45

Windows Workflow Foundation

В ЭТОЙ ГЛАВЕ...

- Типы рабочих потоков, которые можно создавать
- Описание некоторых встроенных действий
- Создание специальных действий
- Краткое рассмотрение рабочего потока

Загружаемый код для этой главы

Загружаемый код для этой главы содержит следующие основные примеры:

- `01_HelloWorld`
- `02_ParallelExecution`
- `03_PickDemo`
- `04_CustomActivities`
- `05_ArgsAndVars`
- `06_WorkflowApplication`
- `07_WorkflowsAsServices`
- `08_DynamicUpdate`
- `09_DesignerRehosting`

Обзор WF

В настоящей главе предлагается краткий обзор инфраструктуры Windows Workflow Foundation 4.5 (в дальнейшем просто Workflow или WF), которая предоставляет модель, позволяющую определять и выполнять процессы с помощью ряда строительных блоков, называемых *действиями*. Инфраструктура WF предлагает визуальный конструктор, по умолчанию размещенный в Visual Studio, который позволяет перетаскивать действия из панели инструментов на поверхность проектирования и тем самым создавать шаблон рабочего потока.

Этот шаблон затем может выполняться разнообразными способами, которые объясняются далее в главе. Во время выполнения рабочему потоку может понадобиться доступ к внешнему миру, и для обеспечения такой возможности существует несколько распространенных методов. Вдобавок рабочему потоку может требоваться сохранять и восстанавливать свое состояние, например, когда необходимо длительное ожидание.

Рабочий поток конструируется из нескольких действий, которые запускаются во время выполнения. Действие может отправлять сообщение по электронной почте, обновлять строку в базе данных или выполнять транзакцию на серверной системе. Существует набор встроенных действий, которые можно применять для выполнения задач общего назначения. Разумеется, можно создавать собственные действия и должным образом включать их в рабочий поток.

На самом деле в Visual Studio 2012 теперь доступны две версии WF с разными типами: версия WF 3.*x*, которая поставляется вместе с .NET Framework 3 (пространство имен `System.Workflow` и его вложенные пространства имен, используемые также в SharePoint 2010), и версия WF 4.*x* (пространство имен `System.Activities` и его вложенные пространства), входящая в состав .NET Framework 4.

Эта глава посвящена последней версии Workflow. Мы начнем с рассмотрения канонического примера, которым пользуются все, когда сталкиваются с любой новой технологией — примера создания приложения “Hello World”.

Приложение “Hello World”

В Visual Studio 2012 имеется встроенная поддержка для создания проектов рабочих потоков, ориентированных на версии .NET Framework 3.*x* и 4.*x*. Открыв диалоговое окно New Project (Новый проект), можно увидеть список таких типов проектов (рис. 45.1).

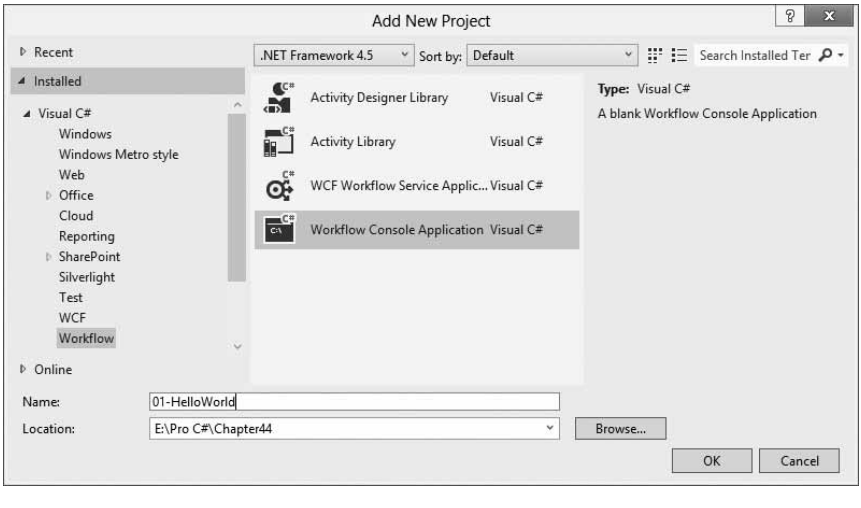

Рис. 45.1. *Список проектов для создания рабочих потоков в Visual Studio*

Удостоверьтесь, что в списке выбрана версия .NET Framework 4 или .NET Framework 4.5, и в поле со списком доступных шаблонов выберите вариант Workflow Console Application (Консольное приложение рабочего потока). Это приведет к созданию простого консольного приложения с шаблоном рабочего потока и главной программой для его выполнения.

Перетащите из панели инструментов на поверхность проектирования действие `WriteLine`, чтобы получить рабочий поток, подобный показанному на рис. 45.2. Действие `WriteLine` находится в категории Primitives (Примитивы) панели инструментов.

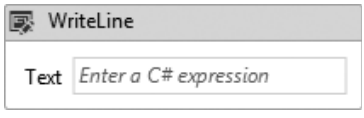

Рис. 45.2. *Добавление действия* `WriteLine`

Действие `WriteLine` включает в себя свойство `Text`, которое можно настраивать либо непосредственно на поверхности проектирования, просто вводя нужный текст в соответствующей строке, либо отобразить таблицу свойств. Позже в разделе "Специальные действия" будет показано, как определять специальные действия так, чтобы в них использовалось аналогичное поведение.

Свойство `Text` представляет собой не просто строку — на самом деле оно является видом аргумента, в котором в качестве источника может использоваться выражение. Выражения вычисляются во время выполнения для получения результата, после чего полученный подобным образом текстовый результат используется в качестве входных данных для действия `WriteLine`. Простое текстовое выражение должно быть заключено в двойные кавычки, т.е. в рассматриваемом примере для свойства `Text` потребуется ввести выражение `"Hello World"`. Если опустить кавычки, при компиляции возникнет ошибка, поскольку без кавычек это выражение не является допустимым. В версии 4.5 для проектов C# используются выражения C#, в то время как в версии 4.0 в редакторе выражений применялся синтаксис VB, что создавало определенную путаницу.

После построения и запуска данной программы текст будет выведен на консоль. При выполнении программы в методе `Main()` создается экземпляр рабочего потока, который затем выполняется с помощью статического метода класса `WorkflowInvoker`. Код этого примера доступен в проекте `01_HelloWorld`.

Класс `WorkflowInvoker` позволяет вызывать рабочий поток синхронным образом. Вдобавок доступно два других метода для выполнения рабочего потока, которые позволяют вызывать его асинхронно; они рассматриваются в разделе "Класс `WorkflowApplication`" далее в главе. Синхронное выполнение было возможно и в WF 3.*x*, однако оно труднее в реализации и влечет за собой более высокие накладные расходы.

Синхронная природа `WorkflowInvoker` делает его идеальным вариантом для выполнения краткосрочных рабочих потоков, которые являются реакцией на производимое в пользовательском интерфейсе действие, например, включение или отключение элементов пользовательского интерфейса.

Действия

Все компоненты рабочего потока являются действиями, в том числе и сам поток. В действительности термин *рабочий поток* — это синоним коллекции действий. В версии 4.*x* не существует никакого фактического класса `Workflow` (как это было в версии 3.*x*). Любое действие представляет собой просто класс, который в конечном итоге является производным от класса `Activity`.

Иерархия классов в WF 4 немного сложнее, чем в WF 3.*x*; основные классы иерархии показаны на рис. 45.3.

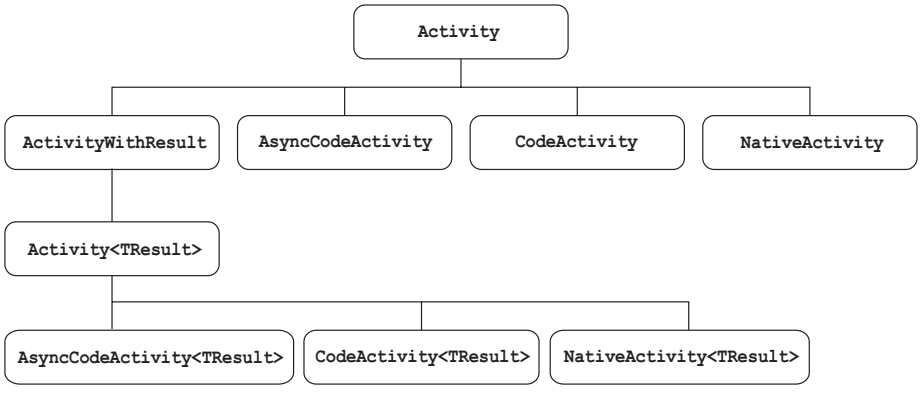

Рис. 45.3. *Иерархия классов WF 4*

Класс `Activity` является корневым классом для всех действий рабочих потоков. От него обычно наследуются классы специальных действий, занимающие второй уровень. Для создания простого специального действия, вроде упоминавшегося ранее действия `WriteLine`, наследование должно производиться от класса `CodeActivity`, потому что этот класс обладает возможностями, достаточными для функционирования специального клона `WriteLine`. Действия, которые должны выполняться и возвращать какой-то результат, должны быть унаследованы от класса `ActivityWithResult`. В этом случае предпочтительнее использовать обобщенный класс `Activity<TResult>`, поскольку он предоставляет типизированное свойство `Result`.

При создании специальных действий сложнее всего выбрать производный базовый класс, поэтому в главе будут приведены примеры, которые помогут принять такое решение. Чтобы действие выполняло какие-то операции, обычно переопределяется метод `Execute()`, сигнатура которого зависит от выбранного базового класса, как показано в табл. 45.1.

Таблица 45.1. Метод `Execute()` в разных базовых классах

Базовый класс	Метод `Execute()`
`AsyncCodeActivity`	`IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object)`
	`void EndExecute(AsyncCodeActivityContext, IAsyncResult)`
`CodeActivity`	`void Execute(CodeActivityContext)`
`NativeActivity`	`void Execute(NativeActivityContext)`
`AsyncCodeActivity<TResult>`	`IAsyncResult BeginExecute(AsyncCodeActivityContext, AsyncCallback, object)`
	`TResult EndExecute(AsyncCodeActivityContext, IAsyncResult)`
`CodeActivity<TResult>`	`TResult Execute(CodeActivityContext)`
`NativeActivity<TResult>`	`void Execute(NativeActivityContext)`

Нетрудно заметить, что параметры, передаваемые методу `Execute()`, отличаются используемым контекстом выполнения. В WF 3.*x* применялся только один класс контекста (`ActivityExecutionContext`), а в WF 4.*x* для разных классов действий используются разные классы контекстов.

Главное отличие состоит в том, что класс `CodeActivityContext` (и производный от него класс `AsyncCodeActivityContext`) обладает ограниченными функциональными возможностями по сравнению с классом `NativeActivityContext`. Это означает, что действия, унаследованные от классов `CodeActivity` и `AsyncCodeActivity`, могут выполнять гораздо меньше операций со своим контейнером. Например, показанное ранее действие `WriteLine` должно лишь выводить строку текста на консоль. Следовательно, возможность доступа к исполняющей среде ему совершенно не нужна. Более сложному действию, однако, может понадобиться планирование своих дочерних действий или взаимодействие с другими системами; в таком случае ему необходима возможность доступа ко всей исполняющей среде, и оно должно быть унаследованным от класса `NativeActivity`. Мы еще вернемся к этой теме при рассмотрении способов создания собственных специальных действий.

