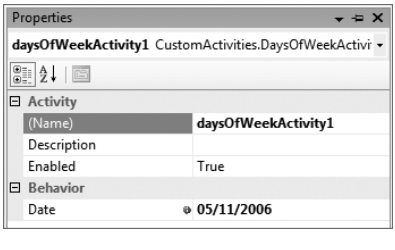

Рис. 57.17. *Привязываемое свойство* `Date`

Двойной щелчок на значке привязки приведет к открытию диалогового окна, показанного на рис. 57.18. Это диалоговое окно позволяет выбрать соответствующее свойство для привязки свойства `Date`.

На рис. 57.18 выбрано свойство `OrderDate` рабочего потока (которое определено как обычное свойство .NET в рабочем потоке). Всякое привязываемое свойство может быть привязано либо к свойству рабочего потока, внутри которого определено действие, либо к свойству любого действия, относящегося к рабочему потоку, которое находится выше текущего действия. Обратите внимание, что тип данных привязываемого свойства должен соответствовать типу данных того свойства, к которому осуществляется привязка — диалоговое окно не позволит связать несоответствующие типы.

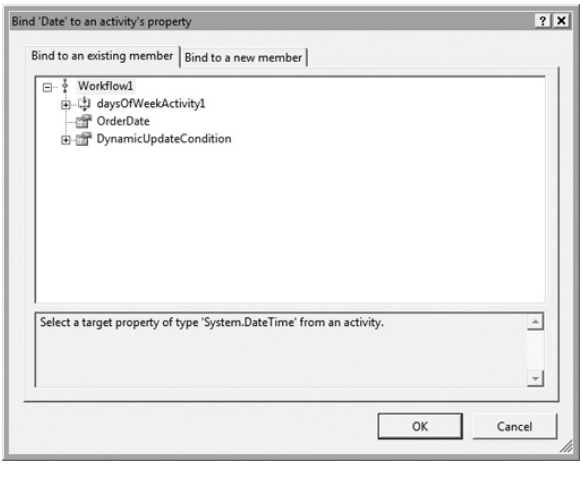

***Рис. 57.18**. Диалоговое окно, отображаемое в результате двойного щелчка на значке привязки*

Ниже повторно приводится код свойства `Date`, чтобы показать, как работает привязка, а объяснения даются далее.

```
public DateTime Date
{
  get { return (DateTime)base.GetValue(DaysOfWeekActivity.DateProperty); }
  set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
}
```

Когда вы привязываете свойство в рабочем потоке, то "за кулисами" конструируется объект типа `ActivityBind`, и это будет тем "значением", которое сохраняется внутри свойства зависимости. Поэтому средству `set` свойства будет передан объект типа `ActivityBind`, и он будет сохранен внутри словаря свойств для данного действия. Объект `ActivityBind` состоит из данных, описывающих действие, к которому выполняется привязка, и свойства данного действия, которое будет использовано во время выполнения.

При чтении значения свойства вызывается метод `GetValue` объекта `DependencyObject`, и этот метод проверяет значение лежащего в основе свойства на предмет того, является ли оно объектом `ActivityBind`. Если это так, затем разрешается действие, к которому выполняется привязка, после чего читается реальное значение свойства из этого действия. Однако если привязанное значение имеет другой тип, тогда просто возвращается объект из метода `GetValue()`.

Исполняющая среда рабочего потока

Для того чтобы запустить рабочий поток, необходимо создать экземпляр класса `WorkflowRuntime`. Как правило, это делается один раз внутри приложения, и такой объект обычно определен как статический член приложения, поэтому он доступен в приложении повсюду.

При запуске исполняющая среда перезагружает все экземпляры рабочих потоков, которые существовали на момент последнего запуска приложения, посредством чтения этих экземпляров из постоянного хранилища. Здесь используется служба, называемая *службой постоянства* (persistence service), которая определяется ниже.

Исполняющая среда содержит методы для конструирования экземпляров рабочих потоков — имеются шесть разных методов `CreateWorkflow`, которые могут применяться для этого, а кроме этого, исполняющая среда также содержит методы для перезагрузки экземпляров и перечисления всех работающих экземпляров.

Исполняющая среда также поддерживает набор событий, которые инициируются на протяжении жизни рабочего потока, такие как `WorkflowCreated` (возникает, когда конструируется новый экземпляр рабочего потока), `WorkflowIdled` (генерируется, когда рабочий поток ожидает ввода, как в приведенном ранее примере обработки отчетов о расходах) и `WorkflowCompleted` (инициируется при завершении рабочего потока).

Службы рабочих потоков

Рабочий поток не существует сам по себе. Как показано в предыдущих разделах, рабочий поток выполняется внутри исполняющей среды `WorkflowRuntime`, и эта исполняющая среда предоставляет *службы* запущенным рабочим потокам.

Служба — это любой класс, который может понадобиться во время выполнения рабочего потока. Некоторые стандартные службы предоставляются потокам исполняющей средой, и по желанию можно конструировать собственные службы, которые будут применяться запущенными рабочими потоками.

В настоящем разделе описаны две стандартных службы, предлагаемые исполняющей средой, и затем будет показано, как создавать собственные службы и некоторые экземпляры, когда это необходимо.

