WINDOWS INSTALLER XML

СОЗДАНИЕ ПРОГРАММЫ УСТАНОВКИ В VISUAL STUDIO

Евгений Воднев

На сегодняшний день наличие профессионально выполненной программы установки является непременным атрибутом успешного программного продукта. На рынке присутствует ряд средств, предназначенных для решения этой задачи, но посвященной данному вопросу документации крайне мало

В данной книге рассматривается использование пакета Windows Installer XML для создания широкого круга программ установки для операционных систем Windows

Версия документа: 2011-05-21

Содержание

Введение

Зачем нужна программа установки

Зачем вообще нужна программа установки? Чтобы ответить на этот вопрос, давайте подумаем: а когда же программа установки не нужна?

- выполняется развертывание веб-приложения. Действительно, если продукт развертывается в количестве не более чем нескольких экземпляров на сервере или ферме серверов, то, возможно, нет необходимости тратить время и усилия на создание полноценной программы установки.
- приложение настолько простое, что допускает развертывание в стиле XCOPY. Если для обеспечения работоспособности достаточно скопировать каталог с необходимыми файлами на целевую машину, то, пожалуй, необходимость в установочном пакете также отсутствует.
- для установки приложения уже используется технология ClickOnce. Тогда вам доступны такие преимущества, как проверка наличия обновлений на сервере и их автоматическая установка.

Но если продукт будет устанавливаться значительным количеством клиентов и при этом требует таких функций, как:

- поддержки транзакционной установки с возможностью отката при неудаче;
- поддержки централизованного развертывания на основе групповых политик;
- установки .NET сборок в GAC;
- установки COM-компонентов или служб Windows;
- использования возможностей, предоставляемых операционной системой – очередей сообщений, счетчиков производительности, управления пользователями;
- осуществления настройки прав доступа и правил для встроенного сетевого экрана;
- развертывания и настройки баз данных;
- наличия развитого пользовательского интерфейса.

Тогда вам необходимо воспользоваться возможностями, предоставляемыми технологией Windows Installer. На сегодняшний день с ее помощью выполняется установка любого серьезного программного продукта для операционных систем семейства Windows. Тем не менее, сам Windows Installer достаточно сложен для освоения, а встроенные средства не обеспечивают высокой гибкости и удобства использования. Поэтому есть смысл внимательнее взглянуть на сопутствующие технологии, повышающие уровень абстракции и упрощающие работу с данной технологией, такие, как Windows Installer XML.

Материалы по Windows Installer XML крайне разрознены. Они представлены в справочной системе, разбросаны по десятку блогов авторов и участников сообщества, представлены в нескольких статьях и руководствах. Кроме того, лишь очень небольшое количество действительно полезных материалов доступно на русском языке. Эта книга – попытка обобщить указанные

материалы в одном месте, объединив их с практическим опытом автора и дополнив примерами. Я не ставил своей целью перевод справочной системы по продукту, а попытался акцентировать внимание на практических вопросах, стараясь сделать книгу одновременно интересной – насколько может быть интересно техническое руководство – и полезной широкому кругу разработчиков. Какой она получилась – судить вам.

Структура книги

В книге семь глав.

В первой главе мы начнем с рассмотрения базовых понятий, лежащих в основе Windows Installer и WiX, после чего создадим с нуля простое решение в Visual Studio. Все действия описываются достаточно подробно, поэтому, если что-либо в последующих главах работает не так, как ожидается, попробуйте вернуться к данному разделу.

Во второй главе мы подробно обсудим создание программ установки с использованием WiX и IDE Visual Studio. Это раздел может быть полезен даже в том случае, если у вас уже есть опыт практического использования инструментов из комплекта Windows Installer XML из командной строки, так как среда разработки позволяет упростить ряд часто выполняемых задач. В этом же разделе описываются вопросы интеграции WiX в Team Foundation Server.

Третья глава посвящена реализации базовой функциональности программы установки: как определить структуру каталогов, скопировать файлы, записать значения ключей реестра и очистить при удалении, описать наборы, использовать свойства и проверить условия. Здесь находится описание наиболее часто используемых возможностей.

