



# 44

## Технология Windows Workflow Foundation 4

### **В ЭТОЙ ГЛАВЕ...**

- Различные типы рабочих потоков, которые позволяет создавать
- Описание некоторых встроенных действий
- Создание специальных действий
- Краткое рассмотрение рабочего потока

В настоящей главе предлагается краткий обзор технологии Windows Workflow Foundation 4 (в дальнейшем просто Workflow 4), которая предоставляет модель, позволяющую определять и выполнять процессы с помощью ряда строительных блоков, называемых *действиями* (activity). В WF имеется визуальный конструктор, который по умолчанию обслуживается в Visual Studio и позволяет перетаскивать действия из панели элементов управления 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 часто применяемых методов. Вдобавок рабочему потоку может требоваться сохранять и восстанавливать свое состояние, например, при возникновении необходимости в длительном ожидании.

Рабочий поток конструируется из ряда 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 работ общего назначения. Разумеется, можно создавать собственные действия и должным образом включать их в рабочий поток.

В Visual Studio 2010 теперь, по сути, предлагается две версии Workflow — Workflow 3.x, которая поставляется вместе с .NET Framework 3, и Workflow 4, входящая в состав .NET Framework 4. В настоящей главе главным образом речь идет о последней версии Workflow — Workflow 4; предшествующая версия Workflow рассматривается в главе 57.

В целом функциональные возможности этих версий Workflow похожи, но масса мелких отличий в них есть. Тем, кто лишь приступает к использованию Workflow, настоятельно рекомендуется начинать с версии Workflow 4; для тех, кому приходилось пользоваться версией 3.x, в главе 57 предлагаются советы, которые помогут перейти на версию Workflow 4.

В начале главы рассматривается канонический пример, которым пользуются все, когда сталкиваются с любой новой технологией — пример создания приложения “Hello World”.

## Пример создания приложения “Hello World”

В Visual Studio 2010 имеется встроенная поддержка для создания проектов рабочих потоков, ориентированных на версии .NET Framework 3 и 4, и при открытии диалогового окна New Project (Новый проект) отображается список таких типов проектов (рис. 44.1).

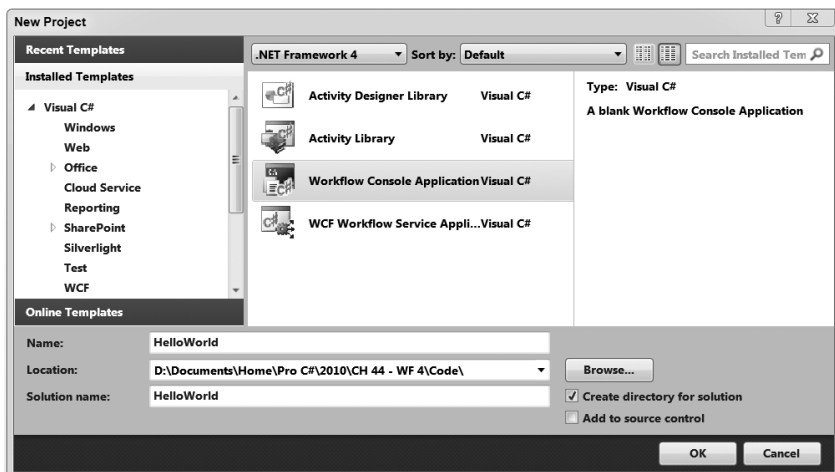


Рис. 44.1. Список проектов для создания рабочих потоков в Visual Studio 2010

Удостоверьтесь, что в списке выбрана версия .NET Framework 4, и выберите среди доступных шаблонов проект 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 элементов управления на поверхность проектирования действие WriteLine, чтобы получить рабочий поток, выглядящий так, как показано на рис. 44.2.

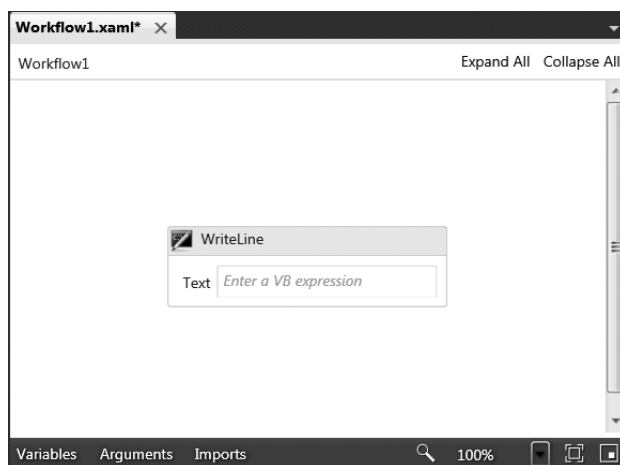


Рис. 44.2. Добавление действия WriteLine

Действие WriteLine включает в себя свойство Text, которое можно настраивать либо прямо на самой поверхности проектирования, введя нужный текст в соответствующей строке, либо отобразив таблицу свойств. Позже в главе будет показано, как определять специальные действия таким образом, 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 помещено в пару двойных кавычек, т.е. в рассматриваемом примере для свойства Text потребуется ввести выражение "Hello World". Если кавычки опущены, при компиляции возникнет ошибка, поскольку без кавычек это выражение не является допустимым.

После сборки и запуска данной программы, именно такой текст будет выведен на консоль. При выполнении программы в методе Main будет создаваться экземпляр рабочего потока, который затем выполняется с помощью содержащегося там же статического метода класса WorkflowInvoker. Код этого примера находится в решении 01\_HelloWorld.

Класс WorkflowInvoker появился в Workflow 4 и позволяет вызывать рабочий поток синхронным образом. Вдобавок доступно два других метода для выполнения рабочего потока, которые позволяют вызывать его асинхронно; они рассматриваются позже в главе. Синхронное выполнение было возможным и в версии Workflow 3.x, но его было немного труднее обеспечивать, причем связанных с этим накладных расходов было гораздо больше.

Синхронная природа WorkflowInvoker делает его идеальным вариантом для выполнения короткоживущих рабочих потоков, в ответ на производимое в пользовательском интерфейсе действие, например, включение или отключение элементов пользовательского интерфейса. Хотя в Workflow 3.x подобное тоже было возможным, обеспечивать синхронное выполнение экземпляра рабочего потока было гораздо сложнее.

## Действия

Все в рабочем потоке является действием, в том числе и сам поток. Рабочий поток представляет собой действие особого типа, внутри которого обычно можно определять другие действия и которое в таком случае называется составным действием; далее в настоящей главе еще будут демонстрироваться различные составные действия. Любое действие представляет собой просто класс, который в конечном итоге унаследован от класса `Activity`.

Иерархия доступных для создания действий классов в Workflow 4 немного сложнее, чем в Workflow 3.x; главные классы в иерархии показаны на рис. 44.3.

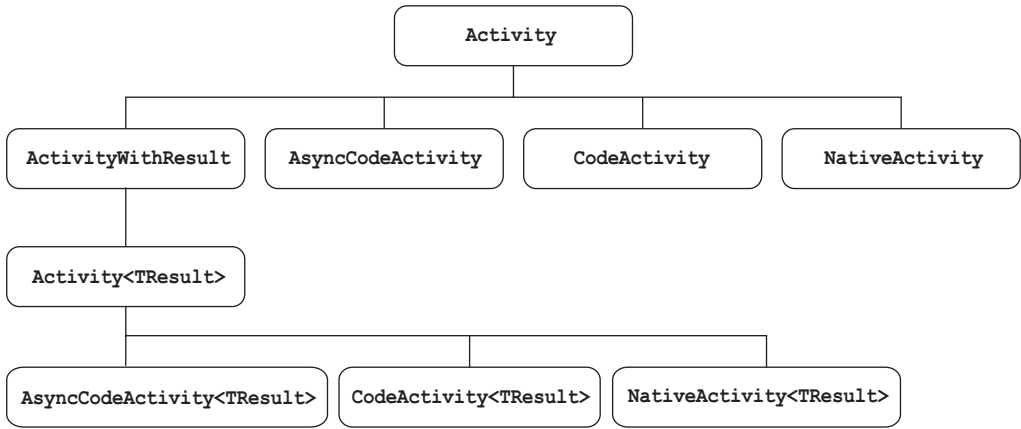


Рис. 44.3. Главные классы в иерархии классов для создания действий в Workflow 4

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 нужно осуществлять от класса `CodeActivity`, потому что этот класс обладает достаточными для функционирования специального клона `WriteLine` возможностями. Действия, которые должны выполняться и возвращать какой-то результат, должны наследоваться от класса `ActivityWithResult`. Обратите внимание, что в этом случае предпочтительнее использовать обобщенный класс `Activity<TResult>`, поскольку он имеет строго типизированное свойство `Result`.