Когда выполняется действие, ему передается некоторая контекстная информация в параметре `ActivityExecutionStatus` метода `Execute`:

```
protected override ActivityExecutionStatus Execute
    (ActivityExecutionContext executionContext)
{
    ...
}
```

Одним из доступных методов в этом контекстном параметре является `GetService<T>`. Как показано в следующем коде, он может использоваться для доступа к службе, присоединенной к исполняющей среде рабочего потока.

```
protected override ActivityExecutionStatus Execute
    (ActivityExecutionContext executionContext)
{
  ICustomService myService = executionContext.GetService<ICustomService>();
... Делать что-нибудь со службой
}
```

Службы, развернутые в исполняющей среде, добавляются в нее до вызова метода `StartRuntime`; если вы попытаетесь добавить службу к исполняющей среде после ее запуска, возникнет исключение.

Для добавления служб к исполняющей среде предусмотрены два метода: вы можете конструировать службы в коде и затем добавлять их к исполняющей среде вызовом метода AddService, либо определить службы внутри конфигурационного файла приложения, и они будут автоматически сконструированы и добавлены к исполняющей среде.

В следующем фрагменте кода показано, как добавлять службы во время выполнения в коде. Добавляемые службы описаны ниже в этом разделе.

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
  workflowRuntime.AddService(
     new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                       new TimeSpan(0,10,0)));
  workflowRuntime.AddService(new SqlTrackingService(conn));
...
}
```

Здесь конструируется экземпляр SqlWorkflowPersistenceService, который используется исполняющей средой для сохранения состояния рабочего потока, а также экземпляр SqlTrackingService, который записывает события, происходящие в рабочем потоке во время его работы.

Чтобы создать службы с помощью конфигурационного файла приложения, нужно добавить соответствующий обработчик раздела для исполняющей среды рабочего потока, после чего добавить службы в этот раздел:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<configSections>
  <section name="WF"
     type="System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,
     System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
     PublicKeyToken=31bf3856ad364e35" />
</configSections>
<WF Name="Hosting">
  <CommonParameters/>
     <Services>
        <add type="System.Workflow.Runtime.Hosting.
        SqlWorkflowPersistenceService,
                  System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
                  PublicKeyToken=31bf3856ad364e35"
             connectionString="Initial Catalog=WF;Data Source=.;
                  Integrated Security=SSPI;"
             UnloadOnIdle="true"
             LoadIntervalSeconds="2"/>
        <add type="System.Workflow.Runtime.Tracking.SqlTrackingService,
                     System.Workflow.Runtime, Version=3.0.00000.0,
                     Culture=neutral,
                     PublicKeyToken=31bf3856ad364e35"
             connectionString="Initial Catalog=WF;Data Source=.;
                              Integrated Security=SSPI;"
             UseDefaultProfile="true"/>
     </Services>
</WF>
</configuration>
```

Внутри конфигурационного файла вы добавляете обработчик раздела WF (имя не имеет значения, но должно совпадать с именем, приведенным ниже в разделе конфигурации), а затем создаете соответствующие элементы для этого раздела. Элемент <Services> может содержать произвольный список элементов, который состоит из типа .NET и параметров, передаваемых службе при ее конструировании исполняющей средой.

Для чтения настроек из файла конфигурации приложения во время выполнения вызывается другой конструктор, как показано ниже:

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime("WF"))
{
    ...
}
```

Этот конструктор создает экземпляры каждой службы, определенной внутри конфигурационного файла, и добавляет их в коллекцию служб во время выполнения.

В следующем разделе рассматриваются некоторые стандартные службы, доступные в WF.

Служба постоянства

Когда рабочий поток выполняется, он может достичь состояния ожидания — такое может произойти, когда выполняется действие ожидания или когда вы ждете внешнего ввода в действии прослушивания. В этот момент говорят, что рабочий поток *простаивает* (idle) и потому является кандидатом на сохранение.

Предположим, что вы начали выполнение 1000 рабочих потоков на сервере, и каждый из этой тысячи потоков попал в состояние простоя. В этот момент нет необходимости поддерживать данные всех этих экземпляров в памяти, так что в идеале было бы иметь возможность выгрузить рабочий поток и освободить ресурсы для других задач. Служба постоянства (persistence service) предназначена специально для этого.

Когда рабочий поток достигает состояния простоя, исполняющая среда проверяет существование службы, унаследованной от класса `WorkflowPersistenceService`. Если такая служба существует, ей передается экземпляр рабочего потока, и она затем может зафиксировать текущее состояние рабочего потока и сохранить его в постоянном хранилище. Вы можете сохранить состояние рабочего потока на диске или в файле либо в базе данных, такой как SQL Server.

Библиотеки рабочих потоков содержат реализацию службы постоянства, которая сохраняет данные в базе SQL Server — это класс `SqlWorkflowPersistenceService`. Для того чтобы использовать упомянутую службу, нужно запустить два сценария на экземпляре SQL Server: один из них конструирует схему, а другой создает хранимые процедуры, используемые службой постоянства. Эти сценарии по умолчанию расположены в каталоге `C:\Windows\Microsoft.NET\Framework\v3.5\Windows Workflow Foundation\SQL\EN`.

Сценарии для выполнения в базе данных называются `SqlPersistenceService_Schema.sql` и `SqlPersistenceService_Logic.sql`. Они должны запускаться по порядку, сначала — файл схемы, затем — файл логики. Схема для службы постоянства SQL содержит две таблицы: `InstanceState` и `CompletedScope`; это, по сути, закрытые таблицы, которые не предназначены для использования вне службы постоянства SQL.

Когда рабочий поток простаивает, его состояние сериализуется посредством бинарной сериализации, и эти данные помещаются в таблицу `InstanceState`. Когда рабочий поток затем повторно активизируется, его состояние считывается из строки таблицы и используется для воссоздания экземпляра рабочего потока. Строка таблицы помечается идентификатором экземпляра рабочего потока и удаляется из базы данных по завершении этого экземпляра.

Служба постоянства SQL может применяться несколькими исполняющими средами одновременно — она реализует механизм блокировки, так что рабочий поток доступен только одному экземпляру исполняющей среды за раз. Когда имеется несколько серверов, и все они выполняют рабочие потоки, взаимодействующие с одним и тем же постоянным хранилищем, это блокирующее поведение становится несущественным.

Чтобы увидеть, что именно добавляется в постоянное хранилище, сконструируйте новый проект рабочего потока и добавьте к исполняющей среде экземпляр `SqlWorkflowPersistenceService`.

Ниже приведен пример применения декларативного кода:

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
  workflowRuntime.AddService(
     new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                       new TimeSpan(0,10,0)));
  // Здесь должен запускаться рабочий поток.
}
```