В WF 4 предоставляется множество стандартных действий, и в последующих разделах приведены примеры некоторых из них вместе со сценариями использования. Тремя главными сборками со стандартными действиями, которые применяются в WF 4.*x*, являются `System.Activities.dll`, `System.Activities.Core.Presentation.dll` и `System.Activities.Presentation.dll`.

Действие `If`

Действие `If` работает подобно оператору `if-else` в C#. После перетаскивания действия `If` на поверхность конструктора оно приобретает вид, показанный на рис. 45.4. Действие `If` является составным и содержит два заполнителя для дочерних действий — один для части `Then` и еще один для `Else`.

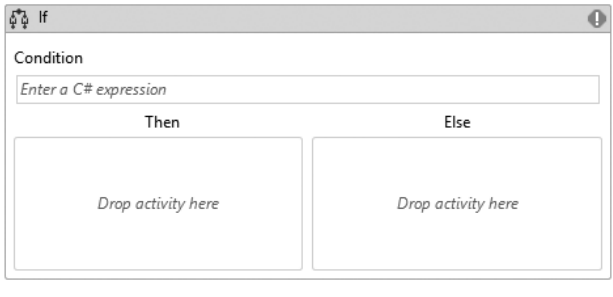

Рис. 45.4. *Действие* `If` *на поверхности конструктора*

Действие `If` (см. рис. 45.4) также включает глиф, указывающий на то, что действие содержит ошибку проверки достоверности; в этом случае необходимо определить свойство `Condition`. Условие, заданное в этом свойстве, будет вычисляться при выполнении действия. Если результатом является `True`, выполняется ветвь `Then`, а если `False` — ветвь `Else`.

Свойство `Condition` принимает выражение, возвращающее булевское значение, так что в нем можно вводить любое выражение, которое отвечает этому требованию.

В выражении можно ссылаться на любые определяемые в рабочем потоке переменные, а также обращаться ко многим статическим классам, которые доступны в .NET Framework. Это значит, что можно, например, определять выражение на основе значения `Environment.Is64BitOperatingSystem`, если оно является критически важным для какой-то части рабочего потока. Естественно, что можно также определять аргументы, которые должны передаваться рабочему потоку и которые могут затем вычисляться в рамках выражения внутри действия `If`. Аргументы и переменные рассматриваются в разделе “Аргументы и переменные”.

Действие InvokeMethod

Это действие является одним из самых полезных в наборе стандартных действий, поскольку позволяет выполнять существующий код и, по сути, помещать его внутрь семантики рабочего потока. Наличие большого объема существующего кода является типичной ситуацией, а данное действие позволяет легко напрямую вызывать этот код из рабочего потока.

Существуют два способа использования действия InvokeMethod. Выбор конкретного из них зависит от того, какой метод должен вызываться — статический или метод экземпляра. В случае вызова статического метода необходимо определить параметры TargetType и MethodName. В случае вызова метода экземпляра можно воспользоваться свойствами TargetObject и MethodName. В такой ситуации экземпляр TargetObject можно создать внутри либо определить с помощью переменной где-то в рабочем потоке. Примеры обоих способов применения действия InvokeMethod можно найти в демонстрационном проекте 02_ParallelExecution.

Если вызываемому методу необходимо передать аргументы, их можно определить с помощью коллекции Parameters. Порядок параметров в этой коллекции должен совпадать с тем, в котором они должны передаваться методу. Кроме того, доступно свойство Result, которому присваивается значение, возвращаемое в результате вызова метода. Это свойство можно привязать к переменной внутри рабочего потока, чтобы использовать его значение надлежащим образом.

Действие Parallel

Действие Parallel имеет не совсем подходящее имя. На первый взгляд может показаться, что оно планирует выполнение своих дочерних действий по-настоящему параллельно на компьютерах с несколькими процессорами, но на самом деле это не так, за исключением некоторых особых случаев.

После перетаскивания действия Parallel на поверхность конструктора можно перетащить в него подчиненные действия, как показано на рис. 45.5. Дочерние действия могут быть одиночными действиями, как на рис. 45.5, либо представлять собой составные действия, подобные Sequence или еще одному действию Parallel.

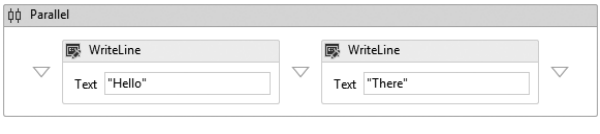

Рис. 45.5. *Действия* Parallel *и его подчиненные действия на поверхности конструктора*

Во время выполнения действие Parallel планирует запуск каждого следующего за ним дочернего действия. Механизм времени выполнения планирует эти дочерние действия согласно принципу “первым пришел — первым обслужен” (first in, first out — FIFO), создавая иллюзию параллельного выполнения. Тем не менее, все дочерние действия выполняются в единственном потоке.

Чтобы обеспечить настоящее параллельное выполнение, действия, перетаскиваемые в Parallel, должны быть унаследованы от класса AsyncCodeActivity. В демонстрационном проекте 02_ParallelExecution приведен пример асинхронной обработки кода в двух ветвях действия Parallel. На рис. 45.6 показано применение двух действий InvokeMethod в действии Parallel.

В используемых здесь действияъ InvokeMethod вызываются два простых метода DoJob1() и DoJob2(), которые обеспечивают засыпание, соответственно, на две и три секунды. Чтобы эти методы выполнялись асинхронно, необходимо внести еще одно изменение.

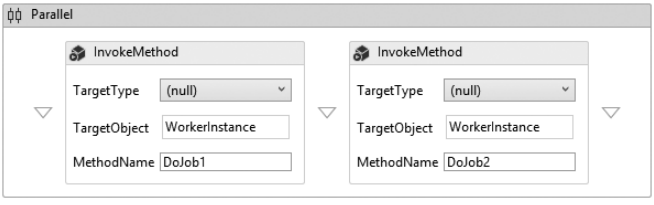

Рис. 45.6. *Использование двух действий* `InvokeMethod` *внутри* `Parallel`

Действие `InvokeMethod` имеет булевское свойство `RunAsynchronously`, стандартным значением которого является `False`. Если оно установлено в `True`, то целевой метод вызывается асинхронно, тем самым делая возможным одновременное выполнение более чем одного действия. На компьютере с одним процессором могут использоваться два потока, создавая иллюзию одновременного выполнения, а в многопроцессорной системе эти потоки смогут выполняться в разных ядрах, обеспечивая действительно параллельное выполнение. При создании собственных действий лучше делать их асинхронными, поскольку тогда конечный пользователь получает преимущества параллельного выполнения.

Действие `Delay`

В бизнес-процессах часто требуется организовать ожидание в течение определенного периода времени перед их завершением. Рассмотрим, например, использование рабочего потока для утверждения расходов. В рамках этого рабочего потока может производиться отправка электронного сообщения непосредственному руководителю с просьбой одобрить заявку на расходы. Затем рабочий поток может быть переведен в состояние ожидания до тех пор, пока не будет получено одобрение (или отказ). Кроме того, целесообразно определить период тайм-аута, чтобы при отсутствии ответа в течение, скажем, одного дня, заявка на расходы перенаправлялась следующему руководителю в цепочке управления.

С помощью действия `Delay` можно реализовать одну часть этого сценария (другую часть будет решать действие `Pick`, рассматриваемое в следующем разделе). Задача этого действия — обеспечение ожидания на протяжении определенного времени перед продолжением выполнения потока.

Действие `Delay` содержит свойство `Duration`, для которого может быть установлено дискретное значение `TimeSpan`, но поскольку это значение определено как выражение, его можно связать с какой-то переменной внутри рабочего потока либо вычислять на основе других значений.

Во время выполнения рабочий поток входит в состояние простоя `Idle`, в котором запускает действие `Delay`. Простаивающие потоки являются кандидатами на постоянное хранение — в этом случае данные экземпляра рабочего потока могут быть записаны в постоянное хранилище (например, базу данных SQL Server), а сам поток выгружен из памяти. Такое поведение позволяет экономить ресурсы, поскольку в любой конкретный момент времени в памяти находятся только выполняющиеся потоки. Любые рабочие потоки, выполнение которых откладывается, сохраняются на диске.

Действие `Pick`

Обеспечение ожидания одного из ряда возможных событий, с помощью, например, метода `WaitAny()` класса `WaitHandle` из пространства имен `System.Threading` является типичной схемой в программировании. Действие `Pick` представляет собой способ, которым это делается в рабочем потоке, поскольку в нем можно определить любое количество ветвей и в каждой из них обеспечить ожидание срабатывания какого-то триггера. После срабатывания триггера могут выполняться остальные действия, содержащиеся в ветви.

Чтобы продемонстрировать все на конкретном примере, обратимся к описанному в предыдущем разделе процессу одобрения заявок на расходы. Здесь понадобится действие `Pick` с тремя ветвями: первой — для обработки случаев принятия заявок, второй — для обработки случаев отклонения заявок, и третьей — для обработки случаев срабатывания тайм-аута.

Код этого примера находится в демонстрационном проекте `03_PickDemo`. В этом коде определен рабочий поток, который состоит из действия `Pick` и трех ветвей. При его выполнении появляется приглашение принять или отклонить заявку. По истечении 10 или более секунд приглашение закрывается и запускается действие из ветви `Delay`.

В рассматриваемом примере первым в рабочем потоке используется действие `DisplayPrompt`. Это действие вызывает метод, определенный в интерфейсе, для отображения руководителю приглашения принять или отклонить заявку. Поскольку эта функциональность определяется в интерфейсе, приглашение может иметь вид сообщения электронной почты, мгновенного сообщения или любого другого уведомления, информирующего руководителя о необходимости обработать заявку на расходы. Затем в рабочем потоке выполняется действие `Pick`, которое предусматривает ожидание получения входных данных из внешнего интерфейса (принятие либо отклонение заявки), а также ожидание в течение заданного периода.

При выполнении `Pick` первое действие в каждой ветви переводится в состояние ожидания. При срабатывании одного из событий все остальные события ожидания отменяются, и начинается обработка остальной части кода в ветви, в которой было сгенерировано событие. То есть в случае одобрения заявки на расходы происходит завершение действия `WaitForAccept` и начинает выполняться следующее действие, которое отображает сообщение о подтверждении заявки. В случае отклонения заявки завершается действие `WaitForReject` и производится вывод сообщения об отклонении заявки.

Если ни `WaitForAccept`, ни `WaitForReject` не завершены, действие `WaitForTimeout` в конечном счете завершается по истечении указанного периода отсрочки. После этого заявка на расходы может быть далее передана другому руководителю, возможно, с использованием контактной информации из Active Directory. В этом примере при выполнении действия `DisplayPrompt` пользователю отображается диалоговое окно. Это значит, что по истечении периода отсрочки данное окно понадобится закрыть, для чего служит действие по имени `ClosePrompt`, как показано на рис. 45.7.

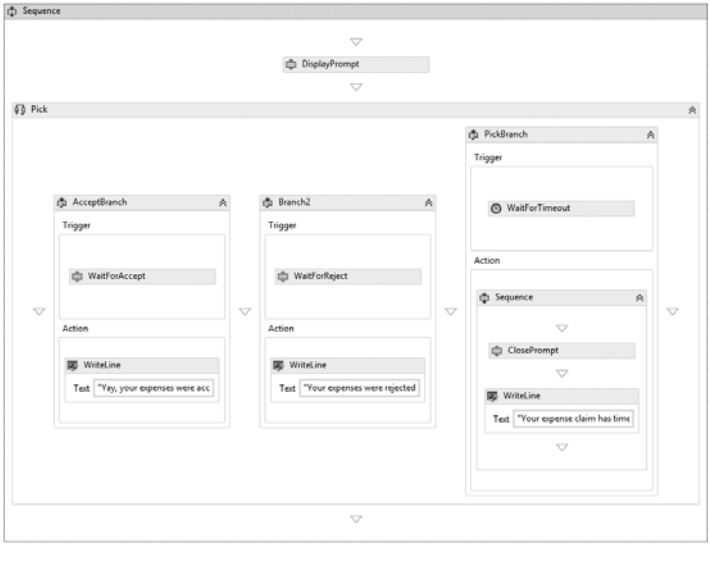

Рис. 45.7. *Пример использования в рабочем потоке действия* `Pick` *с тремя ветвями*

Некоторые концепции, использованные в этом примере, такие как создание специальных действий или ожидание внешних событий, еще не рассматривались. Все они раскрываются в разделе "Специальные действия".