Сложнее всего выбирать базовый класс для наследования при создании специальных действий, и потому в настоящей главе будут приведены примеры, которые помогут понять, как правильно выбирать такой класс.

Чтобы действие выполняло какие-то операции, обычно переопределяется метод `Execute()`, сигнатура которого зависит от выбранного базового класса, как можно видеть в табл. 44.1.

Нетрудно заметить, что параметр или параметры, передаваемые методу `Execute`, отличаются используемым классом контекста. В Workflow 3.x использовался только один класс контекста (`ActivityExecutionContext`), а в Workflow 4 для разных классов действий применяются разные классы контекстов.

Главное отличие состоит в том, что класс `CodeActivityContext` (и производный 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.

Таблица 44.1. Метод Execute в разных базовых классах

Базовый класс	Сигнатура метода Execute
AsyncCodeActivity	IAAsyncResult BeginExecute( AsyncCodeActivityContext, AsyncCallback, object)  void EndExecute(AsyncCodeActivityContext, IAAsyncResult)
CodeActivity	void Execute(CodeActivityContext)
NativeActivity	void Execute(NativeActivityContext)
AsyncCodeActivity< TResult>	IAAsyncResultBeginExecute( AsyncCodeActivityContext,AsyncCallback, object)  TResultEndExecute (AsyncCodeActivityContex, IAAsyncResult)
CodeActivity<TResult>	TResult Execute(CodeActivityContext)
NativeActivity<TResult>	void Execute(NativeActivityContext)

Например, показанное ранее действие WriteLine должно лишь выводить строку текста на консоль. Следовательно, возможность доступа к своей 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 нужна возможность доступа ко всей исполняющей среде, и оно должно наследоваться от класса NativeActivity. Мы еще вернемся к этой теме при рассмотрении способов создания собственных специальных действий.

В Workflow 4 поставляется множество стандартных действий, и в следующих подразделах приводятся примеры некоторых из них вместе со сценариями использования. Тримя главными сборками со стандартными действиями, которые применяются в Workflow 4, 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. 44.4.

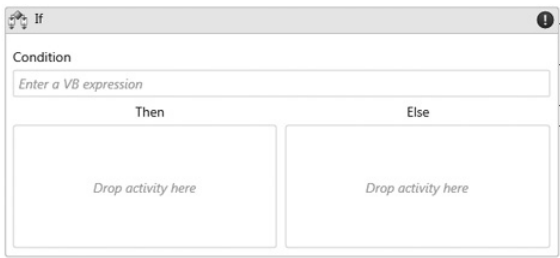


Рис. 44.4. Перетаскивание действия If на поверхность конструктора

If представляет собой составное действие, содержащее два дочерних действия, для добавления которых на экране отображаются две соответствующих ветви Then и Else. Действие, показанное на рис. 44.4, также включает глиф, указывающий на необходимость

определения свойства `Condition`. Условие, задаваемое в этом свойстве, будет вычисляться при выполнении данного действия: если результатом его вычисления является `True`, будет выполняться ветвь `Then`, а если `False` — ветвь `Else`.

Свойство `Condition` предусматривает указание выражения, возвращающего булевское значение, так что в нем можно вводить любое выражение, которое отвечает этому требованию.

В состав `Workflow 4` входит механизм выражений, в котором используется синтаксис `Visual Basic`. Программистам на `C#` это может показаться странным, потому что синтаксис `VB` существенно отличается от синтаксиса `C#`. К сожалению, на данный момент ситуация обстоит именно так, поэтому чтобы использовать встроенные действия, нужно достаточно хорошо разбираться в синтаксисе `VB`.

В выражении можно ссылаться на любые определяемые в рабочем потоке переменные, а также получать доступ 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`. Более подробно об аргументах и переменных будет рассказываться позже в этой главе.

## Действие `InvokeMethod`

Это действие является одним из самых полезных среди встроенных действий, поскольку позволяет выполнять существующий код и, по сути, упаковывать его в оболочку семантики выполнения рабочего потока.

Наличие массы существующего кода является типичной ситуацией, а данное действие позволяет легко напрямую вызывать этот код внутри рабочего потока.

Действие `InvokeMethod` используется двумя способами. 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 использовать свойства `TargetObject` и `MethodName` и создать `TargetObject` в этом экземпляре прямо внутри его строк кода либо определить в переменной внутри рабочего потока. Примеры обоих способов применения действия `InvokeMethod` можно найти в демонстрационном проекте `02_ParallelExecution`.

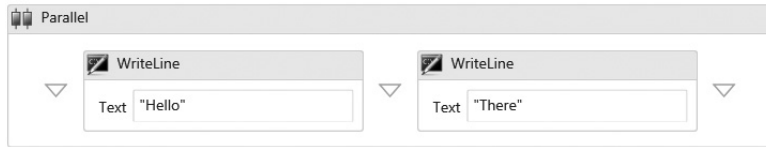
Если вызываемому методу необходимо передать какие-то аргументы, их можно определить с помощью коллекции `Parameters`. Порядок параметров в этой коллекции должен совпадать с тем, в котором их нужно передавать методу. Кроме того, доступно свойство `Result`, которому присваивается значение, возвращаемое 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 машинах с несколькими процессорами, но на самом деле это не так, за исключением некоторых особых случаев.

После перетаскивания действия `Parallel` на поверхность конструктора можно перетаскивать в него подчиненные действия, как показано на рис. 44.5.

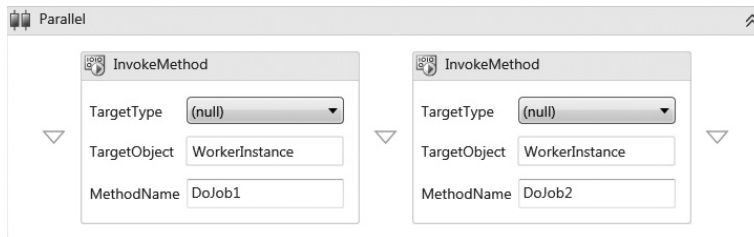
Дочерние действия могут представлять собой одиночные действия, как на рис. 44.5, и составные действия вроде `Sequence` или еще одного действия `Parallel`.



*Рис. 44.5. Перетаскивание действия Parallel и его подчиненных действий на поверхность конструктора*

Во время выполнения действие Parallel планирует выполнение каждого следующего за ним дочернего действия. Механизм выполнения лежащей в основе исполняющей среды затем планирует эти дочерние действия по принципу FIFO (“первым пришел, первым обслужен”), создавая иллюзию их параллельного выполнения, однако, на самом деле они выполняются в рамках одного потока.

Чтобы обеспечить действительно параллельное выполнение, действия, перетаскиваемые в Parallel, должны быть унаследованы от класса `AsyncCodeActivity`. В демонстрационном проекте `02_ParallelExecution` предлагается пример асинхронной обработки кода в двух ветвях действия Parallel. На рис. 44.6 показано применение в действии Parallel двух действий `InvokeMethod`.