Если вы конструируете рабочий поток, содержащий `DelayActivity`, и устанавливаете величину задержки около 10 секунд, то затем можете увидеть его данные в таблице `InstanceState`.

Параметры, передаваемые конструктору службы постоянства, перечислены в табл. 57.3.

Таблица 57.3. Параметры, передаваемые конструктору службы постоянства

Параметр	Описание	Значение по умолчанию
`ConnectionString`	Строка подключения к базе данных, используемой службой постоянства.	Нет
`UnloadOnIdle`	Определяет, будет ли рабочий поток выгружен во время простоя. Всегда следует устанавливать в `true`, иначе никакого постоянного хранения не произойдет.	`False`
`InstanceOwnershipDuration`	Определяет длительность периода времени, когда экземпляр рабочего потока будет принадлежать исполняющей среде, загрузившей этот поток.	Нет
`LoadingInterval`	Интервал, используемый для обновления в базе данных записей постоянного хранения.	2 минуты

Значения могут быть также определены в конфигурационном файле.

Служба отслеживания

Во время выполнения рабочего потока может возникнуть необходимость зафиксировать, какие действия уже были выполнены, а в случае составного действия наподобие `IfElseActivity` или `ListenActivity` — какая именно ветвь выполняется. Эти данные могут применяться в качестве журнала аудита для экземпляра рабочего потока, который позднее можно просмотреть на предмет того, какие действия были выполнены и какие данные использовались в рабочем потоке. Для такого рода протоколирования может применяться служба отслеживания (tracking service), и при необходимости ее можно настроить на фиксацию как можно большего или как можно меньшего объема информации о выполняющемся рабочем потоке.

Как принято в WF, служба отслеживания реализована в виде абстрактного базового класса по имени `TrackingService`, так что очень просто заменить стандартную реализацию отслеживания собственной. Существует одна конкретная реализация службы отслеживания, доступная в сборках Windows Workflow — `SqlTrackingService`.

Чтобы записать данные о состоянии рабочего потока, необходимо определить `TrackingProfile`. С помощью этого объекта указываются события, которые должны быть записаны, так что можно, например, фиксировать только старт и завершение рабочего по-

тока, пропуская все прочие данные, поступающие от выполняющегося экземпляра. Более типичный случай — протоколирование всех событий рабочего потока и каждого действия в нем, что позволяет зафиксировать полную картину профиля выполнения рабочего потока.

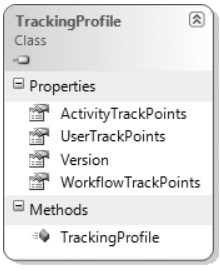

***Рис. 57.19**. Класс профиля отслеживания*

Когда рабочий поток планируется механизмом исполняющей среды, этот механизм проверяет существование службы отслеживания рабочего потока. Если таковая найдена, у нее запрашивается профиль отслеживания для выполняющегося рабочего потока, и затем он применяется для записи работы потока и данных о его действиях. В дополнение можно определять пользовательские данные отслеживания и сохранять их внутри хранилища для таких данных, причем без необходимости изменения схемы.

Класс профиля отслеживания показан на рис. 57.19. Этот класс содержит свойства-коллекции для *точек отслеживания* действий, пользователя и рабочего потока в целом. Точка отслеживания — это объект (вроде `WorkflowTrackPoint`), который обычно определяет *место соответствия* (mach location) и некоторые дополнительные данные для записи при попадании в эту точку отслеживания. Место соответствия определяет, когда точка отслеживания действительна, поэтому, например, можно определить одну точку `WorkflowTrackPoint`, в которой будет выполняться запись некоторых данных при создании рабочего потока, а другую — для записи тех же данных при завершении рабочего потока.