В четвертой главе рассказывается о том, как использовать некоторые расширения из числа существующих в WiX. Они позволяют решать широкий круг стандартных задач: редактировать XML файлы, управлять учетными записями пользователей, работать с базами данных и другие. Увы, но часть поставляемых с Windows Installer XML расширений практически не документирована или находится в состоянии разработки, поэтому здесь они не описаны.

Пятая глава познакомит вас с процессом создания пользовательского интерфейса: использования стандартных наборов диалогов, создания собственных диалоговых окон и их подключения. Подробно рассмотрены элементы управления, предоставляемые Windows Installer.

В шестой главе рассмотрены возможности работы программы установки в различных режимах, а также общие вопросы организации последовательностей. Описано использование стандартных операций. Здесь же описывается создание собственных операций на управляемом коде и их встраивание в последовательность установки.

Седьмая глава будет полезна разработчикам с опытом. Здесь рассматриваются вопросы создания загрузчика (bootstrapper), выпуска обновлений и отладки установочных пакетов, для чего мы чуть глубже изучим структуру базы данных пакетов msi и научимся читать содержимое отладочного вывода. Также мы рассмотрим вопрос декомпиляции существующих пакетов и повторного использования их содержимого.

Примеры

Мне пришлось серьезно задуматься над тем, как лучше организовать приводимые в книге примеры. С одной стороны, возможно пошаговое изложение материала. Это будет полезно тем, кто недавно начал использовать Windows Installer XML, но заставит зевать разработчиков с опытом. С другой стороны, можно углубиться в детали, сопровождая текст фрагментами кода. Однако тогда для создания собственной полноценной программы установки придется предварительно проштудировать большинство глав.

Я постарался найти компромисс между этими двумя крайностями: изложить материал в форме и объеме, который будет полезен разработчику с опытом, но одновременно позволит быстро начать использовать WiX практически «с нуля». Для этого пример в первой главе описывается очень подробно, позволяя создать простой, но работоспособный пакет даже без наличия навыков работы с IDE Visual Studio. В третьей главе создается более функциональный пример, но уже без деталей, описываемых в главе 1. Остальные главы посвящены рассмотрению широкого круга вопросов, и создание единого примера для них было бы слишком громоздким. Поэтому, раскрывая тему, я привожу фрагменты XML-разметки, описывая функционал, добавляемый тем или иным узлом. Это позволит читателю, скопировав и внеся минимальные изменения, добавить аналогичные функции в свою программу установки. Эти же возможности можно добавлять и в пример из главы 3. Таким образом, я постарался сочетать доступность изложения материала с практической применимостью для решения даже достаточно сложных задач.

Некоторые описания, предлагаемые в книге, являются решением типовых задач. Иногда в них одновременно используется материал из нескольких тем. Поскольку размещение в конкретной главе в данном случае достаточно условное, я постарался оформить такие ситуации в виде подразделов, чтобы их было можно найти по оглавлению.

Отличия технологий Windows Installer и ClickOnce

Технология ClickOnce предоставляет альтернативный вариант развертывания приложений, делая установку настольных приложений достаточно простой и удобной. Подготовка приложения к развертыванию заключается в публикации файлов на сервере, откуда конечный пользователь устанавливает его несколькими щелчками мыши. Кроме прочего, ClickOnce предоставляет возможность автоматической установки обновлений по мере их появления на сервере – то, что на сегодняшний день отсутствует в Windows Installer. Что же мешает признать последний устаревшим и полностью заменить его новой, более удобной технологией? Все дело в том, что ClickOnce не позволяет выполнять сложных операций, накладывая ряд существенных ограничений на процесс установки – он в большей степени подходит для развертывания приложений в стиле XCOPY. В таблице 1 приведено сравнение основных возможностей технологий Windows Installer и ClickOnce.

Таблица 1. Сравнение возможностей Windows Installer и ClickOnce.