*Рис. 44.6. Использование двух действий InvokeMethod в действии Parallel*

Используемые здесь действия `InvokeMethod` предусматривают вызов двух простых методов `DoJob1` и `DoJob2`, которые обеспечивают засыпание, соответственно, на две и три секунды. Чтобы эти методы выполнялись асинхронно, необходимо внести еще одно изменение. Действие `InvokeMethod` имеет булевское свойство `RunAsynchronously`, для которого по умолчанию устанавливается значение `False`. Если установить его в `True`, целевой метод сможет вызываться асинхронно и, следовательно, позволить Parallel выполнять более одного дочернего действия одновременно. На машине с одним процессором будут использоваться два потока, создавая иллюзию одновременного выполнения, а на машине с несколькими процессорами эти потоки смогут выполняться в разных ядрах, обеспечивая настоящий параллелизм. При создании собственных действий лучше делать их асинхронными, поскольку тогда 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 переводиться в состояние ожидания до тех пор, пока не будет получено одобрение (или отказ). Кроме того, совсем не плохо определить период тайм-аута, чтобы при отсутствии ответа в течение, скажем, одного дня, заявка на расходы перенаправлялась следующему руководителю в цепочке команды.

С помощью действия `Delay` можно реализовать одну часть этого сценария (другую часть будет решать действие `Pick`, рассматриваемое в следующем разделе). Обязанностью этого действия является заставлять поток находиться в состоянии ожидания на протяжении определенного времени, после чего продолжить работу.

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 `Idle`, в котором запускает действие `Delay`. Потоки, входящие в такое состояние, являются кандидатами на постоянное хранение — именно здесь данные экземпляра рабочего потока могут сохраняться в каком-нибудь постоянном хранилище (например, базе данных `SQL Server`), а сам поток выгружаться 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 примере процессу одобрения заявок на расходы. Здесь понадобится действие `Pick` с тремя ветвями, одна — для обработки случаев принятия заявок, другая — для обработки случаев отклонения заявок, а третья — для обработки случаев срабатывания тайм-аута.

Необходимый код 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. В случае истечения 10 или более секунд приглашение закрывается и запускается действие из ветви `Delay`.

В этом примере первым в рабочем потоке используется действие `DisplayPrompt`. Это действие предусматривает вызов определяемого в интерфейсе метода 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, а 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, предусматривающее отображение сообщения о подтверждении заявки. В случае отклонения заявки завершается действие `WaitForReject` и производится вывод сообщения об отклонении заявки.

Если ни действие `WaitForAccept`, ни действие `WaitForReject` не завершено, действие `WaitForTimeout` в конечном счете завершается по истечении указанного периода отсрочки. После этого заявка на расходы может быть далее передана другому начальнику, воз-



можно, 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 `ClosePrompt`, как показано на рис. 44.7.

Некоторые из использованных в этом примере концепций, такие как создание специальных действий или ожидание поступления внешних событий, рассматриваются далее в главе.

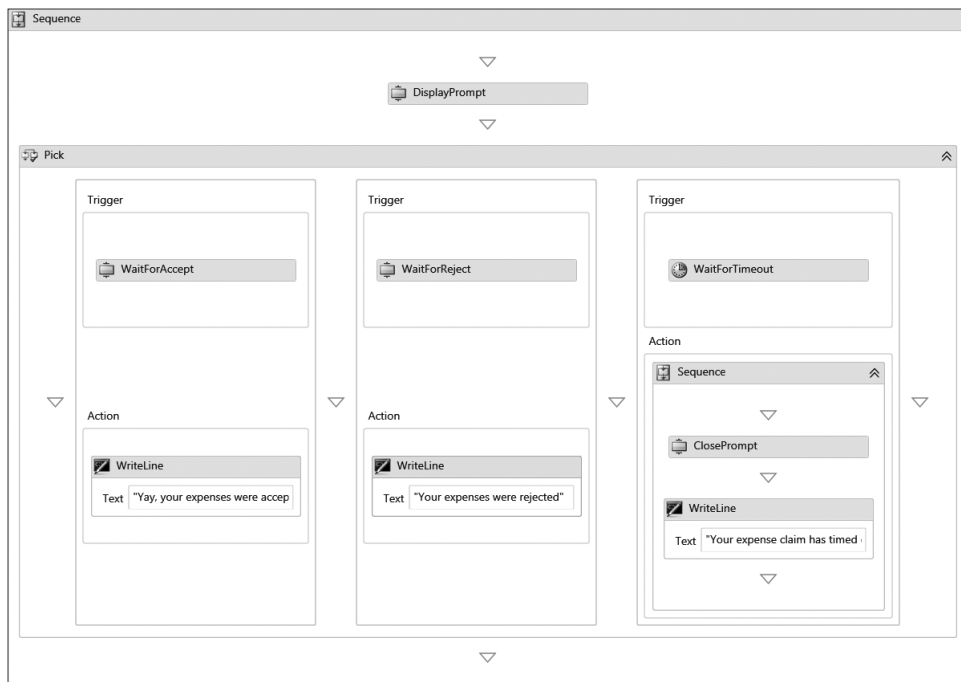


Рис. 44.7. Пример использования в рабочем потоке действия `Pick` с тремя ветвями

## Специальные действия

До сих пор использовались действия, определенные в пространстве имен `System.Activities`. В этом разделе будет показано, как создавать специальные действия и расширять их возможности для улучшения пользовательского впечатления на этапах проектирования и 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 расширяться 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 в среде проектирования Visual Studio (в частности, в проектах рабочих потоков) можно загружать действия из сборок и блокировать сборку, которая должна быть обновлена. По этой причине для конструирования специальных действий лучше создавать проект библиотеки классов. Код рассматриваемого здесь примера находится 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.

```

using System;
using System.Activities;
using System.Diagnostics;

namespace Activities
{
    public class DebugWrite: CodeActivity
    {
        [Description("The message output to the debug stream")]
        // Сообщение для вывода в поток отладки
        public InArgument<string> Message { get; set; }
        protected override void Execute(CodeActivityContext context)
        {
            Debug.WriteLine(Message.Get(context));
        }
    }
}

```

*Фрагмент кода 04\_CustomActivities*

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`, возможно, незнакомым образом. Одним из многочисленных изменений, которые были внесены в `Workflow`, является место сохранения данных состояния. В `Workflow 3.x` было принято использовать стандартные свойства .NET и сохранять данные действия внутри самого действия. Проблемой такого подхода было то, что подобное сохранение происходило, по сути, не видимым для механизма исполняющей среды рабочего потока образом и потому для сохранения рабочего потока требовалось производить бинарное сохранение для всех конструируемых действий, обеспечивая возможность точного восстановления данных.

В `Workflow 4` все данные сохраняются за пределами отдельных действий. Поэтому для получения значения аргумента производится запрос значения у контекста, а для установки значения аргумента — передача нового значения контексту. Благодаря этому, механизм рабочего потока может отслеживать изменения в состоянии, поскольку 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. 44.8).



*Рис. 44.8. Отображение дополнительной информации о свойстве в таблице свойств*

В данных условиях действие уже вполне пригодно для применения, однако есть еще несколько деталей, о которых следует позаботиться для того, чтобы сделать действие более дружелюбным к пользователю. Как было показано, например, при рассмотрении действия `Pick` ранее в этой главе, действие может иметь обязательные свойства, в случае не определения которых в области конструктора появляется глиф с сообщением об ошибке. Чтобы добавить такое поведение к собственному действию, код потребуется расширить.

## Проверка достоверности действий

Когда действие помещено на поверхность проектирования, визуальный конструктор просматривает два места для проверки достоверности. Простейший способ проверки достоверности действия предполагает добавление 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. 44.9.

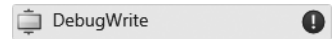


Рис. 44.9. Глиф с изображением восклицательного знака

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, понадобится определить значение для этого атрибута.