Специальные действия

До сих пор применялись действия, определенные внутри пространства имен `System.Activities`. В этом разделе будет показано, как создавать специальные действия и расширять их возможности для повышения удобства использования на этапе проектирования и во время выполнения.

Для начала создадим действие `DebugWrite`, которое может применяться для вывода строки текста в окне консоли в отладочных конфигурациях. Хотя рассматриваемый здесь пример тривиален, он будет расширен для демонстрации всего спектра возможностей, которые доступны для специальных действий. При создании специальных действий можно просто сконструировать класс внутри проекта рабочего потока. Однако более предпочтительный подход предполагает создание специальных действий в отдельной сборке, поскольку тогда действия будут доступны для многократного использования. По этой причине для конструирования специальных действий лучше создавать проект библиотеки классов. Код этого примера доступен в проекте `04_CustomActivities`.

Любое простое действие, такое как `DebugWrite`, должно быть унаследовано непосредственно от класса `CodeActivity`. В следующем коде создается класс действия и определяется свойство `Message`, которое должно отображаться при выполнении этого действия (файл `04_CustomActivities\Activities\DebugWrite.cs`):

```
using System;
using System.Activities;
using System.Diagnostics;
namespace Activities
{
   public class DebugWrite : CodeActivity
   {
      [Description("The message output to the debug stream")]
      public InArgument<string> Message { get; set; }
      protected override void Execute(CodeActivityContext context)
      {
         Debug.WriteLine(Message.Get(context));
      }
   }
}
```

При подготовке класса `CodeActivity` к выполнению вызывается его метод `Execute()` и именно здесь действие фактически должно что-нибудь делать.

В примере определено свойство `Message`, которое выглядит подобно обычному свойству .NET, однако применяется внутри метода `Execute()` несколько необычно. Одним из многочисленных изменений, которые были внесены в WF 4, является место сохранения данных состояния. В WF 3.*x* было принято использовать стандартные свойства .NET и сохранять данные действия внутри самого действия. Проблема такого подхода заключалась в том, что хранилище являлось непрозрачным для исполняющей среды рабочих потоков, поэтому требовалось выполнять двоичное сохранение всех созданных действий, обеспечивая возможность точного восстановления данных.

В WF 4 все данные сохраняются за пределами отдельных действий, поэтому чтобы получить значение аргумента, необходимо запросить контекст этого значения, а чтобы установить значение аргумента, нужно предоставить новое значение контекста. В этом случае исполняющая среда рабочих потоков может отслеживать изменения в состоянии, т.к. поток выполняется и потенциально сохраняет только те изменения, которые произошли в промежутке между точками сохранения, а не все данные рабочего потока.

Атрибут `[Description]`, определенный в свойстве `Message`, используется внутри таблицы свойств в Visual Studio для предоставления дополнительной информации о свойстве (рис. 45.8).

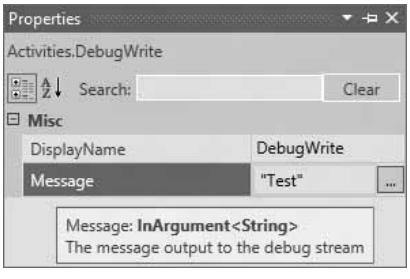

Рис. 45.8. *Отображение дополнительной информации о свойстве*

В таком виде действие уже пригодно к применению; тем не менее, есть еще несколько проблем, которые должны быть решены для того, чтобы сделать действие более дружественным к пользователю. Как было показано на примере действия `Pick` ранее в главе, действие может иметь обязательные свойства, в случае не определения которых в области конструктора появляется глиф с сообщением об ошибке. Чтобы добиться такого же поведения от собственного действия, код нужно расширить.

Проверка достоверности действий

Когда действие помещено на поверхность проектирования, визуальный конструктор ищет информацию проверке достоверности в двух местах. Простейшая форма проверки достоверности заключается в добавлении атрибута `[RequiredArgument]` к свойству аргумента. Если этот аргумент не определен, справа от имени действия будет появляться глиф с изображением восклицательного знака, как показано на рис. 45.9.

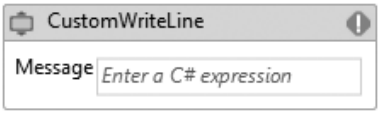

Рис. 45.9. *Глиф, отображаемый в случае, если обязательный аргумент не определен*

При наведении курсора мыши на этот глиф будет отображаться всплывающая подсказка с сообщением “Value for a required activity argument ‘Message’ was not supplied” (“Не было предоставлено значение для обязательного аргумента `Message` данного действия”). Это ошибка компиляции, следовательно, для указанного атрибута понадобится определить значение.

В случае если несколько свойств связаны друг с другом, можно переопределить метод `CacheMetadata()`, добавив дополнительный код для проверки достоверности. Указанный метод вызывается перед началом выполнения действия, поэтому в нем можно проверить, определены ли значения для всех обязательных аргументов и при желании добавить метаданные в передаваемый аргумент. Можно также добавить дополнительные сообщения (или предупреждения), касающиеся проверки, за счет вызова одной из переопределенных версий метода `AddValidationError()` объекта `CodeActivityMetadata`, который передан методу `CacheMetadata()`.

После завершения необходимой проверки достоверности можно изменить поведение визуализации для действия, которое в текущий момент обеспечивается конструктором, сделав его более интересным.

Визуальные конструкторы

Когда действие визуализируется на экране, с ним обычно ассоциирован визуальный конструктор. Задачей такого конструктора является обеспечение экранного представления этого действия, и в WF это делается с помощью XAML. Язык XAML описан в главе 35.

Интерфейс времени проектирования для действия обычно создается в сборке, отдельной от самого действия, т.к. во время выполнения необходимость в ней отпадает. В Visual Studio доступен шаблон проекта Activity Designer Library (Библиотека конструктора действия), который является идеальной отправной точкой, поскольку он предоставляет для действия стандартный визуальный конструктор, который затем можно изменять по своему усмотрению.

В коде XAML для конструктора можно определять все, что угодно, в том числе анимационные эффекты. Для пользовательских интерфейсов обычно требуются более скромные возможности, поэтому вначале стоит взглянуть на уже существующие действия, чтобы понять, что еще может потребоваться. Прежде всего, создадим простой конструктор и свяжем его с действием `DebugWrite`.

Ниже показан шаблон (файл `04_CustomActivities\Activities.Design\DebugWriteDesigner.xaml`), создаваемый автоматически при добавлении в проект конструктора действия (или создании нового проекта типа Activity Designer Library).

```
<sap:ActivityDesigner x:Class="Activities."
   <Presentation.DebugWriteDesigner
   xmlns="http:=""//schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:sap="clr-namespace:System.Activities.Presentation;
              assembly=System.Activities.Presentation"
   xmlns:sapv="clr-namespace:System.Activities.Presentation.View;
              assembly=System.Activities.Presentation">
  <Grid>
  </Grid>
</sap:ActivityDesigner>
```

Созданный XAML-код просто конструирует сетку, а также импортирует ряд важных пространств имен, которые могут требоваться действию. Очевидно, что содержимого в шаблоне очень мало, поэтому для начала добавим к нему элементы управления надписи (`TextBlock`) и текстового поля (`TextBox`), которые будут использоваться для определения сообщения:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <TextBlock Text="Message" Margin="0,0,5,0"/>
  <TextBox Text="{Binding Path=ModelItem.Message, Mode=TwoWay}"
    Grid.Column="1"/>
</Grid>
```

В показанном здесь XAML-коде создается привязка между свойством `Message` действия и элементом управления `TextBox`. Внутри XAML-кода конструктора можно всегда ссылаться на проектируемое действие посредством ссылки `ModelItem`.

Чтобы ассоциировать определенный выше конструктор с действием `DebugWrite`, понадобится изменить действие, добавив к нему атрибут `Designer` (можно также реализовать интерфейс `IRegisterMetadata`, но этот подход в главе не рассматривается):

```
[Designer("Activities.Presentation.DebugWriteDesigner, Activities.
Presentation")]
public class DebugWrite : CodeActivity
{
   ...
}
```

Здесь атрибут [Designer] применяется для определения связи между конструктором и действием. Целесообразно использовать строковую версию этого атрибута, поскольку при этом полностью исключается вероятность попадания ссылки на сборку конструктора в сборку действия.

Теперь при использовании экземпляра действия DebugWrite в Visual Studio оно будет визуализироваться примерно так, как показано на рис. 45.10.

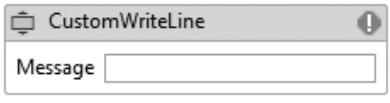

***Рис. 45.10**. Визуализация действия `DebugWrite` в Visual Studio*

Однако в таком случае возникает одна проблема, связанная со свойством Message, поскольку в нем не отображается значение, определенное в таблице свойств, а попытка установить это значение, вводя его в текстовом поле, закачивается генерацией исключения. Причина в том, что в этом случае вы пытаетесь привязать простое текстовое значение к типу InArgument<string>, а для подобной операции должна использоваться другая пара встроенных классов WF — ExpressionTextBox и ArgumentToExpressionConverter. Весь необходимый XAML-код для визуального конструктора теперь выглядит следующим образом (добавленные и измененные строки выделены полужирным):

```
<sap:ActivityDesigner x:Class="Activities.Presentation.DebugWriteDesigner"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sap="clr-namespace:System.Activities.Presentation;
     assembly=System.Activities.Presentation"
  xmlns:sapv="clr-namespace:System.Activities.Presentation.View;
     assembly=System.Activities.Presentation"
  xmlns:sadc="clr-namespace:System.Activities.Presentation.Converters;
     assembly=System.Activities.Presentation">
  <sap:ActivityDesigner.Resources>
    <sadc:ArgumentToExpressionConverter x:Key="argConverter"/>
  </sap:ActivityDesigner.Resources>
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <TextBlock Text="Message" Margin="0,0,5,0" />
    <sapv:ExpressionTextBox Grid.Column="1"
      Expression="{Binding Path=ModelItem.Message, Mode=TwoWay,
                  Converter={StaticResource argConverter},
                  ConverterParameter=In}"
                  OwnerActivity="{Binding ModelItem}"/>
  </Grid>
</sap:ActivityDesigner>
```

В файл включено новое пространство имен System.Activities.Presentation.View, а вместе с ним класс ArgumentToExpressionConverter, который служит для преобразования между выражением, отображаемым на экране, и аргументом, содержащимся в свойстве Message действия, которое добавлено в раздел ресурсов файла XAML.

Стандартный элемент управления TextBox заменен элементом управления ExpressionTextBox. Этот элемент управления позволяет пользователю вводить выражения и простой текст, чтобы действие DebugWrite могло включать выражение, объединяющее в себе множество значений из выполняющегося рабочего потока, а не только простую текстовую строку. Со всеми этими изменениями поведение визуализации действия теперь стало гораздо больше похоже на поведение встроенных действий.

При модернизации решения от .NET 4 до .NET 4.5 может удивить появление текста *Enter a VB Expression* (Введите выражение VB) во всех текстовых полях действий существующих рабочих потоков и текста *Enter a C# Expression* (Введите выражение C#) во всех создаваемых рабочих потоках. Дело в том, что WF 4.5 содержит атрибут, определенный в XAML-коде, при отсутствии которого отображается приглашение на ввод выражения VB. Код XAML нового рабочего потока содержит следующую директиву:

```
sap:2010:ExpressionActivityEditor.ExpressionActivityEditor="C#"
```

Если она опущена, отображается напоминание о необходимости ввода выражения VB. На то, чтобы разобраться с этим недостатком WF 4.5, потребовалась масса времени.

Специальные составные действия

Необходимость в создании составных действий, т.е. действий, содержащих другие дочерние действия, возникает очень часто. Примерами таких действий являются `Pick` и `Parallel`, которые рассматривались ранее в этой главе. Разработчик может настраивать схему выполнения составного действия под конкретные нужды: например, составное действие может выполнять только одно из своих дочерних действий или пропускать какие-то дочерние действия в зависимости от текущего дня недели. Простейшей схемой является выполнение всех дочерних действий, но даже в таком случае разработчик может решать, как должны выполняться эти дочерние действия, и когда должно завершаться выполнение всего составного действия.

В следующем примере создается действие "повторной попытки". Довольно часто необходимо пробовать выполнить операцию, и в случае неудачи повторять попытку еще несколько раз, прежде чем произойдет полный отказ от выполнения операции. Псевдокод для такого действия выглядит следующим образом:

```
int iterationCount = 0;
bool looping = true;
while ( looping )
{
   try
   {
      // Выполнение действия.
      looping = false;
   }
   catch (Exception ex)
   {
      iterationCount += 1;
      if ( iterationCount >= maxRetries )
         rethrow;
   }
}
```

Нужно просто скопировать приведенный код в качестве действия и вместо комментария вставить конкретное действие, которое должно выполняться. Может показаться, что для этого удобно использовать метод `Execute()` специального действия. Однако есть и другой путь: с помощью других действий можно сделать намного больше. В данном случае необходимо создать специальное действие, содержащее "заглушку", в которую конечный пользователь сможет помещать действие для повторной попытки, а также свойство для хранения максимального количества повторных попыток. Ниже приведен код (файл `04_CustomActivities\Activities\Retry.cs`):

```
public class Retry : Activity
{
   public Activity Body { get; set; }
   [RequiredArgument]
   public InArgument<int> NumberOfRetries { get; set; }
```

```
    public Retry()
    {
      Variable<int> iterationCount =
        new Variable<int> ( "iterationCount", 0 );
      Variable<bool> looping = new Variable<bool> ( "looping", true );
      this.Implementation = () =>
      {
        return new While
        {
          Variables = { iterationCount, looping },
          Condition = new VariableValue<bool> { Variable = looping },
          Body = new TryCatch
          {
            Try = new Sequence
            {
              Activities =
              {
                this.Body,
                new Assign
                {
                  To = new OutArgument<bool> ( looping ),
                  Value = new InArgument<bool> { Expression = false }
                }
              }
            },
            Catches =
            {
              new Catch<Exception>
              {
                Action = new ActivityAction<Exception>
                {
                  Handler = new Sequence
                  {
                    Activities =
                    {
                      new Assign
                      {
                        To = new OutArgument<int>(iterationCount),
                        Value = new InArgument<int>
                          (ctx => iterationCount.Get(ctx) + 1)
                      },
                      new If
                      {
                        Condition = new InArgument<bool>
                          (env=>iterationCount.Get(env) >=
                            NumberOfRetries.Get(env)),
                        Then = new Rethrow()
                      }
                    }
                  }
                }
              }
            }
          }
        };
      };
    }
}
```

Здесь сначала определяется свойство `Body` типа `Activity`, которое представляет собой действие для выполнения в цикле. Затем определяется свойство `RetryCount`, позволяющее указать количество попыток выполнения операции.

Это специальное действие унаследовано непосредственно от класса `Activity` и реализуется в виде функции. При выполнении рабочего потока, содержащего такое действие, будет выполняться функция, соответствующая приведенному ранее псевдокоду. В конструкторе создаются локальные переменные, используемые действием, и затем согласно псевдокоду строится набор действий. Код этого примера также доступен в решении `04_CustomActivities`.

Нетрудно догадаться, что рабочие потоки можно также создавать и без XAML-кода, но средства визуализации на этапе проектирования будут недоступны (т.е. перетаскивать действия для генерации кода не получится). Однако в случае, когда написание кода предпочтительнее, нет никаких причин не использовать его вместо XAML.

После создания специального составного действия для него нужно определить визуальный конструктор. Здесь необходимо действие, имеющее заполнитель, на место которого можно было бы перетаскивать другое действие. Если посмотреть на другие стандартные действия, легко обнаружить среди них несколько таких, которые обладают подобным поведением, как, например, `If` и `Pick`. В идеале специальное действие должно работать подобно встроенным действиям, так что имеет смысл взглянуть на их реализации.

Просмотрев в утилите Reflector библиотеки рабочих потоков, можно удостовериться в отсутствии какого-либо XAML-кода визуальных конструкторов. Причина в том, что этот код скомпилирован в сборках как набор ресурсов. Утилита Reflector позволяет видеть эти ресурсы, а ее текущая версия (v7.5) — еще и декомпилировать ресурсы BAML, чтобы их можно было прочитать.

Открыв Reflector, загрузите в него сборку `System.Activities.Presentation`, перейдите к узлу `Resources` в древовидном представлении и откройте ветвь `System.Activites.Presentation.g.resources`. Отобразится список всех ресурсов BAML текущей сборки, в котором можно выбрать образец и просмотреть код XAML.

Получив представление о том, как выглядит код XAML во встроенных действиях, можно создать код для обеспечения визуализации действия `Retry` так, как показано на рис. 45.11.

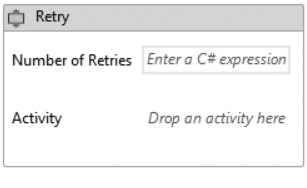

Рис. 45.11. *В*изуализация действия `Retry`

Ключевую роль в визуализации этого действия играет класс `WorkflowItemPresenter`; именно он в XAML-коде отвечает за отображение заполнителя для добавления дочернего действия. Его определение выглядит следующим образом:

```
<sap:WorkflowItemPresenter IsDefaultContainer="True"
     AllowedItemType="{x:Type sa:Activity}"
     HintText="Drop an activity here" MinWidth="100" MinHeight="60"
     Item="{Binding Path=ModelItem.Body, Mode=TwoWay}"
     Grid.Column="1" Grid.Row="1" Margin="2">
```

Этот элемент управления привязан к свойству `Body` действия `Retry`. Элемент имеет свойство `HintText`, определяющее вспомогательный текст, который должен отображаться, когда еще не добавлено никакого дочернего действия. Также в приведенном XAML-коде определяются стили для отображения визуального конструктора в развернут-

той и свернутой версии, что обеспечивает действию поведение, характерное для встроенных действий. Весь код вместе с XAML-кодом этого примера доступен в решении `04_CustomActivities`.

Рабочие потоки

До настоящего момента в этой главе внимание было сосредоточено на действиях, но не о рабочих потоках. *Рабочий поток* — это просто список действий, и на самом деле рабочий поток является просто еще одним типом действия. Применение такой модели упрощает механизм исполняющей среды, поскольку механизм должен знать, как выполнять объекты только одного типа — производного от класса `Activity`.

Ранее был показан класс `WorkflowExecutor`, который может выполнять рабочий поток синхронно, и при этом упоминалось, что он является только одним способом выполнения рабочего потока. Существуют три варианта выполнения рабочих потоков, каждый из которых обладает своими возможностями. Прежде чем переходить к изучению других способов выполнения рабочих потоков, необходимо подробнее ознакомиться с аргументами и переменными.

Аргументы и переменные

Рабочий поток можно считать программой, а одним из аспектов любого языка программирования является возможность создания переменных и передачи аргументов в и из программы. Разумеется, в WF поддерживаются обе конструкции, и в настоящем разделе показано, как определять аргументы и переменные.

Предположим, что рабочий поток обрабатывает страховой полис, следовательно, передаваемым аргументом должен быть номер страхового полиса. Для определения аргумента рабочего потока необходимо перейти в окно визуального конструктора и щелкнуть на кнопке Arguments (Аргументы) в его левой нижней части. Это приводит к отображению списка всех определенных для рабочего потока аргументов, показанного на рис. 45.12, в котором также можно создавать собственные аргументы.

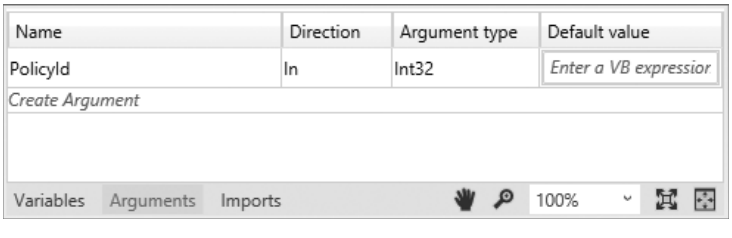

Рис. 45.12. *Список аргументов рабочего потока*

Для определения аргумента необходимо указать его имя (в столбце Name), направление (в столбце Direction), которым может быть `In`, `Out` и `InOut`, а также тип данных (в столбце Argument type). Дополнительно можно задать стандартное значение, которое будет использоваться, если аргумент не предоставлен.

Направление служит для определения того, каким должен быть аргумент: входным, выходным либо тем и другим.

В первом разделе настоящей главы было показано, как применять для выполнения рабочего потока класс `WorkflowInvoker`. У метода `Invoke()` имеется несколько переопределенных версий, позволяющих передавать аргументы в рабочий поток. Аргументы передаются в виде словаря пар "имя-значение", в которых имя должно точно соответствовать имени аргумента, учитывая регистр символов. В следующем коде рабочему потоку передается значение `PolicyId` (файл `05_ArgsAndVars\Program.cs`):

```
Dictionary<string, object> parms = new Dictionary<string, object>();
parms.Add("PolicyId", 123);
WorkflowInvoker.Invoke(new PolicyFlow(), parms);
```

В коде производится вызов рабочего потока с передачей в словаре значения `PolicyId` для именованного параметра. Если в словаре предоставляется имя, для которого аргумента не существует, генерируется исключение `ArgumentException`. Если же значение для аргумента `In` не указано, исключение генерироваться не будет. Такое поведение не подходит, поскольку нужно, чтобы исключение генерировалось для любых аргументов `In`, которые не были определены, и чтобы оно не возникало в случае передачи большего количества аргументов.

После завершения рабочего потока может понадобиться извлечь выходные аргументы. Для этого предусмотрена специальная переопределенная версия метода `WorkflowInvoker.Invoke()`, которая возвращает словарь. Этот словарь содержит только аргументы `Out` или `InOut`.

Внутри рабочего потока можно определять переменные. Делать это в рабочих потоках XAML из WF 3.*x* было непросто, однако в WF 4 сложности были устранены и теперь параметры можно легко определять в коде XAML.

Как и в любом языке программирования, с переменными рабочих потоков связано понятие области действия. Переменные могут определяться как "глобальные" за счет их определения в корневом действии рабочего потока. Такие переменные доступны для всех действий внутри рабочего потока, и время их существования зависит от времени жизни самого рабочего потока.

Переменные также могут определяться и внутри отдельных действий. В этом случае они доступны только для действий, в которых определены, а также для их дочерних действий. По завершении действия его переменные покидают область видимости и перестают быть доступными.

Класс `WorkflowApplication`

Хотя класс `WorkflowInvoker` удобен для синхронного выполнения рабочих потоков, иногда требуется создать длительные рабочие потоки, которые могли бы сохраняться в базе данных и затем возобновлять свое выполнение в какой-то момент в будущем. В этом случае используется класс `WorkflowApplication`.

Класс `WorkflowApplication` похож на класс `WorkflowRuntime` из WF 3 тем, что позволяет запускать рабочий поток и реагировать на события, возникающие в экземпляре этого рабочего потока. Ниже показан простейший код работы с классом `WorkflowApplication`:

```
WorkflowApplication app = new WorkflowApplication(new Workflow1());
ManualResetEvent finished = new ManualResetEvent(false);
app.Completed = (completedArgs) => { finished.Set(); };
app.Run();
finished.WaitOne();
```

В коде создается экземпляр рабочего потока, который подключается к делегату `Completed` для установки события ручного сброса (`ManualResetEvent`). Затем вызывается метод `Run()`, чтобы начать выполнение рабочего потока, и напоследок организуется ожидание срабатывания события.

Здесь проявляется одно из главных различий между классами `WorkflowExecutor` и `WorkflowApplication`: последний является асинхронным. При вызове метода `Run()` система будет использовать для выполнения рабочего потока пул потоков, а не вызывающий поток. Таким образом, необходима какая-то форма синхронизации, чтобы исключить возможность завершения работы приложения, обслуживающего рабочий поток, раньше завершения самого рабочего потока.

Типичный продолжительный рабочий поток может много раз находиться в спящем состоянии, из-за чего поведение большинства рабочих потоков характеризуется как периоды эпизодического выполнения. В начале рабочий поток обычно выполняет некоторую работу, затем ожидает поступления входных данных или истечения паузы. Получив входные данные, рабочий поток обрабатывает их и переходит в следующее состояние ожидания.

Таким образом, было бы идеально, если бы при переходе в спящее состояние рабочий поток выгружался из памяти и загружался туда обратно только при срабатывании события, требующего продолжения работы. Чтобы сделать это, в `WorkflowApplication` необходимо добавить объект `InstanceStore` и внести незначительные изменения в показанный ранее код. В инфраструктуре доступна только одна реализация абстрактного класса `InstanceStore` — `SqlWorkflowInstanceStore`.

Для использования этого класса понадобится база данных; соответствующие сценарии по умолчанию находятся в каталоге `Windows\Microsoft.NET\Framework64\v4.0.30319\SQL\en`. Обратите внимание, что номер версии может изменяться.

В этом каталоге можно обнаружить множество файлов SQL, из которых интересуют два: `SqlWorkflowInstanceStoreSchema.sql` и `SqlWorkflowInstanceStoreLogic.sql`. Их можно запустить применительно к существующей базы данных либо же создать совершшунно новую базу; при этом можно использовать как полную версию SQL Server, так версию SQL Express.

После подготовки базы данных в код необходимо внести изменения. Первым делом, следует создать экземпляр `SqlWorkflowInstanceStore` и добавить его в приложение рабочего потока (файл `06_WorkflowApplication\Program.cs`):

```
SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore
  (ConfigurationManager.ConnectionStrings["db"].ConnectionString);
AutoResetEvent finished = new AutoResetEvent(false);
WorkflowApplication app = new WorkflowApplication(new Workflow1());
app.Completed = (e) => { finished.Set(); };
app.PersistableIdle = (e) => { return PersistableIdleAction.Unload; };
app.InstanceStore = store;
app.Run();
finished.WaitOne();
```

Новые строки выделены полужирным. Также можно заметить, что в приложение рабочего потока к делегату `PersistableIdle` добавлен обработчик событий. Во время выполнения рабочий поток будет запускать столько действий, сколько сможет, до тех пор, пока больше не останется какой-либо работы. После этого он перейдет в состояние `Idle`, а любой находящийся в таком состоянии поток является кандидатом на сохранение. Делегат `PersistableIdle` определяет, что должно происходить с потоком в состоянии простоя. По умолчанию никакие операции бездействующим потоком не выполняются. Можно также указать обработчик `PersistableIdleAction.Persist`, который сохраняет копию рабочего потока в базе данных, оставляя рабочий поток в памяти, или обработчик `PersistableIdleAction.Unload`, который сохраняет копию рабочего потока в базе данных и выгружает его из памяти.

Запросить сохранение рабочего потока можно также посредством действия `Persist`, а при разработке специального действия, если оно унаследовано от `NativeActivity` — вызывая метод `RequestPersist()` класса `NativeActivityContext`.

Теперь возникает следующая проблема: имеется возможность выгружать рабочий поток из памяти и сохранять его в постоянном хранилище, но пока не описано, как извлечь поток из хранилища и снова инициировать его выполнение.

Закладки

Традиционно закладка используется для запоминания страницы в книге, чтобы потом иметь возможность продолжить чтение с места останова. В контексте рабочего потока закладка указывает место, с которого должно возобновляться выполнение этого рабочего

потока, и обычно закладки применяются тогда, когда ожидается поступление входных данных извне.

В качестве примера рассмотрим написание приложения для обработки данных по расценкам на услуги страхования. Конечный пользователь может создавать свои расценки в онлайновом режиме, с которыми ассоциируется рабочий поток. Эти расценки действительны на протяжении 30 дней, поэтому должна быть предусмотрена отмена расценок по истечении этого срока. Может также запрашиваться подтверждение снижения расценок, и если оно не поступит в течение указанного срока, страховой полис должен быть аннулирован. При таком сценарии рабочий поток должен иметь периоды выполнения и периоды перехода в спящее состояние, во время которых поток выгружается из памяти. Однако перед выгрузкой в потоке нужно указать место, с которого будет возобновлена обработка, для чего и применяются закладки.

Для определения закладки необходимо создать специальное действие, унаследованное от `NativeActivity` (файл `06_WorkflowApplication\CustomActivities\Task.cs`). Затем можно создать саму закладку внутри метода `Execute()` — тогда выполнение кода продолжится с места, определенного закладкой. В приведенном ниже примере определяется упрощенное действие `Task`, которое создает закладку и завершается при возобновлении выполнения с указанного в закладке места.

```
public class Task : NativeActivity<Boolean>
{
   [RequiredArgument]
   public InArgument<string> TaskName { get; set; }
   protected override bool CanInduceIdle
   {
      get { return true; }
   }
   protected override void Execute(NativeActivityContext context)
   {
      context.CreateBookmark(TaskName.Get(context),
         new BookmarkCallback(OnTaskComplete));
   }
   private void OnTaskComplete(NativeActivityContext context,
      Bookmark bookmark, object state)
   {
      bool taskOK = Convert.ToBoolean(state);
      this.Result.Set(context, taskOK);
   }
}
```

При вызове методу `CreateBookmark()` передается имя закладки и функция обратного вызова. Этот обратный вызов происходит при возобновлении выполнения кода с закладки. Обратному вызову передается произвольный объект — в рассматриваемом случае это объект типа `Boolean`, поскольку было решено, что каждая задача должна сообщать об успехе или неудаче, и затем эту информацию можно использовать для выбора последующих шагов в рабочем потоке. Тем не менее, рабочему потоку можно передавать любой другой объект, даже сложного типа с множеством полей.

После того как действие написано, обслуживающий код необходимо изменить так, чтобы он возобновлял свое выполнение с места, указанного в закладке. Но тут возникает другая проблема: откуда обслуживающему коду знать, что в рабочем потоке была создана закладка? Поскольку именно обслуживающий код отвечает за возобновление выполнения с закладки, ему должно быть известно о ее существовании.

Созданное выше действие `Task` на самом деле должно делать дополнительную работу — сообщать внешнему миру о том, что задача была создана. В производственной системе это обычно приводит к сохранению соответствующей записи в таблице очереди и представление этой очереди персоналу центра обработки вызовов в виде списка задач.

Обмен данным с хостом является темой следующего раздела.

Расширения

Расширение — это просто класс или интерфейс, который добавляется в контекст времени выполнения, связанный с приложением рабочего потока. В WF 3.*x* расширения назывались службами, однако из-за возникающей путаницы со службами WCF в WF 4 они были переименованы в расширения.

Обычно для расширений определяется интерфейс и строится его реализация времени выполнения. Действия просто обращаются к этому интерфейсу, что позволяет при необходимости изменять реализации. Хорошим примером может служить создание расширения для отправки сообщения по электронной почте. Можно создать действие `SendEmail`, вызывающее расширение внутри своего метода `Execute()`, а затем определить расширение электронной почты на основе протокола SMTP или расширение Outlook на основе Exchange, которое будет действительно отправлять сообщения электронной почты во время выполнения. Изменять действие для работы с каким-то другим поставщиком услуг отправки электронной почты не понадобится — нужно просто подключить нового поставщика, изменив конфигурационный файл приложения.

Для рассматриваемого примера задачи необходимо расширение, которое будет получать уведомление при переходе действия `Task` в состояние ожидания возобновления работы с указанного в закладке места. В этом расширении может производиться запись имени закладки и другой имеющей к этому отношение информации в базу данных, чтобы впоследствии предоставить пользователю очередь задач. Для определения такого расширения применяется следующий интерфейс (файл `06_WorkflowApplication\SharedInterfaces\ITaskExtension.cs`):

```
public interface ITaskExtension
{
   void ExecuteTask(string taskName);
}
```

Теперь действие `Task` можно обновить так, чтобы оно уведомляло расширение о своем выполнении, изменив метод `Execute()` следующим образом (файл `06_WorkflowApplication\CustomActivities\Task.cs`):

```
protected override void Execute(NativeActivityContext context)
{
   context.CreateBookmark(TaskName.Get(context),
      new BookmarkCallback(OnTaskComplete));
   context.GetExtension<ITaskExtension>().
      ExecuteTask(TaskName.Get(context));
}
```

Объект `context`, передаваемый методу `Execute`, запрашивается для интерфейса `ITaskExtension`, после чего код вызывает метод `ExecuteTask()`. Класс `WorkflowApplication` поддерживает коллекцию расширений, поэтому в нем можно создать класс, реализующий такой интерфейс расширения, после чего использовать его для составления списка задач. Далее можно перейти к созданию и запуску нового рабочего потока, и тогда каждая задача будет уведомлять расширение о своем выполнении. Какой-то другой процесс может просматривать список задач и предоставлять его пользователю.

Чтобы не усложнять пример кода, мы создали только один экземпляр рабочего потока. Этот экземпляр содержит действие `Task` и следующее за ним действие `If`, которое предусматривает вывод соответствующего сообщения в зависимости от того, принимает или отклоняет пользователь выполнение задачи.

Сбор воедино

Теперь вам уже известно, как запускать, сохранять и выгружать рабочий поток, а также доставлять в него события посредством закладок. Осталось узнать лишь о том, как загружать рабочий поток повторно. В случае применения класса `WorkflowApplication` это можно делать вызовом метода `Load()` с передачей ему уникального идентификатора рабочего потока. Каждый рабочий поток имеет уникальный идентификатор, который можно извлечь из объекта `WorkflowApplication`, обращаясь к свойству `Id`. Следовательно, псевдокод приложения, обслуживающего рабочий поток, должен выглядеть так (файл `06_WorkflowApplication\Program.cs`):