После того как данные записаны, может оказаться необходимым отобразить путь выполнения рабочего потока, например, как показано на рис. 57.20. Здесь можно видеть выполненный рабочий поток, причем каждое его запущенное действие помечено специальным значком — галочкой. Данные для этого отображения читаются из хранилища информации об отслеживании данного экземпляра рабочего потока.

Для того чтобы прочесть данные, сохраненные посредством `SqlTrackingService`, можно напрямую отправлять запросы базе данных SQL, однако Microsoft предлагает для этого класс `SqlTrackingQuery`, определенный в пространстве имен `System.Workflow.Runtime.Tracking`.

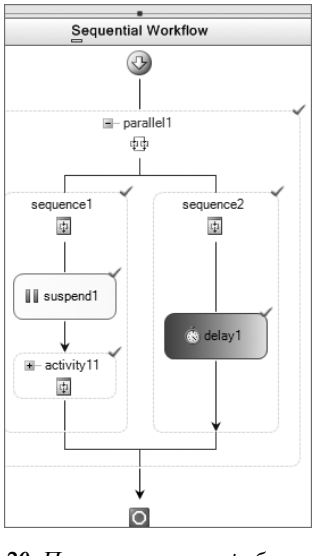

***Рис. 57.20**. Путь выполнения рабочего потока*

В приведенном ниже примере кода показано, как извлечь список рабочих потоков, которые отслеживались между двумя датами:

```
public IList<SqlTrackingWorkflowInstance> GetWorkflows
    (DateTime startDate, DateTime endDate, string connectionString)
{
  SqlTrackingQuery query = new SqlTrackingQuery (connectionString);
  SqlTrackingQueryOptions queryOptions = new SqlTrackingQueryOptions();
  query.StatusMinDateTime = startDate;
  query.StatusMaxDateTime = endDate;
  return (query.GetWorkflows (queryOptions));
}
```

Здесь используется класс `SqlTrackingQueryOptions`, определяющий параметры запроса. Для дополнительного ограничения выборки рабочих потоков можно определить другие свойства этого класса.

На рис. 57.20 видно, что все действия потока были выполнены — так может быть не всегда, если рабочий поток все еще выполняется, или если некоторые решения, принятые внутри потока, привели к выбору разных путей выполнения.

Данные отслеживания содержат такие подробности, как то, какие действия были выполнены, и эта информация может сравниваться с общим списком действий, чтобы сгенерировать картинку вроде показанной на рис. 57.20. Можно также извлечь данные из рабочего потока во время выполнения, которые будут применяться для формирования журнала аудита по выполнению этого рабочего потока.

Пользовательские службы

В дополнение к встроенным службам, таким как служба постоянства и служба отслеживания, можно также добавлять собственные объекты в коллекцию служб, поддерживаемую `WorkflowRuntime`. Эти службы обычно определяются с применением интерфейса и реализации, так что легко заменять одну службу другой, не перекодируя рабочий поток.

Ранее представленный в этой главе конечный автомат использует следующий интерфейс:

```
[ExternalDataExchange]
public interface IDoorService
{
  void LockDoor();
  void UnlockDoor();
  event EventHandler<ExternalDataEventArgs> RequestEntry;
  event EventHandler<ExternalDataEventArgs> OpenDoor;
  event EventHandler<ExternalDataEventArgs> CloseDoor;
  event EventHandler<ExternalDataEventArgs> FireAlarm;
  void OnRequestEntry(Guid id);
  void OnOpenDoor(Guid id);
  void OnCloseDoor(Guid id);
  void OnFireAlarm();
}
```

Интерфейс состоит из методов, применяемых рабочим потоком для вызова службы, и событий, инициируемых службой, которая используется рабочим потоком. Применение атрибута `ExternalDataExchange` указывает на то, что исполняющая среда рабочего потока применяется для коммуникаций между работающим потоком и реализацией службы.

Внутри конечного автомата присутствует ряд экземпляров `CallExternalMethod Activity`, используемых для вызова методов этого внешнего интерфейса. Примером может быть ситуация, когда дверь запирается и отпирается. Рабочий поток при этом должен вызвать метод `UnlockDoor` или `LockDoor`, и служба реагирует на это отправкой соответствующей команды дверному замку.

Когда служба нуждается во взаимодействии с рабочим потоком, то делает это через события; исполняющая среда рабочих потоков также включает службу по имени ExternalDataExchangeService, которая служит прокси для таких событий. Прокси используется при инициировании события, но поскольку рабочий поток может и не быть загруженным в память на момент доставки события, это событие сначала маршрутизируется внешней службе обмена, которая проверяет, загружен ли рабочий поток, и если нет, то восстанавливает его из постоянного хранилища и затем передает событие в этот рабочий поток.