При наличии нескольких свойств, которые могут быть связаны, можно переопределить метод `CacheMetadata`, добавив дополнительный код для проверки достоверности действия. Этот 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. Кроме того, в нем можно добавить дополнительные сообщения (или предупреждения) об ошибках в процессе проверки правильности за счет вызова одной из переопределенных версий `AddValidationError` на объекте `CodeActivityMetadata`, передаваемом методу `CacheMetadata`.

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

## Визуальные конструкторы

При визуализации действия на экране с ним обычно ассоциируется визуальный конструктор. Задачей такого конструктора является обеспечение экранного представления этого действия, и в Workflow 4 это представление осуществляется с помощью XAML. Тем, кому ранее не приходилось работать с XAML, рекомендуется сначала почитать главу 35.

Среда времени проектирования для действия обычно создается в отдельной сборке из самого действия, так как во время выполнения необходимость в ней отпадает. В Visual Studio доступен проект типа `Activity Designer Library` (Библиотека конструктора действия), который является идеальной отправной точкой, поскольку при создании проекта с использованием такого шаблона в распоряжение предоставляется стандартный конструктор действия, который затем можно изменять любым нужным образом.

В XAML-коде для конструктора можно определять все, что угодно, в том числе анимационные эффекты. При построении пользовательских интерфейсов важна умеренность, потому рекомендуется сначала взглянуть на существующие действия и получить представление о том, что является нужным, а что нет.

Прежде всего, создадим простой конструктор и сопоставим его с действием `DebugWrite`. Ниже показан шаблон, создаваемый автоматически при добавлении в проект конструктора действия (или создании нового проекта типа `Activity Designer Library`). Этот код находится в решении `04_CustomActivities`.

```

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

```

Фрагмент кода 04\_CustomActivities

Как видите, в XAML-коде конструируется сетка, а также импортируются пространства имен, которые могут быть необходимы действию. Очевидно, что содержимого в шаблоне очень мало, поэтому для начала добавим к нему элементы управления `TextBlock` и `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 за счет применения 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 данной главе не рассматривается):

```

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

```

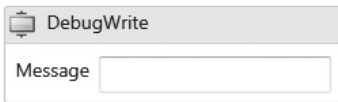



Рис. 44.10. Визуализация действия `DebugWrite` в Visual Studio

Здесь атрибут `[Designer]` применялся для определения связи между конструктором и действием. Использование строковой версии этого атрибута считается хорошим приемом, поскольку полностью исключает 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. 44.10.

Со свойством `Message` здесь связана одна проблема, поскольку в нем не отображается значение, определенное в таблице свойств, а при попытке установки этого значения за счет его ввода в текстовом поле генерируется исключение. Это объясняется тем, что в таком случае, по сути, предпринимается попытка привязать простое текстовое значение к типу `InArgument<string>`, а для подобной операции должна использоваться пара встроенных классов `Workflow 4` — `ExpressionTextBox` и `ArgumentToExpressionConverter`. Весь необходимый XAML-код для визуального конструктора теперь выглядит так, как показано ниже. 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>

```

Фрагмент кода 04\_CustomActivitiesFirst

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, а также 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.

## 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 дочернее действие 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.

Первым типом составного действия, создание которого рассматривается ниже, является действие "повторной попытки". Довольно часто необходимо попробовать 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;
    }
}
```

Нам требуется просто скопировать предыдущий код в качестве действия и на месте комментария поместить конкретное действие, которое должно выполняться. Может показаться удобным использовать для этого метод `Execute` специального действия. Однако есть и другой путь: с использованием других действий можно сделать намного больше. В данном случае необходимо создать специальное действие, содержащее заглушку, в которую конечный пользователь сможет помещать подлежащее повторной попытке действие, а также свойство для хранения максимального количества повторных попыток. Ниже приведен соответствующий код.



```
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()
            }
        }
    }
}
};
```

### Фрагмент кода 04 CustomActivities

Здесь сначала определяется свойство `Body` типа `Activity`: в нем будет размещаться действие, подлежащее повторному выполнению в цикле. Затем определяется свойство `RetryCount`: в нем будет указано, сколько раз должны предприниматься повторные попытки выполнения операции.

Это специальное действие унаследовано прямо от класса `Activity` и реализуется в виде функции. При выполнении рабочего потока, содержащего такое действие, будет, по сути, выполняться функция, примерно соответствующая приведенному ранее псевдокоду. В конструкторе создаются используемые действием локальные переменные, и затем в соответствии с псевдокодом строится ряд действий. Код этого примера тоже доступен в решении 04 `CustomActivities`.

Нетрудно догадаться, что рабочие потоки можно также создавать и без XAML-кода, тогда никакого поведения визуализации на стадии проектирования не будет (т.е. перетаскивать действия для генерации кода не удастся). Однако в случае, когда написание кода является предпочтительным вариантом, нет никакой причины не использовать его вместо XAML. После создания специального составного действия для него нужно определить визуальный конструктор. Здесь необходимо действие, имеющее метку-заполнитель, на место которой можно было бы перетаскивать другое действие. Если посмотреть на другие стандартные действия, легко обнаружить среди них несколько таких, которые обладают подобным поведением, как, например, действия `If` и `Pick`. В идеале, хотелось бы, что действие работало похожим на такие стандартные действия образом, поэтому настало время посмотреть на их реализации.

Применив утилиту Reflector для просмотра библиотек рабочих потоков, можно удостовериться в отсутствии какого-либо XAML-кода визуальных конструкторов. Все дело в том, что этот код скомпилирован в сборках как часть ресурсов. Утилиту Reflector можно использовать для просмотра этих ресурсов, но сначала для этого необходимо загрузить надстройку BAML Viewer (Инструмент для просмотра данных в формате BAML). Эта надстройка декомпилирует данные в двоичном формате BAML и производит из них текст. Поэтому

ее можно применять для просмотра используемого в стандартных действиях кода XAML. Для нахождения и загрузки этой надстройки выполните в Интернете поиск по ключевым словам “Reflector BAML Viewer”.

После установки надстройки BAML Viewer в программе Reflector достаточно загрузить в нее сборку `System.Activities.Presentation` и выбрать в меню Tools (Сервис) пункт BAML Viewer (Инструмент для просмотра данных в формате BAML). После этого появится список всех обнаруженных в текущей сборке ресурсов BAML, в котором можно выбрать образец и посмотреть, как в нем выглядит XAML-код.

Получив представление о том, как выглядит XAML во встроенных действиях, можно создать код для обеспечения визуализации действия `Retry` так, как показано на рис. 44.11.