```
WorkflowApplication app = BuildApplication();
Guid id = app.Id;
app.Run();
// Ожидание до тех пор, пока будет создана задача, и затем
// выполнение повторной загрузки рабочего потока.
app = BuildApplication();
app.Load(id);
app.ResumeBookmark()
```

Предоставленный пример кода сложнее предыдущего, поскольку также включает в себя реализацию интерфейса `ITaskExtension`, но в целом следует той же схеме, что и ранее. Как не трудно заметить, метод `BuildApplication()` здесь вызывается дважды. Это метод, который использовался в коде для создания экземпляра `WorkflowApplication` и настройки всех требуемых свойств, таких как `InstanceStore` и делегаты для `Completed` и `PersistableIdle`. После первого вызова этого метода выполняется метод `Run()`, который запускает новый экземпляр рабочего потока.

Во второй раз приложение загружается после точки сохранения, так что к этому моменту рабочий поток уже был выгружен и, следовательно, экземпляр приложения является неактуальным. Поэтому мы создаем новый экземпляр `WorkflowApplication`, но вместо метода `Run()` теперь вызываем метод `Load()`, который с помощью поставщика сохранения состояния производит загрузку существующего экземпляра из базы данных. Затем вызовом `ResumeBookmark()` работа этого экземпляра возобновляется.

После запуска этого примера на экране появится приглашение. Хотя приглашение присутствует, рабочий поток сохраняется и выгружается из памяти, в чем легко убедиться, запустив SQL Server Management Studio и выполнив команду, показанную на рис. 45.13.

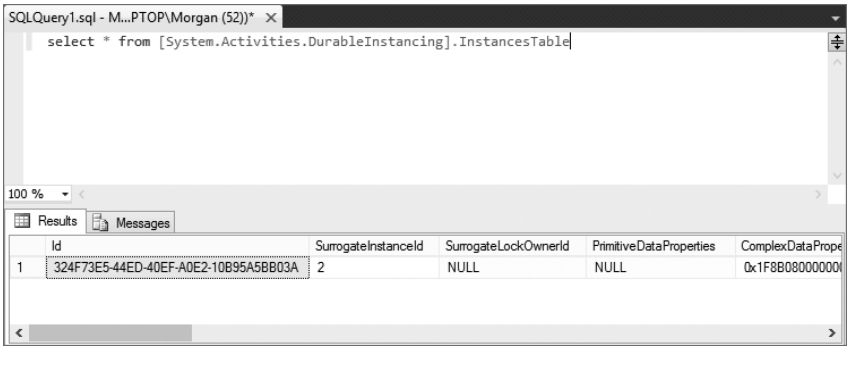

***Рис. 45.13**. Команда для проверки сохранения и выгрузки рабочего потока*

Экземпляры рабочих потоков сохраняются в таблице `InstancesTable` схемы `System.Activities.DurableInstancing`. На рис. 45.13 приведен пример сохраненного экземпляра рабочего потока.

При продолжении выполнения рабочего потока он в конечном итоге завершит работу и после этого удалится из таблицы экземпляров, поскольку у хранилища экземпляров имеется опция под названием `InstanceCompletionAction`, для которой по умолчанию установлено `DeleteAll`. Указанное значение гарантирует, что любые данные, сохраняемые в базе данных для экземпляра рабочего потока, будут сразу же удалены после его завершения. Это имеет смысл, поскольку обычно после завершения рабочего потока его внутренние данные не нужны. Изменить эту опцию можно при определении хранилища экземпляров, установив действие, отвечающее за поведение после завершения экземпляра, в `DeleteNothing`.

Если теперь продолжить выполнение тестового приложения и снова запустить SQL-команду, показанную на рис. 45.13, можно удостовериться в том, что экземпляр рабочего потока был удален.

Обслуживание рабочих потоков WCF

Как упоминалось ранее, для обслуживания рабочих потоков доступны три способа. Последний из них предполагает использование класса `WorkflowServiceHost`, который предоставляет рабочий поток через WCF. Одной из главных областей, на которые ориентирована технология WF, является серверная часть для служб WCF. Типовая служба WCF обычно содержит набор связанных методов, которые, как правило, вызываются в каком-то определенном порядке. Главная проблема здесь в том, что методы могут быть вызваны в любом порядке, поэтому обычно порядок должен определяться так, чтобы, например, подробности заказа не загружались раньше самого заказа.

С помощью WF можно легко открывать доступ к службам WCF, которые также поддерживают упорядоченный вызов методов. Для этого используются два класса действий — `Receive` и `Send`. В приведенном ниже примере кода (из решения `07_WorkflowsAsServices`) демонстрируется сценарий с агентом по недвижимости, которому требуется выгружать информацию об объектах недвижимости на веб-сайт.

Обслуживание потоков с применением WCF можно реализовать двумя способами. Первый из них — явная реализация в коде за счет создания экземпляра `WorkflowServiceHost` (который во многом подобен обычному WCF-классу `ServiceHost`). Но службу можно также создать в виде файла `.xamlx` — в этом случае рабочий поток, скорее всего, будет представляться с помощью IIS или WAS. Именно этот вариант применяется в примере, но стоит помнить, что при необходимости все это можно сделать и вручную.

Этот пример демонстрирует также две новые функциональные возможности WF 4.5. Первая связана с применением рабочего потока типа State Machine (Конечный автомат), а вторая с тем, что реализация этого рабочего потока использует контракт службы WCF, определенный в стиле “сначала контракт”. Тип рабочего потока State Machine отсутствовал в .NET 4, но снова появился в последней версии платформы и оказывается весьма кстати в рассматриваемом примере. Работа конечного автомата, созданного в этом примере, показана на рис. 45.14.

Конечный автомат имеет начальное состояние, любое количество промежуточных состояний и необязательное конечное состояние. Кроме состояний существуют также переходы, которые дожидаются поступления события (как правило, действия, принимающего определенный ввод) и затем переводят конечный автомат в другое состояние.

Конечный автомат работает, ожидая завершения действия перехода. После этого автомат переходит в новое состояние. На входе состояния можно вызывать действие, как и на выходе из состояния. Затем конечный автомат дожидается завершения одного из переходов и переходит в новое состояние.

На приведенной выше блок-схеме (см. рис. 45.14) первое состояние называется Initial State (Начальное состояние). Оно имеет лишь один переход, который ожидает поступления сообщения по WCF.

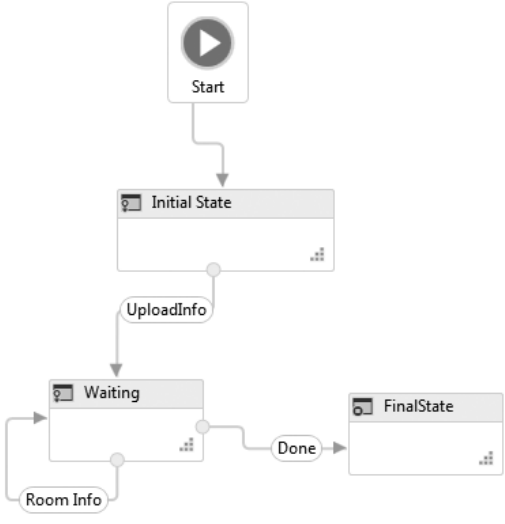

***Рис. 45.14**. Пример конечного автомата*

После получения этого сообщения конечный автомат переходит в состояние Waiting (Ожидание), имеющее два перехода. Они ведут к двум операциям, доступным в интерфейсе WCF: пользователь получает возможность либо отправить сведения о комнате, либо указать, что выгрузка данных завершена. Использованная в этом примере служба WCF реализована следующим образом (файл `07_WorkflowsAsServices\SharedInterfaces\IPropertyInformation.cs`):

```
[ServiceContract(Namespace="http://www.morganskinner.com")]
public interface IPropertyInformation
{
   [OperationContract()]
   Guid UploadPropertyInformation(string ownerName, string address, float price);

   [OperationContract(IsOneWay=true)]
   void UploadRoomInformation(Guid propertyId, string roomName, float width,
      float length);