Возможность	ClickOnce	Windows Installer
Установка файлов	●	●
Создание ярлыков	●	●

Назначение обработчиков для расширений файлов	●	●
Установка служб Windows		●
Установка сборок в GAC		●
Управление источниками данных ODBC		●
Управление компонентами COM+		●
Создание записей в реестре		●
Установка элементов при первом обращении и возможность запуска с источника		●
Восстановление исходного состояния (при повреждении)		●
Назначение прав доступа к файлам, каталогам и реестру		●
Развитый пользовательский интерфейс программы установки		●
Возможность установки для всех пользователей		●
Дополнительные операции при установке/удалении		●
Проверка условий при запуске и в процессе установки		●
Автоматическое обновление при запуске или по расписанию	●	
Принудительное обновление	●	
Изолированный запуск приложения (так называемая «песочница» - Sandbox)	●	
Скачивание и установка сборок по мере необходимости	●	
Возможность отката к предыдущим версиям	●	

Таким образом, ClickOnce предоставляет те же базовые функции, что и Windows Installer, но выигрывает за счет удобства обновления продуктов. Чтобы обойти это ограничение Windows Installer, в главе 7, в разделе, посвященном подготовке и выпуску обновлений, перечисляются некоторые решения, позволяющие встроить поиск и установку обновлений в ваш продукт.

Почему именно Windows Installer XML

Чем же так хорош Windows Installer XML, что есть смысл использовать именно его? Ведь он не обладает графическим интерфейсом, как альтернативные продукты. В то время как даже входящий в любую поставку Visual Studio шаблон проекта Setup Project может похвастаться наличием графического интерфейса, позволяющего создать простой установочный пакет буквально несколькими щелчками мыши. Верно. Однако если попробовать создать с помощью Setup Project более сложную программу установки, мы столкнемся с рядом существенных ограничений:

- невозможность выборочной установки элементов;
- ограниченные расширяемость и возможности по настройке интерфейса;
- зависимость от графического интерфейса и, как следствие, скромные возможности по автоматизации;
- невозможность использования преимуществ контроля версий.

Может быть, есть смысл воспользоваться продуктами от сторонних поставщиков? Возможно. Однако для создания полноценной программы установки все равно придется потратить сравнимое время на изучение конкретного продукта. Более того, даже развитый интерфейс

пользователя не избавит разработчика от необходимости знакомства с основными понятиями Windows Installer. Также нельзя забывать о времени, которое потребуется разработчикам для переноса новых функций в инструмент от стороннего производителя. В то же время Windows Installer XML, являясь относительно тонкой оболочкой над Windows Installer, предоставляет весь его функционал. Кроме того, сложно найти продукт с более подходящей ценой – ведь WiX совершенно бесплатен.

Строго говоря, Windows Installer XML представляет собой набор утилит командной строки и сопутствующие библиотеки, что позволяет создавать программы установки в любом текстовом редакторе. Однако извлечь максимум преимуществ от использования пакета позволяет его интеграция в Visual Studio, упрощающая и ускоряющая процесс разработки. В этой книге я описываю работу с WiX именно с использованием IDE Visual Studio, а отдельные утилиты рассматриваю только тогда, когда интегрированные в среду средства не обеспечивают искомого функционала.

Глава 1. Основы и простой пример

Структура установочного пакета Windows Installer

Для создания программ установки важно иметь общее представление о структуре msi-пакета и порядке действий, выполняемых в процессе его работы. Сам пакет представляет собой базу данных, в которой хранятся ресурсы и описание последовательности действий, выполняемых в процессе их установки. При запуске установочного пакета служба Windows Installer транзакционно выполняет все необходимые операции, позволяя разработчику сосредоточиться на том, что необходимо выполнить, а не на том, как это сделать.

Поскольку программа установки создается в первую очередь для копирования на клиентский компьютер различных ресурсов: файлов, каталогов, ярлыков, записей в реестр, поэтому необходимо понимать, как ресурсы размещаются в пакете. Все ресурсы должны быть размещаться внутри компонентов. В свою очередь, компонент, являясь оберткой для ресурса, управляется службой Windows Installer, контролирующей целостность установленного пакета. Из компонентов образуются наборы (Features), объединяющие связанный функционал. Так, например, в приложении в отдельные наборы могут быть выделены обязательные файлы и компоненты, дополнительные инструменты, файлы справки и т.п. Наборы могут вкладываться друг в друга, образуя иерархию. Число наборов в установочном пакете нечасто превышает десяток, в то же время количество компонентов в нем может исчисляться сотнями.