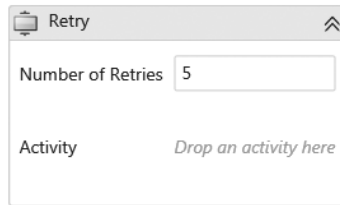


Рис. 44.11. Пример визуализации действия `Retry`

Ключевую роль в визуализации этого действия играет элемент управления `WorkflowItemPresenter`; именно он в XAML-коде отвечает за отображение метки-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"
    // Перетащите сюда желаемое действие
    Item="{Binding Path=ModelItem.Body, Mode=TwoWay}"
    Grid.Column="1" Grid.Row="1" Margin="2">
```

Здесь видно, что этот элемент привязывается к свойству `Body` действия `Retry`. Элемент имеет свойство `HintText`, в котором определен вспомогательный текст, который должен отображаться, когда еще не добавлено никакого дочернего действия. Также в приведенном XAML-коде определяются стили для отображения конструктора в развернутой и свернутой версии, что обеспечивает данному действию поведение, которым обладают встроенные действия. Весь код вместе с XAML-кодом этого примера доступен в решении `04_CustomActivities`.

## Рабочие потоки

До настоящего момента в этой главе речь шла в основном о действиях, но не о самих рабочих потоках. Рабочий поток представляет собой просто список действий и на самом деле является еще одним видом действия. Применение такой модели упрощает механизм исполняющей среды, поскольку в таком случае механизм должен знать, как выполнять объекты только одного типа — любые объекты, унаследованные от класса `Activity`.

Ранее был показан класс `WorkflowExecutor`, который может применяться для выполнения рабочего потока, и при этом упоминалось, что он является лишь одним из способов, с помощью которых можно выполнять рабочий поток. На самом деле для выполнения рабочих потоков доступно три варианта, каждый со своими возможностями. Прежде чем переходить к изучению других методов выполнения рабочих потоков, нужно подробнее ознакомиться с тем, что собой представляют аргументы и переменные.



## Аргументы и переменные

Рабочий поток можно считать программой, а одним из аспектов любого языка программирования является наличие возможности создавать переменные и передавать аргументы в и из программы. Разумеется, в Workflow 4 поддерживаются обе конструкции, и в настоящем разделе показано, как определять аргументы и переменные.

Для начала давайте предположим, что требуется создать рабочий поток для работы со страховыми полисами и, следовательно, что передаваемым ему аргументом должен быть номер страхового полиса. Для определения аргумента, передаваемого рабочему потоку, 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, как показано на рис. 44.12, в котором также можно создавать и собственные аргументы.

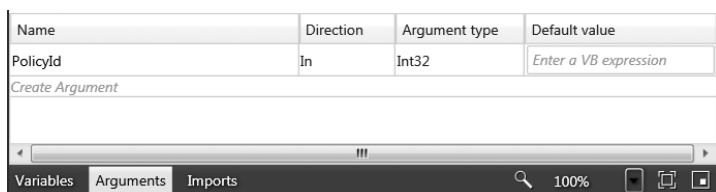


Рис. 44.12. Список аргументов, определенных для рабочего потока

Для определения собственного аргумента необходимо специфицировать его имя (Name), направление (Direction), которым может быть In, Out и InOut, и тип данных (Argument type). При желании для аргумента можно указать значение по умолчанию, которое должно использоваться, если никакого значения не предоставляется.

Направление аргумента служит для определения того, каким должен быть аргумент: входным, выходным либо тем и другим вместе (InOut).

В первом разделе настоящей главы было показано, как применять для выполнения рабочего потока класс WorkflowInvoker. У метода Invoke имеется несколько переопределенных версий, которые могут использоваться для передачи в рабочий поток различных аргументов. Передаются они в виде словаря пар “имя-значение”, в которых имя должно в точности соответствовать имени аргумента, с учетом регистра. Ниже приведен пример кода, служащего для передачи в рабочий поток значения PolicyId; весь код примера доступен в решении 05\_ArgsAndVars.

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

Фрагмент кода 05\_ArgsAndVars

В коде производится вызов рабочего потока с передачей именованному параметру значения PolicyId из словаря. Если в словаре предоставляется имя, для которого аргумента не существует, генерируется исключение ArgumentException. Если же значение для аргумента In не предоставлено, исключение генерироваться не будет. Такое поведение нас не устраивает, поскольку нужно, чтобы исключение выдавалось для любых аргументов In, которые не были определены, и чтобы оно не генерировалось в случае передачи слишком большого количества аргументов.

После завершения рабочего потока может понадобиться извлечь какие-то выходные аргументы. Для решения этой задачи метод WorkflowInvoker.Invoke имеет специальную переопределенную версию, которая возвращает словарь. Этот словарь будет содержать только аргументы Out или InOut.

Внутри рабочего потока можно определять переменные. Делать подобное в рабочих потоках XAML из Workflow 3.x было нелегко, однако в Workflow 4 сложности были устранены и теперь параметры можно определять в XAML без особых проблем.

Как и в любом языке программирования, с переменными в рабочих потоках 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 жизни 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`

Класс `WorkflowInvoker` удобен для обеспечения синхронного выполнения рабочих потоков, однако иногда возникает необходимость в создании длительных рабочих потоков, которые могли бы сохраняться в базе данных и затем возобновлять свое выполнение в какой-то момент в будущем. В этом случае удобно использовать класс `WorkflowApplication`.

Класс `WorkflowApplication` похож на класс `WorkflowRuntime`, который предлагался в Workflow 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();

```

*Фрагмент кода 06 `WorkflowApplication`*

Здесь сначала создается экземпляр приложения рабочего потока, который подключается к делегату `Completed` для установки события ручного сброса (`ManualResetEvent`). Затем вызывается метод `Run`, чтобы начать выполнять рабочий поток, а напоследок организуется ожидание срабатывания события.

Тут проявляется одно из главных отличий между классами `WorkflowExecutor` и `WorkflowApplication`: последний является асинхронным. При вызове метода `Run` в системе для выполнения рабочего потока будет использоваться пул потоков, а не вызывающий поток. Соответственно, нужен какой-то механизм синхронизации, чтобы исключить возможность завершения работы приложения, обслуживающего рабочий поток, раньше завершения самого рабочего потока.

Типичный продолжительный рабочий поток может много раз пребывать 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 периода отсрочки, а после получения этих входных данных обрабатывает их и погружается 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 потока. Для обеспечения такого поведения необходимо добавить в `WorkflowApplication` объект `InstanceStore`, а также внести другие незначительные изменения в предыдущий код. В каркасе доступна только одна реализация абстрактного класса `InstanceStore` — `SqlWorkflowInstanceStore`. Чтобы использовать этот

класс, сначала необходимо обзавестись базой данных; сценарии для базы данных по умолчанию можно найти в каталоге C:\Windows\Microsoft.NET\Framework\v4.0.21006\SQL\en. Обратите внимание, что номер версии может выглядеть по-другому.

В этом каталоге можно обнаружить множество файлов SQL, но двумя необходимыми из них являются `SqlWorkflowInstanceStoreSchema.sql` и `SqlWorkflowInstanceStoreLogic.sql`. Их можно запустить в отношении существующей базы данных либо создать новую базу данных; при этом можно использовать как полную версию SQL Server, так версию SQL Express.

После подготовки базы данных в обслуживающий код понадобится внести некоторые изменения. Первым делом, следует создать экземпляр `SqlWorkflowInstanceStore` и добавить его в приложение рабочего потока:

```

↓ 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();

```

Фрагмент кода 06 *WorkflowApplication*

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. Для определения того, что должно происходить с потоком в бездействующем состоянии, служит делегат `PersistableIdle`. По умолчанию никакие операции бездействующим потоком не выполняются. Можно также специфицировать обработчик `PersistableIdleAction.Persist`, который будет брать копию рабочего потока и сохранять ее в базе данных, оставляя рабочий поток в памяти, или обработчик `PersistableIdleAction.Unload`, который будет сохранять копию рабочего потока в базе данных и выгружать его из памяти.

Запрашивать сохранение рабочего потока можно также за счет применения действия `Persist`, а при разработке специального действия, если оно унаследовано от `NativeActivity` — за счет вызова метода `RequestPersist` класса `NativeActivityContext`.

Возникла следующая проблема: теперь, когда есть возможность выгружать рабочий поток из памяти и сохранять его в постоянном хранилище, как потом извлечь поток из хранилища и заставить его снова выполняться?

## Закладки

Традиционно закладка применяется для запоминания страницы в книге, чтобы потом иметь возможность продолжать чтение с места останова. В контексте рабочего потока закладка специфицирует место, с которого должно возобновляться выполнение этого рабочего потока, и закладки обычно применяются тогда, когда ожидается поступление входных данных извне.

Примером может служить написание приложения для обработки данных по расценкам на услуги страхования. Пользователь может создавать свои расценки в онлайн-режиме, с которыми, как не трудно догадаться, ассоциируется рабочий поток. Эти расценки действительны на протяжении 30 дней, поэтому должна быть предусмотрена соответствующая проверка.

Может также запрашиваться подтверждение снижения расценок и в случае не поступления такового 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. Перед выгрузкой в потоке должно указываться место, с которого будет возобновляться обработка, для чего и применяются закладки.