Код, применяемый для конструирования ExternalDataExchangeService и прокси для событий, определенных в службе, показан ниже:

```
WorkflowRuntime runtime = new WorkflowRuntime();
ExternalDataExchangeService edes = new ExternalDataExchangeService();
runtime.AddService(edes);
DoorService service = new DoorService();
edes.AddService(service);
```

Этот код конструирует экземпляр внешней службы обмена, добавляет его в исполняющую среду, затем создает экземпляр DoorService (реализующий IDoorService) и добавляет его во внешнюю службу обмена данными.

Метод ExternalDataExchangeService.Add конструирует прокси для каждого события, определенного пользовательской службой, так что сохраненный рабочий поток может быть загружен до того, как будет доставлено событие. Если вы не помещаете службу внутрь внешней службы обмена данными, то когда вы станете инициировать события, никто не будет их прослушивать, поэтому они не будут доставлены соответствующему рабочему потоку.

События используют класс ExternalDataEventArgs, поскольку он включает идентификатор экземпляра рабочего потока, которому должно быть доставлено событие. Если существуют и другие значения, которые нужно передать из внешнего события в рабочий поток, то вы должны унаследовать класс от ExternalDataEventArgs и добавить требуемые значения в виде свойств в этот класс.

Интеграция с Windows Communication Foundation

В .NET 3.5 доступны два новых действия, которые поддерживают интеграцию между рабочими потоками и WCF — SendActivity и ReceiveActivity. Действие SendActivity часто может сразу вызывать CallActivity, потому что оно запрашивает службу WCF и дополнительно организует результаты в виде параметров, которые могут быть привязаны к внутренностям вызывающего рабочего потока.

Однако более интересным является новое действие ReceiveActivity. Оно позволяет рабочему потоку стать реализацией службы WCF, так что рабочий поток становится службой. В следующем примере представлена служба, использующая рабочий поток и также новый инструмент тестирования служб, который позволяет протестировать ее без написания отдельного тестового кода.

В Visual Studio 2010 в меню New Project (Новый проект) выберите WCF и затем — Sequential Workflow Service Library (Библиотека службы последовательного потока), как показано на рис. 57.21.

Это создаст библиотеку, содержащую рабочий поток, как показано на рис. 57.22, а также конфигурационный файл приложения и интерфейс службы. Код этого примера находится в подкаталоге 06 ExposeWFService.

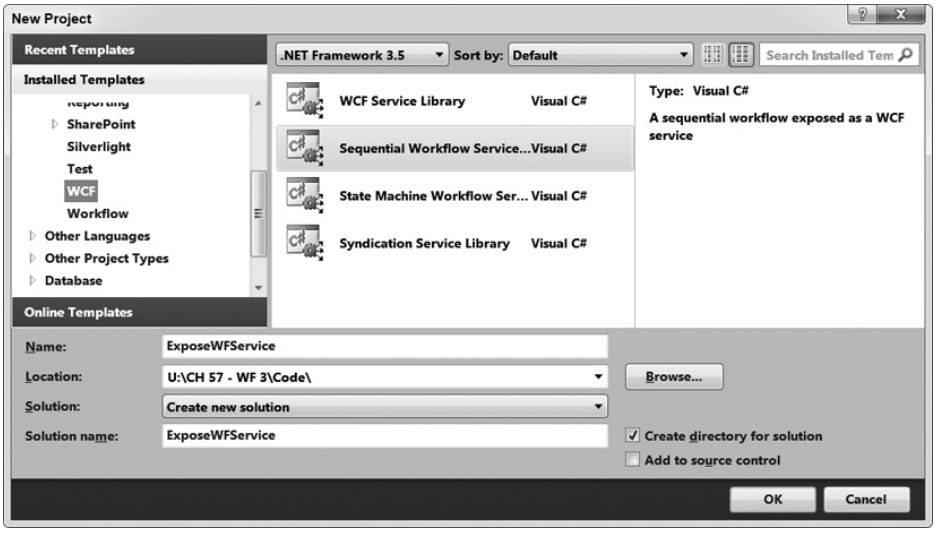

***Рис. 57.21**. Создание библиотеки службы последовательного потока*

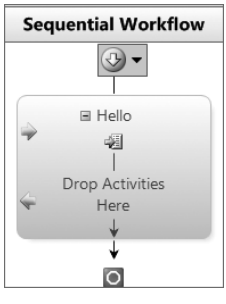

***Рис. 57.22**. Библиотека содержит последовательный поток*

Рабочий поток представляет операцию `Hello` контракта и также определяет свойства для аргументов, передаваемых этой операции, и возвращаемое значение этой операции. Затем все, что остается сделать — это добавить код, представляющий исполняемое поведение службы — и служба готова.

Чтобы сделать это в примере, перетащите `CodeActivity` на `ReceiveActivity`, как показано на рис. 57.23, а затем дважды щелкните на этом действии для применения реализации службы:

```
public sealed partial class Workflow1: SequentialWorkflowActivity
{
  public Workflow1()
  {
    InitializeComponent();
  }
  public String returnValue = default(System.String);
  public String inputMessage = default(System.String);
  private void codeActivity1_ExecuteCode(object sender, EventArgs e)
  {
    this.returnValue = string.Format("You said {0}", inputMessage);
  }
}
```

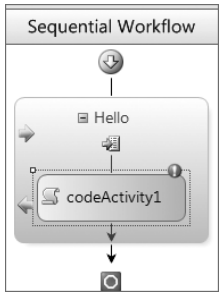

***Рис. 57.23**. Добавление действия* `CodeActivity`

Поскольку контракт службы для операции `Hello` включает как параметр (`inputMessage`), так и возвращаемое значение, все это представлено в рабочем потоке в виде общедоступных полей. В коде потребуется установить `returnValue` в строковое значение, и это будет возвращено из вызова службе WCF.