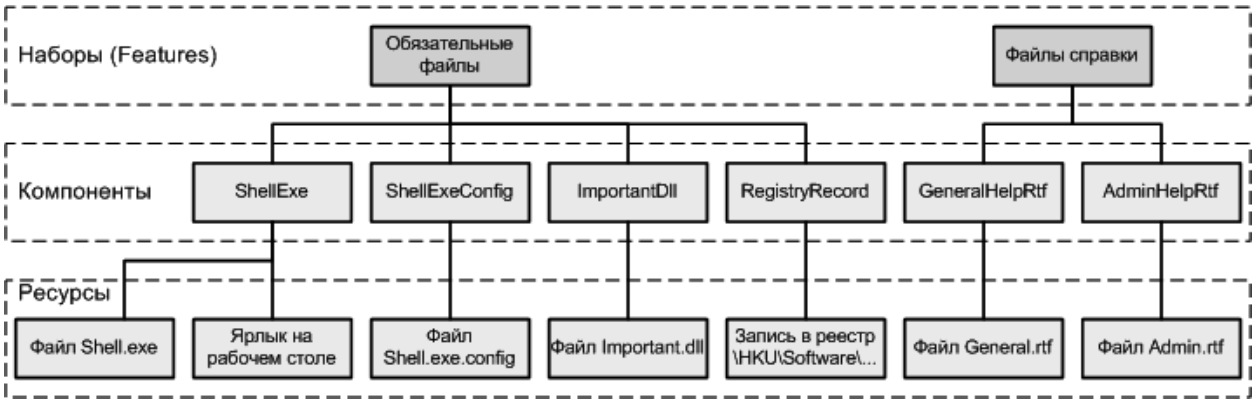

Рисунок 1.1 Иерархия «ресурс – компонент – набор».

На рисунке 1.1 показан пример с двумя наборами; предполагается, что файлы справки могут быть установлены пользователем по желанию. Как правило, рекомендуется создавать по одному компоненту на файл, однако нередко целесообразно группировать связанные элементы, например, исполняемые файлы и ярлыки для них.

Однако мало описать структуру ресурсов – для их установки необходимо выполнить множество операций. Порядок производимых действий описывается последовательностями, о которых рассказывается в главе 6; сейчас достаточно понимать, что операции выполняются поочередно, причем часть операций является обязательной и порядок их запуска фиксирован, в то время как остальные могут настраиваться более гибко, в зависимости от потребностей разработчика. Так,

любая программа установки начинается с поиска установленных продуктов, затем проверяется наличие свободного места на диске, после чего производится копирование ресурсов. К доступным, но не используемым по умолчанию операциям, можно отнести, например, ForceReboot – требование выполнения перезагрузки в процессе установки. Или DisableRollback, отключающую возможность отката для всех последующих операций.

Если вы создаете простую программу установки – вам достаточно понимать принципы организации ресурсов, а встроенные средства скроют от вас детали работы с последовательностями. Но для построения сложных пакетов потребуется разобраться с деталями их устройства – для этого и предназначен материал главы 6.

Основы Windows Installer XML

Структура базы данных Windows Installer достаточно сложна, поэтому непосредственная работа с ней неудобна и непродуктивна. В то же время Windows Installer XML позволяет создавать установочные пакеты без непосредственного заполнения таблиц, а его компилятор и компоновщик при работе проверяют выполнение ряда условий, в том числе наличие и целостность ссылок между элементами. Использование WiX основывается на редактировании XML файла (или нескольких файлов) определенной структуры. Схема данного файла достаточно сложна, поэтому, чтобы начать использовать Windows Installer XML, необходимо познакомиться с основными элементами и понятиями.

На момент написания книги полноценных инструментов с графическим интерфейсом, позволяющих воспользоваться функционалом WiX, еще не было. Хотя это делает использование Windows Installer XML достаточно сложным, результат способен оправдать усилия, потраченные на его освоение.