Для определения закладки необходимо создать специальное действие, унаследованное от `NativeActivity`. Затем можно создать саму закладку внутри метода `Execute` и указать в ней, с какой места должно продолжаться выполнение кода. В приведенном ниже примере определяется упрощенное действие `Task` с закладкой, которое 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);
    }
}
```

Фрагмент кода 06\_WorkflowApplication

При вызове `CreateBookmark` передается имя закладки и также функция обратного вызова. Этот обратный вызов будет происходить при возобновлении выполнения кода с закладки. В самом обратном вызове передается произвольный объект, в данном случае `Boolean`, поскольку решено, что каждая задача должна сообщать о том, успешно ли прошло ее выполнение. Эту информацию можно использовать для принятия соответствующего решения на последующих шагах в рабочем потоке. Тем не менее, рабочему потоку можно передавать любой другой объект, пусть даже сложного типа с множеством полей.

Написав такое действие, теперь необходимо изменить обслуживающий код таким образом, 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? Обслуживающему коду должно быть известно о существовании закладки.

Созданное выше действие `Task` на самом деле должно выполнять еще кое-какую работу, а именно — сообщать внешнему миру о том, что задача была создана. В производственной системе это обычно подразумевает сохранение соответствующей записи в таблице очереди и представление этой очереди персоналу центра обработки вызовов в виде списка задач.

Налаживание коммуникаций с хостом является темой следующего раздела.

## Расширения

Под расширением понимается просто класс или интерфейс, который добавляется в контекст времени выполнения для приложения рабочего потока. В Workflow 3.x расши-

рения назывались службами, однако из-за возникающей путаницы со службами WCF в Workflow 4 они были переименованы в “расширения”.

Обычно для расширений определяется некоторый интерфейс и строится его реализация времени выполнения. Действия будут просто обращаться к этому интерфейсу, что позволит при необходимости изменять реализации. Хорошим примером может служить создание расширения для отправки сообщения по электронной почте. В этом случае можно создать действие `SendEmail` с вызовом расширения внутри его метода `Execute`. Затем для фактической отправки сообщения во время выполнения можно определить расширение, основанное на использовании протокола SMTP, или расширение, применяющее Outlook из Exchange Server. Изменять действие для работы с каким-то другим поставщиком услуг отправки электронной почты не понадобится — нужно просто подключить новый поставщик через конфигурационный файл приложения.

Для рассматриваемого примера с задачей 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 отношению информации в базу данных, чтобы впоследствии предоставить пользователю очередь задач. Для определения такого расширения применяется следующий интерфейс:

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

Теперь можно обновить действие `Task` так, чтобы оно уведомляло расширение о своем выполнении, изменив метод `Execute` следующим образом:

```
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`, и затем в коде вызывается метод `ExecuteTask`. Класс `WorkflowApplication` поддерживает коллекцию расширений, поэтому в нем можно создать класс, реализующий такой интерфейс-расширение, после чего использовать его для составления списка задач. Далее можно перейти к созданию и выполнению 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 следом за ним действие `If`, которое предусматривает вывод соответствующего сообщения в зависимости от того, разрешает или отклоняет пользователь выполнение задачи.

## Объединение всего вместе

Теперь уже известно, как запускать, сохранять и выгружать рабочий поток, а также доставлять в него события посредством закладок, а осталось узнать еще 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 извлекать из объекта `WorkflowApplication`, обращаясь к свойству `Id`. Следовательно, в псевдокоде рабочий поток, обслуживающий приложение, должен выглядеть следующим образом:

```

WorkflowApplication app = BuildApplication();
Guid id = app.Id;
app.Run();

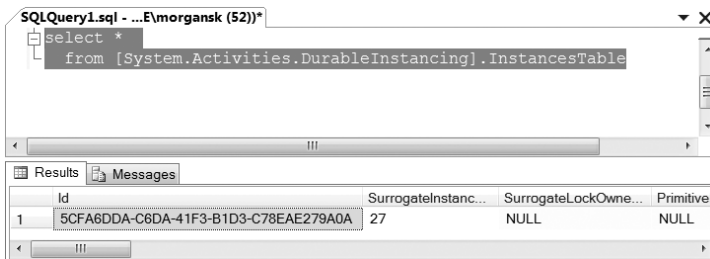
// Ожидание до тех пор, пока будет создана задача, и
// затем выполнение повторной загрузки рабочего потока
app = BuildApplication();
app.Load(id);
app.ResumeBookmark()

```

Предоставленный пример кода немного сложнее предыдущего, поскольку также включает в себя реализацию интерфейса `ITaskExtension`, но в целом следует той же схеме, что ранее. Как не трудно заметить, метод `BuildApplication` здесь вызывается два раза. 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, по сути, 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 функции `ResumeBookmark` работа этого экземпляра возобновляется.

После запуска этого примера на экране появится приглашение. Пока это приглашение остается на экране, рабочий поток сохраняется и выгружается из памяти, в чем легко убедиться, запустив `SQL Server Management Studio` и выполнив команду, показанную на рис. 44.13.



**Рис. 44.13.** Команда, позволяющая удостовериться в сохранении и выгрузке рабочего потока

Экземпляры рабочих потоков сохраняются в таблице `InstancesTable` схемы `System.Activities.DurableInstancing`. На рис. 44.13 можно видеть 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 значение `DeleteAll`. Это значение гарантирует, что любые данные, сохраняемые для экземпляра того или иного рабочего потока, будут сразу же удалены после его завершения. Это имеет смысл, поскольку обычно после завершения рабочего потока его внутренние данные совершенно не нужны. Однако при желании значение этой опции можно изменить, установив во время определения экземпляра для действия, отвечающего за поведение после завершения экземпляра, значение `DeleteNothing`.

Если теперь продолжить выполнение тестового приложения и снова запустить команду, показанную на рис. 44.13, можно увидеть, что экземпляр рабочего потока был удален.



идентифицирует данный экземпляр рабочего потока и применяется при выполнении обратного вызова к этому экземпляру в других операциях.

После выполнения этих двух первоначальных действий в рабочем потоке запускается действие While, внутри которого имеется действие Pick с двумя ветвями. В первой ветви содержится действие Receive с операцией UploadRoomInformation, а во второй — действие Receive с операцией DetailsComplete.

Код, показанный в следующем фрагменте, используется для обслуживания рабочего потока. В нем применяется класс WorkflowServiceHost, который очень похож на доступный в WCF класс ServiceHost.

```
⬇ string baseAddress = "http://localhost:8080/PropertyService";
using (WorkflowServiceHost host =
    new WorkflowServiceHost(GetPropertyWorkflow(), new Uri(baseAddress)))
{
    host.AddServiceEndpoint(XName.Get("IProperty", ns),
        new BasicHttpBinding(), baseAddress);
    ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
    smb.HttpGetEnabled = true;
    host.Description.Behaviors.Add(smb);
    try
    {
        host.Open();
        Console.WriteLine("Service host is open for business...");
        // Хост службы открыт
        Console.ReadLine();
    }
    finally
    {
        host.Close();
    }
}
```

Фрагмент кода 07 *WorkflowsAsServices*

В коде сначала создается экземпляр класса WorkflowServiceHost, и предоставляемый рабочий поток поставляется за счет вызова метода GetPropertyWorkflow. Рабочий поток может поставляться как с применением соответствующего кода, как в этом примере, так и с помощью XAML. Затем добавляется конечная точка и задается базовая HTTP-привязка, а также разрешается публикация метаданных, чтобы в этой конечной точке клиент мог указывать svcutil.exe загружать определение службы. После этого просто производится вызов метода Open на классе хоста службы и подготовка к получению входящих вызовов.

### **Гарантирование возможности создания рабочего потока**

Первым действием в рабочем потоке является Receive, и оно было определено со значением True в свойстве CanCreateInstance. Это важный момент, поскольку тем самым хост уведомляется, что в случае обращения к этой операции вызывающей стороной на основе этого вызова должен создаваться новый экземпляр рабочего потока. “За кулисами” WorkflowServiceHost зондирует определение рабочего потока, отыскивая в нем действия Receive, и создает на их основании контракт службы. Кроме того, он смотрит, нет ли среди них таких, для свойства CanCreateInstance у которых установлено значение True, поскольку это позволяет ему узнать, что необходимо делать при получении сообщения.