Если вы скомпилируете эту службу нажатием <F5>, то заметите другое новое средство Visual Studio 2010 — приложение WCF Test Client (Тестовый клиент WCF), показанное на рис. 57.24.

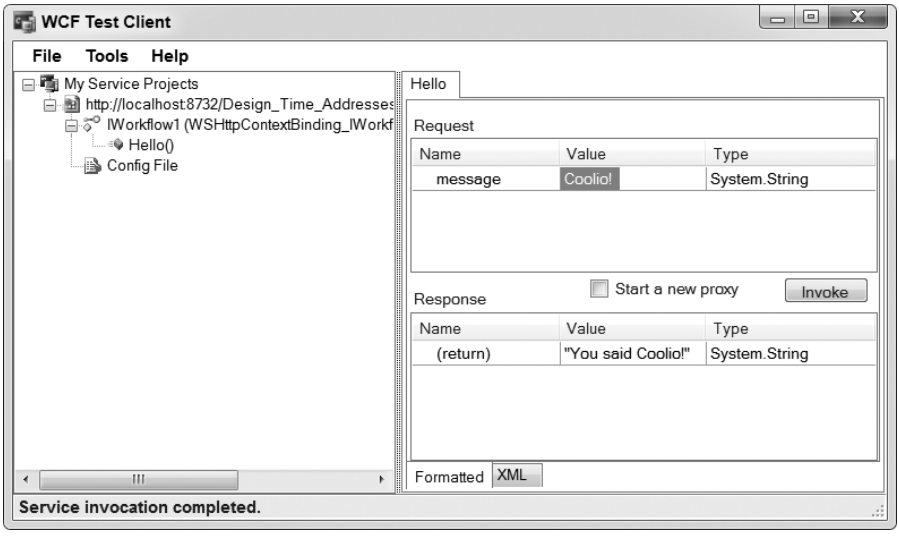

***Рис. 57.24**. Приложение WCF Test Client*

Здесь можно просмотреть операции, представленные службой, а при двойном щелчке на операции будет отображена правая сторона окна со списком параметров, используемых службой, а также ее возвращаемые значения.

Чтобы протестировать службу, введите значение для свойства сообщения и щелкните на кнопке Invoke (Вызвать). Это инициирует запрос к службе через WCF, который сконструирует и запустит рабочий поток, вызовет код действия, запускающий отделенный код, и в конечном итоге вернет в WCF Test Client результат из рабочего потока.

Если хотите вручную развернуть рабочий поток как службу, то можете воспользоваться классом `WorkflowServiceHost`, определенным внутри сборки `System.WorkflowService`.

В следующем фрагменте кода демонстрируется минимальная реализация хоста:

```
using (WorkflowServiceHost host = new WorkflowServiceHost
                            (typeof(YourWorkflow)))
{
  host.Open();
  Console.WriteLine("Press [Enter] to exit");
  Console.ReadLine();
}
```

Здесь мы конструируем экземпляр `WorkflowServiceHost` и передаем ему рабочий поток, который будет выполнен. Это подобно тому, как если бы вы использовали класс `ServiceHost` при хостинге служб WCF. Будет прочитан конфигурационный файл для определения конечных точек, прослушиваемых службой, и затем будет инициировано ожидание запросов.

В следующем разделе описаны некоторые прочие опции, которые доступны при хостинге рабочих потоков.

Хостинг рабочих потоков

Код для хостинга `WorkflowRuntime` в процессе будет варьироваться в зависимости от самого приложения.

Для приложений Windows Forms или Windows Service (служба Windows) типичным является конструирование исполняющей среды при запуске приложения и сохранение ее в свойстве главного класса приложения.

В ответ на некоторый ввод в приложении (например, щелчок на кнопке пользовательского интерфейса) можно сконструировать экземпляр рабочего потока и запустить этот экземпляр локально. Рабочему потоку может понадобиться взаимодействие с пользователем, поэтому, например, можно определить внешнюю службу, которая запросит пользовательского подтверждения, прежде чем отправит запрос серверу баз данных.

При хостинге рабочих потоков в ASP.NET обычно вы не будете предупреждать пользователя через окно сообщения, а вместо этого выполните переход на другую страницу сайта, которая сначала затребует подтверждения и затем предоставит соответствующую страницу подтверждения. При развертывании исполняющей среды внутри ASP.NET принято переопределять событие `Application_Start` и конструировать в нем экземпляр исполняющей среды рабочего потока, который будет доступен всем прочим частям сайта. Сохранять экземпляр исполняющей среды можно в статическом свойстве, но лучше сохранить его в состоянии приложения и предусмотреть метод для доступа, который будет извлекать исполняющую среду из состояния приложения, чтобы ее можно было использовать в любом месте приложения.