   [OperationContract(IsOneWay = true)]
   void DetailsComplete(Guid propertyId);
}
```

Метод `UploadPropertyInformation()` запускает конечный автомат. В сущности это создает новый экземпляр рабочего потока, переводит конечный автомат в состояние Waiting, а затем дожидается вызова либо `UploadRoomInformation()`, либо `DetailsComplete()`.

Чтобы использовать в WF существующий контракт службы WCF, нужно открыть контекстное меню для проекта WF и снять отметку с пункта меню Import Service Contract (Импортировать контракт службы). В результате будут созданы действия для каждой операции в контракте службы. Чтобы увидеть эти действия, щелкните на кнопке Show All Files (Показать все файлы) в панели инструментов и разверните элемент текущего проекта в Visual Studio. В результате отобразятся имя контракта (в данном примере это `IPropertyInformation`) и автоматически сгенерированные действия, как показано на рис. 45.15.

Как видите, на основе контракта службы были созданы три действия, которые затем применяются в рабочем потоке для реализации контракта службы.

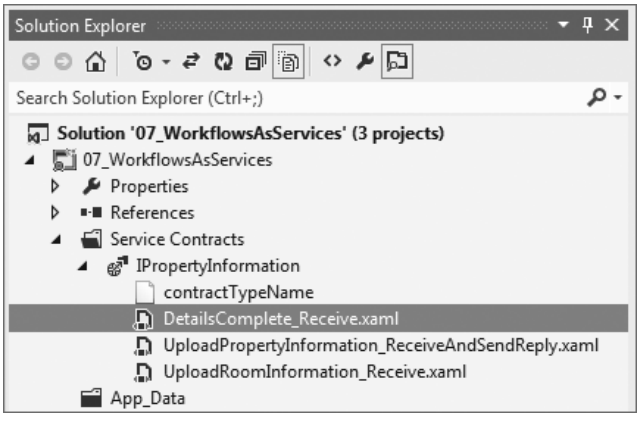

***Рис. 45.15**. Просмотр автоматически сгенерированных действий*

Корреляция

Еще один важный нюанс, который требует понимания при работе с рабочими потоками WCF — способ выполнения вызова из клиента и обеспечение того, чтобы этот вызов мог найти соответствующий экземпляр рабочего потока на сервере. Если взглянуть на интерфейс службы WCF, становится понятно, что для инициализации выгрузки посредством операции `UploadPropertyInformation` нужно обратиться к серверу.

Эта операция возвращает уникальный идентификатор, который передается последующим вызовам `UploadRoomInformation()` и `DetailsComplete()`. Для этого в WF существует концепция корреляции (Correlation), которая, если говорить простыми словами, позволяет использовать произвольный фрагмент информации (или, в действительности, фрагменты информации) для маршрутизации вызова на соответствующий экземпляр рабочего потока. Этот идентификатор возвращается в первом вызове, и до тех пор, пока в следующих вызовах рабочего потока применяется тот же самый идентификатор, данной информации достаточно для определения нужного экземпляра рабочего потока, к которому необходимо маршрутизировать сообщение.

Для работы корреляции требуются два компонента: `CorrelationHandle` (дескриптор корреляции) и инициализатор корреляции. Дескриптор — это просто переменная, определенная с подходящей областью действия (т.е. за пределами всех действий, которые нуждаются в ней). Инициализатор корреляции определяет элемент (элементы), которые используются либо во входящих, либо в исходящих сообщениях для уникальной идентификации экземпляров рабочего потока.

В рассматриваемом примере дескриптор `CorrelationHandle` определен в качестве переменной в корневом действии рабочего потока. Переход `UploadInfo` содержит пару действий Receive/Send (Прием/Отправка). Свойство `CanCreateInstance` действия `Receive` установлено в `True`, которое указывает WF, что когда клиент вызывает сервер, этой операции разрешено быть первой в цепочке выполнения рабочего потока. Действие `Receive` может считывать любые данные из параметров, переданных вызову службы (и сохранять их в рабочем потоке для применения впоследствии, если это потребуется).

Использованное в этом примере действие `Send` устанавливает корреляцию в уникальный идентификатор, созданный внутри рабочего потока. В данном случае, как видно на рис. 45.16, корреляция применяет выражение XPath для извлечения глобально уникального идентификатора, отправленного обратно пользователю в качестве возвращаемого значения из метода `UploadPropertyInformation()`.

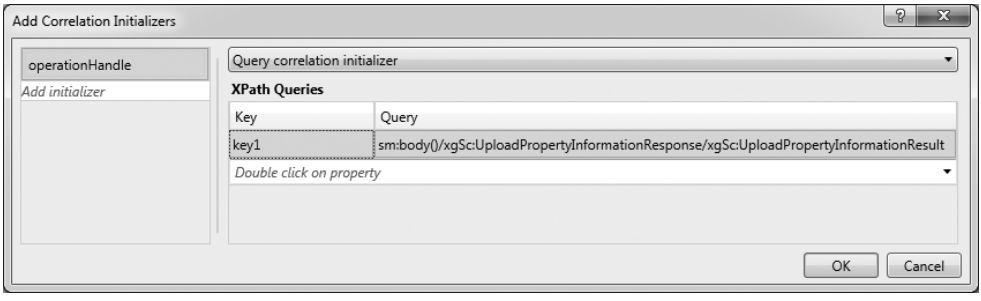

***Рис. 45.16**. Установка корреляции*

Теперь, когда экземпляр рабочего потока фактически связан с этим идентификатором, корреляцию можно использовать снова, чтобы сделать возможными последующие вызовы одного из прочих методов применительно к контракту службы. Они могут найти нужный экземпляр рабочего потока при повторном использовании корреляции — в данном случае для извлечения значения из параметров, переданных операции (операциям), обрабатывающим данные на серверной стороне. Это очень важно. В WF 3.*x* можно было применять только идентификатор встроенного экземпляра рабочего потока. Однако в WF 4.*x* можно выбирать любые уникальные фрагменты данных, чтобы клиент мог находить требуемый экземпляр рабочего потока.

Аналогично можно выполнять корреляцию различных данных во время жизни рабочего потока. Так, например, можно возвратить глобально уникальный идентификатор из первого вызова, но затем возвращать уникальный целочисленный ключ из последующих вызовов методов.

Вызов служб рабочего потока

При обслуживании рабочего потока в виде файла `.xamlx` необходим какой-то способ обращения к этому рабочему потоку. Это аналогично вызову любой стандартной службы WCF в том смысле, что требуется добавить ссылку на эту службу. В данном случае URL-адрес службы заканчивается не файлом `.svc`, а определением рабочего потока `.xamlx`.

Это позволяет клиенту взаимодействовать со службой, выполняя вызовы WCF. Единственное отличие в том, что WF не будет принимать любой вызов этого интерфейса WCF в любое время. Операция (операции), которые разрешены в любой конкретный момент времени, зависят от состояния рабочего потока. При первоначальном вызове рабочего потока используется вызов `UploadPropertyInformation`, который благодаря приемнику этого вызова, чье свойство `CanCreateInstance` установлено в `true`, позволяет создать новый экземпляр рабочего потока.

После этого применительно к интерфейсу можно вызвать любой другой метод, поскольку именно этого теперь ожидает экземпляр рабочего потока.

Загружаемый код этого примера включает в себя прокси-класс, аналогичный тому, что создавался при добавлении ссылки на службу. Создавать эти экземпляры в коде предпочтительнее, поскольку в таком случае процесс более понятен, чем при добавлении ссылки на службу, а также лучше подходит для поддержки больших групп разработчиков, когда интерфейс службы может часто изменяться. Применение в данном случае метода подразумевает необходимость внесения изменений только в одно место, а не повторную генерацию ссылок на службы в нескольких проектах.

Код, создающий экземпляр этого прокси-класса, выглядит следующим образом:

```
PropertyInformationClient client = new PropertyInformationClient("state");
```

Этот код использует конечную точку, определенную внутри файла `App.config`:

```
<endpoint address="http://localhost:1353/StateService.xamlx"
          binding="basicHttpBinding"
          contract="SharedInterfaces.IPropertyInformation"
          name="state"/>
```

Как видите, адрес службы заканчивается расширением `.xamlx`, который свидетельствует о том, что служба является рабочим потоком, а не стандартной службой WCF.

Управление версиями рабочих потоков

Поскольку рабочий поток может поддерживать постоянство и потенциально выполняться в течение длительного времени, необходим способ изменения рабочего потока. Например, рассмотрим рабочий поток для обработки данных по продлению договоров страхования. Экземпляр этого рабочего потока может выполняться для каждого нового выданного страхового полиса. Через 10 месяцев после выдачи полиса он может снова активизироваться для отправки запроса на продление страхового договора и, в зависимости от ответа клиента, отправки одного или двух напоминаний.

Теперь предположим, что вы реализуете новый механизм взаимодействия с клиентами, который позволяет отправлять текстовые сообщения для информирования клиентов о продлении их договоров. А что делать со всеми рабочими потоками, уже существующими в базе данных?

До выхода WF 4.5 были доступны только две возможности, причем обе не особо привлекательны. Первая состояла в игнорировании изменений любых существующих рабочих потоков и реализации изменений только в новых экземплярах рабочих потоков. Второй возможностью была остановка всех существующих экземпляров и их перезапуск. Обе эти возможности были далеки от идеала, и в WF 4.5 добавлено несколько функциональных средств, которые могут помочь в этом плане.

Динамическое обновление

Функция динамического обновления позволяет изменять постоянный экземпляр рабочего потока для добавления в него определенной новой функциональности. Для обновления рабочего потока понадобится предпринять несколько шагов, которые можно разделить на два этапа.

На первом этапе выполняется подготовка к обновлению существующих рабочих потоков. При этом используется новый класс `DynamicUpdateServices` посредством вызова метода `PrepareForUpdate()`, как показано в следующем коде. Затем рабочий поток должным образом изменяется, например, за счет добавления в него новых действий. Пример содержит новое действие `WriteLine`, добавленное в конец текущего рабочего потока. После того как все изменения внесены, можно вызвать метод `CreateUpdateMap()`, который возвращает экземпляр класса `DynamicUpdateMap`, используемого при загрузке существующих экземпляров для их обновления до нового определения рабочего потока (файл `08_DynamicUpdate\Program.cs`):

```
Activity workflowDefinition = GetInitialWorkflow();
DynamicUpdateServices.PrepareForUpdate(workflowDefinition);

// Обновление рабочего потока — добавление в него нового действия.
Sequence seq = workflowDefinition as Sequence;
seq.Activities.Add(new WriteLine { Text = "Second version of workflow" });

// Создание карты после внесения всех изменений.
return DynamicUpdateServices.CreateUpdateMap(workflowDefinition);
```

Карту обновления можно использовать многократно для обновления экземпляров рабочих потоков. Именно поэтому было сказано о двух этапах — второй этап связан с загрузкой существующих рабочих потоков и их обновлением. В следующем фрагменте кода показано, как можно было бы загрузить существующий экземпляр рабочего потока и применить ранее созданную карту обновления для обновления этого экземпляра:

```
SqlWorkflowInstanceStore store = new SqlWorkflowInstanceStore(
   ConfigurationManager.ConnectionStrings["db"].ConnectionString);
WorkflowApplicationInstance instance = WorkflowApplication.GetInstance(id, store);
WorkflowApplication app = new WorkflowApplication(GetUpdatedWorkflow());
app.Load(instance, map);
app.Unload();
```

В этом примере сначала с помощью метода `GetUpdatedWorkflow()` загружается новое определение рабочего потока. Затем данные существующего экземпляра загружаются в это новое определение рабочего потока посредством объекта `map`, указывающего, как должно осуществляться перемещение постоянных данных между существующими и новыми действиями. Поскольку постоянные данные рабочего потока содержат только данные, но не определение рабочего потока, обновление сводится к простому перемещению постоянной информации между узлами рабочего потока.

После того как рабочий поток обновлен, он должен быть запущен применительно к новому определению рабочего потока. Новый класс `WorkflowIdentity`, который может быть присоединен к `WorkflowApplication` во время создания приложения, упрощает задачу выяснения того, какая версия рабочего потока внесена в базу данных. Эта информация является частью постоянной информации рабочего потока и может быть доступна посредством свойства `DefinitionIdentity` класса `WorkflowApplicationInstance`. Это свойство можно использовать для итерации по всем рабочим потокам в постоянном хранилище, чтобы выяснить, какие из них были обновлены, а какие требуют изменения.

Существование разных версий рабочих потоков бок о бок

При обслуживании рабочих потоков с помощью класса `WorkflowApplication` теперь можно включить объект `WorkflowIdentity`, который позволяет связать информацию о версии с определением рабочего потока, чтобы при сохранении рабочего потока в базе данных информация о версии также сохранялась. Тогда при считывании постоянных экземпляров рабочих потоков из базы данных можно найти рабочие потоки определенной версии, чтобы сохраненное состояние рабочего потока в базе данных можно было отобразить определением рабочего потока времени выполнения.

Чтобы можно было загружать и возобновлять любой экземпляр рабочего потока, объект `WorkflowApplication`, обслуживающий этот рабочий поток, должен быть инициализирован конкретным определением рабочего потока, которое применялось для создания данного рабочего потока. Считывая информацию о версии до возобновления рабочего потока, эти сведения можно использовать для отображения на конкретное определение рабочего потока.

Кроме того, такая поддержка версий была расширена на рабочие потоки, обслуживаемые с помощью класса `WorkflowServiceHost`. При обслуживании рабочих потоков, имеющих версии, с помощью `WorkflowServiceHos` нужно указать информацию о версии в свойстве `DefinitionIdentity` службы рабочего потока.

Описанные добавления делают рабочий поток значительно более гибким вариантом, чем было раньше, особенно для производственных процессов, которые могут длиться неделями или месяцами. На протяжении более длительного периода необходимость внесения изменений становится значительно более вероятной. Хотя реализация этих изменений — задача непростая, теперь, по крайней мере, мы располагаем средством ее решения.

Размещение визуального конструктора

Визуальный конструктор рабочих потоков, используемый в Visual Studio, можно также размещать внутри собственного приложения; это позволяет конечным пользователям создавать свои рабочие потоки, не имея копии Visual Studio. На данный момент это одна из наиболее привлекательных особенностей WF 4. Традиционные механизмы расширения приложений всегда требуют некоторого участия разработчика — либо написания DLL-

библиотеки расширений и подключения ее к системе, либо создания макросов или сценариев. Инфраструктура WF позволяет конечным пользователям настраивать приложение, просто перетаскивая действия на поверхность конструктора.

Изменение места размещения конструктора в WF 3.*x* было делом непростым. В WF 4 оно превратилось в практически тривиальную задачу. Конструктор представляет собой элемент управления WPF, поэтому проект WPF можно использовать в качестве главного приложения. Код рассматриваемого здесь примера находится в проекте `09_DesignerRehosting`.

В первую очередь понадобится включить сборки рабочих потоков, а затем определить XAML-код для визуализации главного окна. При конструировании пользовательских интерфейсов целесообразно использовать шаблон Model-View-ViewModel (MVVM), поскольку это упрощает процесс кодировании, а также обеспечивает возможность применения другого XAML к той же модели представления. Ниже показан XAML-код для визуализации главного окна (файл `09_DesignerRehosting\MainWindow.xaml`):