Пример содержит клиентское приложение, в котором сначала создается новый экземпляр рабочего потока вызовом UploadPropertyInformation, затем производится выгрузка информации о трех комнатах с помощью операции UploadRoomInformation и, наконец, вызывается DetailsComplete. Это приводит к установке флага Finished, используемого в действии While, и завершению работы данного рабочего потока.



Код рабочего потока выглядит довольно сложно, поскольку в нем производится очень много скрытой работы. Однако важно понять, как он работает, поэтому далее рассматриваются некоторые его части.

Определение первого действия внутри рабочего потока выглядит следующим образом:

```
Variable<string> address = new Variable<string>();
Variable<string> owner = new Variable<string>();
Variable<double> askingPrice = new Variable<double>();

// Первоначальное действие Receive - это запускает рабочий поток
Receive receive = new Receive
{
    CanCreateInstance = true,
    OperationName = "UploadPropertyInformation",
    ServiceContractName = XName.Get("IProperty", ns),
    Content = new ReceiveParametersContent
    {
        Parameters =
        {
            {"address", new OutArgument<string>(address)},
            {"owner", new OutArgument<string>(owner)},
            {"askingPrice", new OutArgument<double>(askingPrice)}
        }
    }
};
```

Это действие Receive имеет флаг CanCreateInstance, установленный в True, и в нем определены имя операции и имя контракта службы. Имя контракта службы (ServiceContractName) включает имя пространства имен (в данном примере установлено в `http://pro-csharp/`), что позволяет уникальным образом идентифицировать данную службу. В свойстве Content определено, передача каких данных ожидается от клиента при вызове данной операции; в данном случае ожидается передача данных об адресе имени владельца и запрашиваемой им цены. Операция службы определяется здесь полностью в коде, и экспортируемые метаданные используют этот код для создания определения операции.

Аргументы, получаемые из этой операции, привязываются к определяемым в рабочем потоке переменным. Именно так эти аргументы передаются из внешнего мира в рабочий поток, обслуживаемый WCF.

Внутри рабочего потока создается уникальный идентификатор, который будет использоваться при выполнении обратного вызова этого экземпляра рабочего потока. Идентификатор присваивается специальной переменной; как видно в показанном ниже коде, это делается с помощью действия Assign.

```
return new Sequence
{
    Variables = { propertyId, operationHandle, finished, address, owner, askingPrice },
    Activities =
    {
        receive,
        new WriteLine { Text = "Assigning a unique ID" },
        // Присваивание уникального идентификатора
        new Assign<Guid>
        {
            To = new OutArgument<Guid>(propertyId),
            Value = new InArgument<Guid>(Guid.NewGuid())
        },
        new SendReply
        {

```

```
Request = receive,
Content = SendContent.Create
(new InArgument<Guid>(env => propertyId.Get(env))),
CorrelationInitializers =
{
    new QueryCorrelationInitializer
    {
        CorrelationHandle = operationHandle,
        MessageQuerySet = extractGuid
    }
}
// Для простоты другие действия опущены
};
```

Далее используется действие `SendReply` для отправки вызывающей стороне ответа с определенным в рабочем коде значением `Guid`. Эта часть кода является довольно сложной.

Действие `SendReply` связывается с начальным действием `Receive` за счет присваивания его свойству `Request` экземпляра действия `Receive`. Значение, которое должно возвращаться из операции, определяется с помощью свойства `Content`. Существуют три вида ответов, которые могут возвращаться из операции: дискретное значение, такое как используемое здесь `Guid`, набор параметров, определенных в словаре пар “имя-значение” или сообщение. `SendContent` — это вспомогательный класс, который создает надлежащий объект автоматически.

Последняя часть кода самая сложная. Свойство `CorrelationInitializers` используется для извлечения маркера корреляции из текущего сообщения и его применения в последующих вызовах рабочего потока для обозначения этого потока уникальным образом. Свойство `MessageQuerySet` служит извлечения и определяется для данного исходящего сообщения так, как показано ниже. После определения запрос связывается со значением дескриптора. Этот дескриптор может использоваться любым другим действием, желающим быть частью той же самой группы корреляции.

```
MessageQuerySet extractGuid = new MessageQuerySet
{
    { "PropertyId",
      new XPathMessageQuery( "sm:body()/ser:guid", messageContext ) }
};
```

В данном запросе с помощью выражения `XPath` данные извлекаются из сообщения, отправляемого обратно клиенту. Здесь оно предусматривает чтения элемента `body` и поиск внутри него элемента `guid`.

### **Ожидание событий в действии *Pick***

Итак, мы создали новый экземпляр потока и отправили вызывающей стороне в ответ уникальный идентификатор операции. Следующим этапом является ожидание поступления дальнейших данных от клиента, которое реализуется с помощью двух добавочных действий `Receive`. Первое из них соответствует операции `UploadRoomInformation`.

```
Variable<string> roomName = new Variable<string>();
Variable<double> width = new Variable<double>();
Variable<double> depth = new Variable<double>();

// Получение информации о комнате
Receive receiveRoomInfo = new Receive
{
    OperationName = "UploadRoomInformation",
```

```

ServiceContractName = XName.Get("IProperty", ns),
CorrelatesWith = operationHandle,
CorrelatesOn = extractGuidFromUploadRoomInformation,
Content = new ReceiveParametersContent
{
    Parameters =
    {
        {"propertyId", new OutArgument<Guid>()},
        {"roomName", new OutArgument<string>(roomName)},
        {"width", new OutArgument<double>(width)},
        {"depth", new OutArgument<double>(depth)},
    }
}
};

```

Определенные первоначально переменные используются для записи данных, передаваемых от клиента рабочему потоку. Затем в действии `Receive` снова указываются имя операции (`OperationName`) и имя контракта службы (`ServiceContractName`), чтобы обеспечить возможность генерации метаданных для данной операции. Значение `CorrelatesWith` играет критически важную роль, поскольку связывает различные операции рабочего потока вместе. В действии `SendReply` создан соответствующий дескриптор, чтобы иметь возможность ссылаться на него в последующих операциях. Без этого дескриптора невозможно выяснить, для какого экземпляра рабочего потока предназначено входящее сообщение.

Свойство `CorrelatesOn` определяет, что должно извлекаться из входящего сообщения для проверки, предназначено ли это сообщение данному экземпляру рабочего потока. Далее снова используется свойство `MessageQuerySet`, которое в этом случае извлекает уникальный идентификатор из входящего сообщения:

```

MessageQuerySet extractGuidFromUploadRoomInformation = new MessageQuerySet
{
    { "PropertyId",
      new XPathMessageQuery
        ( @"sm:body()/local:UploadRoomInformation/local:propertyId", messageContext ) }
};

```

Здесь выражение `XPath` производит поиск внутри вызова `UploadRoomInformation` элемента `propertyId`.

В последней части данного действия `Receive` определено, что должно происходить с входящими параметрами, передаваемыми операции службы. Тут ожидается получение уникального идентификатора `propertyId`, имени комнаты и сведений о ее размерах, и указывается, что эти данные должны извлекаться в переменные и просто выводятся на консоль. В реальном приложении эти данные, скорее всего, использовались бы для обновления базы данных или чего-то подобного.

Последнее действие `Receive` применяется для ожидания поступления от клиента сообщения `DetailsComplete`. В нем все делается по примерно той же схеме, что и в остальных действиях `Receive`, т.е. определяется имя контракта службы и имя операции, используется тот же дескриптор корреляции, что и в первоначальных действиях `Send` и `Receive` для операции `UploadRoomInformation`, и применяется свойство `MessageQuerySet` для извлечения из входящего сообщения уникального идентификатора `propertyId`.