При любом сценарии — Windows Forms или ASP.NET — будет конструироваться экземпляр исполняющей среды рабочего потока с добавлением служб так, как показано ниже.

```
WorkflowRuntime workflowRuntime = new WorkflowRuntime();
workflowRuntime.AddService(
   new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                     new TimeSpan(0,10,0)));
// Выполнить рабочий поток.
```

Для выполнения рабочего потока понадобится создать экземпляр этого рабочего потока, вызвав для этого метод `CreateInstance` исполняющей среды. У этого метода есть ряд перегрузок, которые можно применять для конструирования экземпляра рабочего потока на основе кода либо рабочего потока, определенного с помощью XML.

Вплоть до этого места настоящей главы вы рассматривали рабочие потоки как классы .NET — и в самом деле это одно представление рабочего потока. Однако можно определить

рабочий поток, используя для этого XML, а исполняющая среда создаст его представление в памяти и затем выполнит его, когда вы вызовете метод `Start` из `WorkflowInstance`.

Внутри Visual Studio можно создать рабочий поток на основе XML, выбрав элемент Sequential Workflow (with code separation) (Последовательный рабочий поток (с разделением кода)) или State Machine Workflow (with code separation) (Рабочий поток в виде конечного автомата (с разделением кода)) в диалоговом окне Add New Item (Добавить новый элемент). Это создаст XML-файл с расширением `.xoml` и загрузит его в визуальный конструктор.

По мере добавления действий в конструкторе они будут отражаться в XML, и структура элементов определит отношения "родительский–дочерний" между действиями. В приведенном ниже XML демонстрируется простой последовательный рабочий поток, содержащий `IfElseAcvtivity` и два кодовых действия, по одному на каждую ветвь `IfElseAcvtivity`.

```
<SequentialWorkflowActivity x:Class="DoorsWorkflow.Workflow1"
x:Name="Workflow1"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
<IfElseActivity x:Name="ifElseActivity1">
  <IfElseBranchActivity x:Name="ifElseBranchActivity1">
    <IfElseBranchActivity.Condition>
       <CodeCondition Condition="Test" />
    </IfElseBranchActivity.Condition>
    <CodeActivity x:Name="codeActivity1" ExecuteCode="DoSomething" />
  </IfElseBranchActivity>
  <IfElseBranchActivity x:Name="ifElseBranchActivity2">
    <CodeActivity x:Name="codeActivity2" ExecuteCode="DoSomethingElse" />
  </IfElseBranchActivity>
  </IfElseActivity>
</SequentialWorkflowActivity>
```

Свойства, определенные в действиях, сохраняются в XML в виде атрибутов, а каждое действие сохраняется как элемент. В XML несложно заметить, что отношение между родительскими действиями (такими как `SequentialWorkflowActivity` и `IfElseActivity`) и дочерними действиями определяется структурой.

Выполнение рабочего потока на основе XML никак не отличается от выполнения рабочего потока, описанного в коде — вы просто используете перегрузку метода `CreateWorkflow`, принимающего экземпляр `XmlReader`, и затем запускаете этот экземпляр, вызывая метод `Start`.

Одно из преимуществ рабочих потоков на основе XML перед рабочими потоками на основе кода — это возможность сохранения в базе данных его определения. Этот XML-код можно загрузить во время выполнения и запустить рабочий поток; кроме того, очень легко вносить изменения в определение рабочего потока без необходимости перекомпиляции какого-либо кода.

Изменение рабочего потока во время выполнения поддерживается независимо от того, определен он в XML или в коде. Вы просто конструируете объект `WorkflowChanges`, содержащий все новые действия, которые нужно добавить к рабочему потоку, и затем вызываете метод `ApplyWorkflowChanges`, определенный в классе `WorkflowInstance`, для фиксации этих изменений. Это исключительно удобно, поскольку бизнес-правила часто меняются и, к примеру, может потребоваться применить изменения к рабочему потоку политики страхования, чтобы клиенту отправлялось электронное письмо с соответствующим уведомлением за месяц до наступления даты изменений. Изменения вносятся на уровне экземпляра, поэтому при наличии 100 таких рабочих потоков политики в системе придется внести изменения в каждый из них.

Инструмент Workflow Designer

В завершение этой главы мы рассмотрим самое интересное. Инструмент проектирования Workflow Designer, который применяется для проектирования рабочих потоков, не привязан к Visual Studio. При необходимости его можно развернуть в среде собственного приложения.

Это означает возможность поставки системы, включающей рабочие потоки, которая позволяет пользователям проводить настройку под свои нужды без необходимости приобретения копии Visual Studio. Развертывание конструктора, однако, представляет собой довольно сложную задачу, и ей можно было бы посвятить несколько глав. В Интернете можно найти немало примеров развертывания этого конструктора, кроме того, рекомендуется почитать статью MSDN на данную тему, которая доступна по адресу:

```
http://msdn.microsoft.com/ru-ru/library/aa480213.aspx
```

Традиционный способ позволить пользователям настраивать систему заключается в определении интерфейса и затем предоставлении им самостоятельно его реализовывать для расширения функциональности.