```
<Window x:Class="HostApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Menu IsMainMenu="True">
      <MenuItem Header="_File">
        <MenuItem Header="_New" Command="{Binding New}"/>
        <MenuItem Header="_Open" Command="{Binding Open}"/>
        <MenuItem Header="_Save" Command="{Binding Save}"/>
        <Separator/>
        <MenuItem Header="_Exit" Command="{Binding Exit}"/>
      </MenuItem>
      <MenuItem Header="Workflow">
        <MenuItem Header="_Run" Command="{Binding Run}"/>
      </MenuItem>
    </Menu>
    <Grid Grid.Row="1">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="4*"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <ContentControl Content="{Binding Toolbox}" />
      <ContentControl Content="{Binding DesignerView}"
        Grid.Column="1"/>
      <ContentControl Content="{Binding PropertyInspectorView}"
        Grid.Column="2"/>
    </Grid>
  </Grid>
</Window>
```

Как видите, компоновка очень проста и предусматривает размещение в главном окне лишь меню и сетки с заполнителями для вставки панели инструментов, конструктора и таблицы свойств. Также нетрудно заметить, что все элементы здесь имеют привязку, в том числе и команды.

Созданный класс `ViewModel` состоит из свойств каждого из основных элементов пользовательского интерфейса: панели инструментов, конструктора и таблицы свойств. Кроме этих свойств существуют также свойства для каждой из команд, такой как `New`, `Save` и `Exit` (файл `09_DesignerRehosting\ViewModel.cs`):

```
public class ViewModel : BaseViewModel
{
   public ViewModel()
   {
      // Обеспечение регистрации всех конструкторов для встроенных действий
      new DesignerMetadata().Register();
   }
   public void InitializeViewModel(Activity root)
   {
      _designer = new WorkflowDesigner();
      _designer.Load(root);
      this.OnPropertyChanged("DesignerView");
      this.OnPropertyChanged("PropertyInspectorView");
   }
   public UIElement DesignerView
   {
      get { return _designer.View; }
   }
   public UIElement ProperttInspectorView
   {
      get { return _designer.PropertyInspectorView; }
   }
   private WorkflowDesigner _designer;
}
```

Класс `ViewModel` унаследован от базового класса `BaseViewModel`, который используется при каждом создании модели представления, потому что он предоставляет реализацию интерфейса `INotifyPropertyChanged`. Этот класс образован из ряда фрагментов, написанных Джошем Твистом (Josh Twist) и доступен на сайте `www.thejoyofcode.com`.

Далее в конструкторе обеспечивается регистрация метаданных для всех встроенных действий — без этого вызова никакие из визуальных конструкторов конкретных типов не будут появляться в пользовательском интерфейсе. Затем в методе `InitializeViewModel()` создается экземпляр конструктора WF и производится загрузка действия в него. Класс `WorkflowDesigner` примечателен тем, что после загрузки в него одного рабочего потока загрузка другого невозможна. Поэтому в данном случае этот класс воссоздается при каждом создании нового рабочего потока.

Напоследок в методе `InitializeViewModel()` вызывается функция уведомления об изменениях в свойствах, чтобы указать пользовательскому интерфейсу на необходимость обновления свойств `DesignerView` и `PropertyInspectorView`. Поскольку пользовательский интерфейс привязан к этим свойствам, он произведет загрузку новых значений из нового экземпляра визуального конструктора рабочих потоков.

Следующей частью пользовательского интерфейса, которую требуется создать, является панель инструментов. В `WF 3.x` такой элемент управления нужно было создавать самостоятельно, а в `WF 4` для этого предусмотрен элемент управления `ToolboxControl`, который очень прост в применении (файл `09_DesignerRehosting\ViewModel.cs`).

```
public UIElement Toolbox
{
   get
   {
      if (null == _toolbox)
      {
         _toolbox = new ToolboxControl();
         ToolboxCategory cat = new ToolboxCategory
            ("Standard Activities");
         cat.Add(new ToolboxItemWrapper(typeof(Sequence),
            "Sequence"));
         cat.Add(new ToolboxItemWrapper(typeof(Assign), "Assign"));
```

```
            _toolbox.Categories.Add(cat);
            ToolboxCategory custom = new ToolboxCategory("Custom Activities");
            custom.Add(new ToolboxItemWrapper(typeof(Message), "MessageBox"));
            _toolbox.Categories.Add(custom);
         }
         return _toolbox;
      }
   }
```

Здесь сначала создается элемент управления `ToolboxControl`, затем добавляются два элемента `ToolboxItem` в первую категорию и один — во вторую. Класс `ToolboxItemWrapper` применятся для упрощения кода, необходимого для добавления заданного действия в панель инструментов.

После написания всего этого кода мы получаем почти готовое работающее приложение. Остается лишь связать `ViewModel` с кодом XAML, что и делается в конструкторе главного окна:

```
public MainWindow()
{
   InitializeComponent();
   ViewModel vm = new ViewModel();
   vm.InitializeViewModel(new Sequence());
   this.DataContext = vm;
}
```

В приведенном коде создается модель представления, в которую по умолчанию добавляется действие `Sequence`, чтобы при запуске приложения на экране хоть что-нибудь отображалось.

Теперь не хватает только некоторых команд. При написании команд на основе интерфейса `ICommand` для приложения WPF используется класс `DelegateCommand`, поскольку в этом случае код в модели представления получается более понятным. Реализация команд выглядит довольно тривиально, как можно убедиться на примере приведенной ниже команды `New`:

```
public ICommand New
{
   get
   {
      return new DelegateCommand(unused =>
      {
         InitializeViewModel(new Sequence());
      });
   }
}
```

Эта команда привязывается к элементу меню New (Создать), чтобы после щелчка на нем выполнялся делегат, который в данном случае предусматривает просто вызов метода `InitializeViewModel()` с новым действием `Sequence`. Поскольку этот метод также предусматривает генерацию уведомления об изменении свойств для визуального конструктора и таблицы свойств, они будут также обновляться.

Реализация команды `Open` несколько сложнее, но не намного:

```
public ICommand Open
{
   get
   {
      return new DelegateCommand(unused =>
      {
         OpenFileDialog ofn = new OpenFileDialog();
         ofn.Title = "Open Workflow";
         ofn.Filter = "Workflows (*.xaml)|*.xaml";
```

```
            ofn.CheckFileExists = true;
            ofn.CheckPathExists = true;
            if (true == ofn.ShowDialog())
               InitializeViewModel(ofn.FileName);
         });
      }
   }
```

Здесь применяется еще одна переопределенная версия метода `InitializeViewModel()`, принимающая на этот раз не действие, а имя файла. В главе код не показан, но он доступен в примерах для настоящей главы. Эта команда `Open` отображает элемент управления `OpenFileDialog` и при выборе файла загружает рабочий поток в визуальный конструктор. Кроме того, имеется команда `Save`, которая вызывает метод `WorkflowDesigner.Save()` для сохранения XAML-данных рабочего потока на диске. Теперь при выполнении приложения отображается окно, подобное показанному на рис. 45.17.

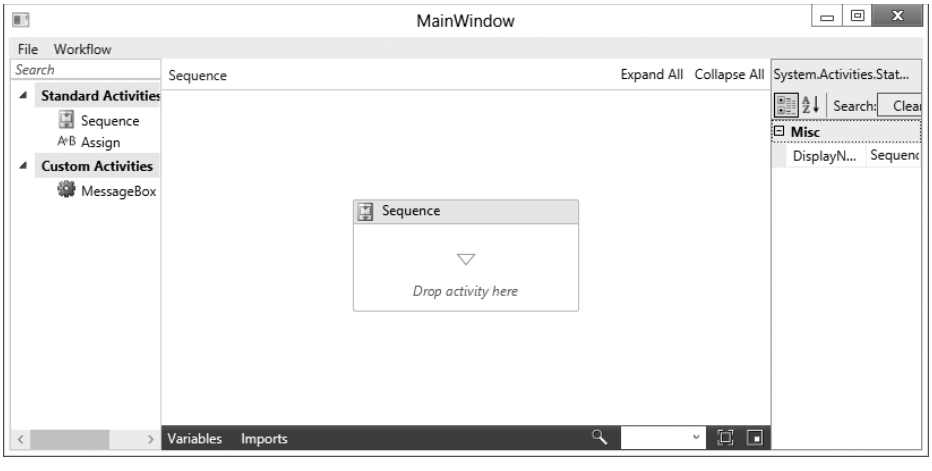

***Рис. 45.17**. Пример приложения во время выполнения*

В последнем разделе кода модели представления реализована команда `Run`. От возможности проектировать рабочие потоки без их выполнения мало толку, поэтому в модель представления включено и такое средство. Все выглядит довольно тривиально: конструктор включает свойство `Text`, которое является XAML-представлением действий внутри рабочего потока. Все, что необходимо сделать — это преобразовать его в `Activity` и затем выполнить с помощью класса `WorkflowInvoker`.

```
public ICommand Run
{
   get
   {
      return new DelegateCommand(unused =>
      {
         Activity root = _designer.Context.Services.
            GetService<ModelService>().Root.
            GetCurrentValue() as Activity;
         WorkflowInvoker.Invoke(root);
      },
      unused => { return !HasErrors; }
      );
   }
}
```

```
public bool HasErrors
{
   get { return (0 != _errorCount); }
}
public void ShowValidationErrors(IList<ValidationErrorInfo> errors)
{
   _errorCount = errors.Count;
   OnPropertyChanged("HasErrors");
}
private int _errorCount;
```

Чтобы приведенный выше код уместился на странице, пришлось добавить разрывы строк: первая строка команды делегата, которая извлекает корневое действие из конструктора, является очень длинной. После этого осталось воспользоваться методом `WorkflowInvoker.Invoke()` для выполнения рабочего потока.

В инфраструктуре команд WPF предусмотрен способ для отключения команд в случае невозможности получения к ним доступа; как раз для этого предназначена вторая лямбда-функция в `DelegateCommand`. Эта функция возвращает значение `HasErrors` — булевского свойства, которое было добавлено в модель представления. Это свойство показывает, не были ли внутри рабочего потока обнаружены какие-то ошибки проверки достоверности, т.к. в модели представления реализован интерфейс `IValidationErrorService`, который уведомляется при каждом изменении допустимого состояния рабочего потока.

Этот пример можно было бы расширить, добавив возможность отображения ошибок проверки достоверности, а также дополнительные действия в панель инструментов.

Резюме

Инфраструктура WF радикально изменяет подход к созданию приложений. Уже сейчас она позволяет представлять сложные части приложения в виде действий и давать пользователям возможность изменять предлагаемый системой способ обработки, просто перетаскивая эти действия в рабочий поток.

Рабочий поток можно применять практически к любым приложениям, начиная с простейших утилит командной строки и заканчивая сложнейшими системами со многими сотнями модулей. Там где раньше требовалось привлекать разработчика для написания специального модуля расширения к системе, теперь можно предложить простой и расширяемый механизм настройки, которым сможет пользоваться почти кто угодно. Производителям приложений раньше нужно было самим предоставлять специальные действия для взаимодействия с их системами, а также код, способный вызывать рабочие потоки. Однако теперь они могут оставлять выбор за своими потребителями и позволить им самостоятельно определять, что должно происходить при возникновении в их приложении того или иного события.

В настоящее время WF 3.*x* интенсивно замещается версией WF 4, и если вы впервые планируете использовать рабочие потоки, начните именно с этой версии, полностью отказавшись от WF 3.*x*.