После всех этих действий в остальной части кода рабочего потока идет цикл `While`, в котором содержится действие `Pick` со своими ветвями:

```

new While
{
    Condition = ExpressionServices.Convert<bool>
        (env => !finished.Get(env)),
    Body = new Pick
    {

```

```
Branches =
{
    new PickBranch
    {
        Variables = { roomName, width, depth },
        Trigger = receiveRoomInfo,
        Action = new WriteLine { Text = "Room Info Received"},
                                     // Информация о комнате получена
    },
    new PickBranch
    {
        Trigger = receiveDetailsComplete,
        Action = new Sequence
        {
            Activities =
            {
                new Assign<bool>
                {
                    To = new OutArgument<bool>(finished),
                    Value = new InArgument<bool>(true)
                },
                new WriteLine { Text = "Finished" }
                                     // Готово
            }
        }
    }
}
```

Здесь в цикле `While` определяется свойство `Condition`, в котором вычисляется значение переменной `finished`. Это свойство типа `Activity<bool>`, поэтому для преобразования выражения переменной в действие применяется вспомогательный класс `ExpressionServices`.

Далее идет действие `Pick`, содержащее две ветви. В каждой из этих ветвей ожидается срабатывание события, указанного в `Trigger`, для выполнения операции, заданной в `Action`. Построение рабочих потоков в коде является несколько длительным подходом, но при изучении `Workflow 4` он позволит лучше разобраться в том, что происходит, чем подход с созданием рабочих потоков с помощью `XAML`.

## Размещение визуального конструктора

Чаще всего самое лучшее припасают напоследок. Мы решили не нарушать эту традицию и поступили аналогично в этой главе. Визуальный конструктор `Workflow`, используемый в `Visual Studio`, можно также размещать в собственном приложении; это позволяет конечным пользователям создавать свои рабочие потоки, не имея копии `Visual Studio`. Пожалуй, это самая лучшая возможность `Workflow 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. `Windows Workflow` позволяет конечным пользователям настраивать приложение, просто перетаскивая действия на поверхность конструктора.

Изменение места размещения конструктора в `Workflow 3.x` было делом непростым. В `Workflow 4` оно превратилось практически в тривиальную задачу. Сам конструктор представляет собой элемент управления `WPF`, поэтому в рассматриваемом примере в качестве главного приложения будет использоваться проект `WPF`. Полный код примера доступен в решении `08_DesignerRehosting`.

В первую очередь понадобится включить сборки рабочих потоков, а затем определить XAML-код для визуализации главного окна. При определении пользовательских интерфейсов мы всегда используем шаблон Model-View-ViewModel (MVVM), поскольку это упрощает процесс кодирования, а также обеспечивает возможность применения другого XAML к той же модели представления. Ниже показан XAML-код для визуализации главного окна.

```

❶ <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>

```

Фрагмент кода 08\_DesignerRehosting

Как видите, компоновка очень проста и предусматривает размещение в главном окне лишь меню и сетки с метками 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` включает в себя свойства для каждого из основных элементов пользовательского интерфейса: `Toolbox` (Панель инструментов), `Designer` (Конструктор) и `Property Grid` (Таблица свойств), а также свойства для всех команд: `New` (Создать), `Save` (Сохранить) и `Exit` (Выход).

```

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 PropertyInspectorView
{
    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 `INotifyPropertyChanged`. Этот класс происходит из ряда фрагментов, написанных Джошем Твистом (Josh Twist) и доступен на сайте [www.thejoyofcode.com](http://www.thejoyofcode.com).

Далее в конструкторе гарантируется регистрация метаданных для всех встроенных действий — без этого вызова никакие из конструкторов стандартных типов не будут появляться в пользовательском интерфейсе. Затем в методе `InitializeViewModel` создается экземпляр конструктора `Workflow` и производится загрузка действия в него. Класс `WorkflowDesigner` интересен тем, что после загрузки одного рабочего потока в него загружать еще один нельзя, поэтому здесь экземпляр этого класса создается заново 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 конструктора `Workflow`.

Следующей частью пользовательского интерфейса, которую требуется создать, является панель инструментов. В `Workflow 3.x` такой элемент управления нужно было создавать самостоятельно, а в `Workflow 4` для этого предлагается элемент управления `ToolboxControl`, который очень прост в применении.

```
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` применяется для упрощения кода, необходимого для добавления заданного действия в панель инструментов.

После написания всего этого кода получается почти готовое работающее приложение. Все, что останется сделать — это связать `ViewModel` с 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. На рис. 44.15 показано, как будет выглядеть изображение на экране в случае запуска данного приложения на этом этапе.

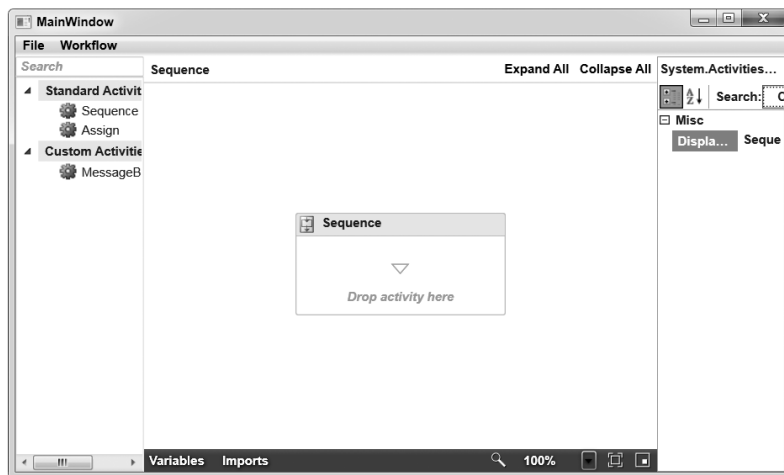


Рис. 44.15. Пробный запуск демонстрационного приложения

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(used =>
        {
            InitializeViewModel(new Sequence());
        });
    }
}
```

Эта команда привязывается к элементу меню **New**, 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` выглядит 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 `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 конструктор. Кроме того, имеется команда `Save`, которая вызывает метод `WorkflowDesigner.Save` для сохранения XAML-данных рабочего потока на диске.

В последнем разделе кода модели представления реализована команда `Run`. От возможности проектировать рабочие потоки без их выполнения мало толку, поэтому в модель представления включена и такая возможность. Все выглядит довольно тривиально: конструктор включает свойство `Text`, которое обеспечивает XAML-представление 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");
}
```

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 при невозможности получения к ним доступа; как раз для этого предназначена вторая лямбда-функция в `DelegateCommand`. Эта функция возвращает значение `HasErrors`, представляющее собой булевское свойство, которое было добавлено в модель представления. Это свойство показывает, не были ли внутри рабочего потока обнаружены какие-то ошибки проверки достоверности, поскольку в модели представления реализован интерфейс `IVValidationErrorService`, который уведомляется при каждом изменении правильного состояния рабочего потока.

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

## Резюме

Технология Windows Workflow приведет к радикальным изменениям в подходе к конструированию приложений. Уже сейчас она позволяет представлять сложные части приложения в виде действий и предоставлять пользователям возможность менять предлагаемый системой способ обработки, просто перетаскивая эти действия в рабочий поток.

Нет практически ни одного приложения, к которому нельзя было бы применить рабочий поток, начиная с простейших утилит командной строки и заканчивая сложнейшими системами со многими сотнями модулей. Там где раньше требовалось привлечь разработчика для написания специального модуля расширения к системе, теперь можно предложить простой и расширяемый механизм настройки, которым сможет пользоваться практически любой. Производителям приложений раньше нужно было самим предоставлять специальные действия для взаимодействия с их системами, а также код, способный вызывать рабочий поток или потоки. Однако теперь они могут оставлять выбор за своими потребителями и позволить тем самим определять, что должно происходить при возникновении в их приложении того или иного события.

Версия Workflow 3.x сегодня практически повсеместно заменена версией Workflow 4, поэтому тем, кто планирует использовать рабочие потоки впервые, рекомендуется начать с новой версии и вообще не заниматься Workflow 3.x.