В Windows Workflow это расширение становится в значительной степени графическим, поскольку пользователю можно предложить пустой рабочий поток в качестве шаблона и обеспечить панелью инструментов, включающей специальные действия, которые отвечают нуждам приложения. Затем пользователь самостоятельно может конструировать свои рабочие потоки, добавляя ваши действия или разрабатывая собственные.

Переход от WF 3.x к WF 4

Вместе с .NET 4 и Visual Studio .NET 2010 поставляется новая версия WF, которая не поддерживает обратную совместимость с WF 3.x. Концепции в основном остаются прежними, однако реализация совершенно иная. Можно продолжать использовать рабочие потоки WF 3.x без каких-либо модификаций, однако, если нужно получить преимущества от новых средств, потребуется провести миграцию существующих приложений. В этом разделе описаны основные моменты процесса миграции и приведены некоторые советы по упрощению этого процесса.

Более подробно технология WF 4 рассматривается в главе 44.

Извлечение кода действий в службы

Иерархия классов WF 4 существенно отличается от WF 3.x; однако класс действия в основном остается прежним, поэтому для упрощения обновления рекомендуется исключить встроенный код из действий и перенести его в набор служб (таких как простые классы .NET, которые содержат статические методы).

В WF 4 существенные изменения претерпела привязка данных: в то время как в WF 3.x можно использовать свойства зависимости и/или стандартные свойства .NET, в WF 4 должны применяться аргументы-объекты. Простейший способ выполнить миграцию кода предусматривает перенос обработки из действия в отдельный класс. В качестве тривиального примера рассмотрим фрагмент действия `WriteLine`:

```
public class WriteLineActivity : Activity
{
    public string Message { get; set; }
```

```
    public override ActivityExecutionStatus Execute
        (ActivityExecutionContext context)
    {
        Console.WriteLine(Message);
        return ActivityExecutionStatus.Closed;
    }
}
```

Чтобы преобразовать это в действие WF 4, можно создать класс следующего вида:

```
public static class WriteLineService
{
    public static void WriteLine(string message)
    {
        Console.WriteLine(message);
    }
}
```

Затем его можно использовать внутри текущего действия (и также выполнять модульное тестирование в изоляции), а после обновления до WF 4 придется писать меньше кода. Этот тривиальный пример демонстрирует, как упростить обновление.

Удаление кода действий

В WF 4 не поддерживается тот же стиль отделенного кода для действий, что в WF 3.x, поэтому при наличии какого-то кода в файле отделенного кода нужно последовать приведенной ранее рекомендации и перенести его в библиотеку, чтобы иметь возможность использовать его в WF 4.

Запуск WF 3.x и WF 4 бок о бок

По возможности старайтесь выполнять потоки WF 3.x бок о бок с потоками WF 4 до тех пор, пока существуют потоки WF 3.x. Постоянное хранилище и хранилище отслеживания отличаются в WF 4, поэтому комбинированные системы будут самым простым способом. При отсутствии давно используемых рабочих потоков вопрос запуска WF 3.x и WF 4 бок о бок является спорным; однако если такие потоки есть, то проще оставить их неизменными, а новые рабочие потоки создавать в версии WF 4.

Когда приложение запускает рабочий поток, можно ли легко заменить данный код чем-то, что запустит рабочий поток WF 4? Если нет, рекомендуется добавить в код некоторую политику поддержки разных версий. Это позволит при планировании запуска нового экземпляра рабочего потока не изменять класс, вызывающий рабочий поток.

Перевод конечных автоматов в блок-схемы

В настоящее время WF 4 не поддерживает конечные автоматы — во всяком случае, пока что встроенная их поддержка отсутствует. Лежащая в основе модель действий может обрабатывать конечные автоматы (как и любая модель рабочего потока); однако в начальном выпуске WF 4 нет действия типа конечного автомата и не реализована их поддержка времени проектирования.

Ближайшим аналогом конечных автоматов WF 3.x в версии WF 4 является блок-схема (flowchart). Это не точное соответствие, но блок-схемы допускают переходы к более ранним стадиям обработки, что обычно делают рабочие потоки типа конечных автоматов.

Резюме

Технология Windows Workflow приведет к радикальным изменениям в способе конструирования приложений. Вы можете теперь сформировать сложные части приложения в виде действий и позволить пользователям изменять работу системы простым перетаскиванием действий в рабочий поток.

Практически не существует приложений, к которым нельзя было бы применить концепцию рабочего потока — от простейших утилит командной строки до наиболее сложных систем, состоящих из многих сотен модулей. В то время как новые средства WCF и средства пользовательского интерфейса WPF являются значительным шагом вперед для приложений в целом, появление технологии Windows Workflow приведет к фундаментальным изменениям в способах разработки и конфигурирования приложений.

В Visual Studio 2010 версия WF 3.x в основном заменена версией WF 4, и в этой главе были приведены некоторые советы, позволяющие упростить переход к новой версии. Если вы только планируете впервые приступить к использованию рабочих потоков, рекомендуется сразу начать с версии WF 4, полностью минуя WF 3.x.