Общая структура файла сценария

Сценарий установки представляет собой XML-файл определенной структуры. После стандартного пролога (XML declaration) всегда следует корневой элемент Wix, где описываются используемые пространства имен. Далее мы будем использовать тег Product, хотя для других типов пакетов (отдельно распространяемые модули, патчи) допустимы и иные значения. Предлагаемая средой заготовка файла сценария приведена ниже.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
   <Product Id="????????-5BFA-4743-A28B-59CA465AC591" Name="SimpleSetup" Language="1033"
Version="1.0.0.0" Manufacturer="SimpleSetup" UpgradeCode="????????-60D1-4131-B5B1-
4FCCF8308DEA">
      <Package InstallerVersion="200" Compressed="yes" />
      <Media Id="1" Cabinet="media1.cab" EmbedCab="yes" />

      <Directory Id="TARGETDIR" Name="SourceDir">
         <Directory Id="ProgramFilesFolder">
            <Directory Id="INSTALLLOCATION" Name="SimpleSetup">
            </Directory>
```

```
        </Directory>
      </Directory>
      <Feature Id="ProductFeature" Title="SimpleSetup" Level="1">
      </Feature>
   </Product>
</Wix>
```

В текст также добавлены заготовки для описания структуры каталогов (Directory и дочерние узлы), наборов (Feature), а также Package – общие свойства пакета и Media – контейнер для содержимого. Эти и остальные элементы подробно описываются ниже.

GUID – зачем он нужен и как его получить

Все основные компоненты пакета контролируются установщиком Windows Installer на основе уникальных идентификаторов (GUID – Globally Unique Identifier). К ним относятся атрибуты Id и UpgradeCode элемента Product, атрибут Guid элемента Component. Наличие указанных атрибутов является обязательным и позволяет проверять, установлен или нет в системе данный компонент. При создании собственной программы установки следует обращать внимание на назначение уникальных идентификаторов для каждого требующего этого элемента.

Как же получить уникальное значение GUID? Для этого можно воспользоваться различными подходами: получить значение идентификатора программно из Visual Studio (в коде программы или из Immediate Windows); можно использовать, например, PowerShell или воспользоваться одной из многочисленных сторонних утилит.

Для получения значения GUID из Visual Studio откройте Immediate Window и наберите:

```
System.Guid.NewGuid() <Enter>
```

Вы получите результат вида: {????????-4547-468F-917F-75089DD068AC}. Строка внутри фигурных скобок и является искомым идентификатором. При работе с Windows Installer XML идентификатор может указываться как в фигурных скобках, так и без них.

Используя PowerShell, также можно получить значение уникального идентификатора. Для этого необходимо использовать несколько отличающийся синтаксис:

```
[System.Guid]::NewGuid().ToString()
```

получаемое значение будет аналогичным.

Замечание: если вы являетесь обладателем Visual Studio 2010, то для получения GUID можете воспользоваться пунктом меню Tools –> Create GUID. При этом будет запущена утилита guidgen.exe, входящая в состав Windows SDK.

Во всех примерах данной книги первые восемь символов идентификатора заменены вопросительными знаками. Это сделано для того, чтобы читатель не скопировал случайно значение из примера, а сгенерировал собственное, поскольку пересечение идентификаторов в коммерческих системах крайне нежелательно: продукт с дублирующимися значениями идентификаторов не будет корректно установлен.

Замечание: Windows Installer XML версии 3 и старше позволяет задавать GUID в смешанном (mixed-case) регистре. Но если в свойствах проекта на странице Build указать Warning level: Pedantic, то компилятор будет требовать приведения всех символов идентификатора к верхнему регистру.

Кроме того, Windows Installer XML позволяет указывать значение GUID как в фигурных скобках, так и без них.

Требования к системе и установка WiX

Требования к системе, предъявляемые непосредственно Windows Installer XML, крайне невелики, ведь в первую очередь это набор утилит командной строки. Но не стоит забывать о том, что мы хотим использовать WiX совместно со средой разработки Visual Studio, которая достаточно требовательна к аппаратным ресурсам. Установка Visual Studio подробно рассматривается в многочисленных источниках, а для Windows Installer XML дополнительно потребуется около 50Мб места на жестком диске. Скачать программу установки WiX (x86 и x64 версии) и исходные коды можно по адресу http://wix.sourceforge.net/releases/.

Может возникнуть ситуация, что пакет установки откажется производить установку надстройки для Visual Studio 2005 без предварительного наличия компонента ProjectAggregator2, выведя при этом соответствующее сообщение. Данный компонент является элементом VSIP SDK и используется для интеграции пакетов от сторонних производителей в Visual Studio. На этот случай разработчики Windows Installer XML позаботились о нас и положили пакет ProjectAggregator2.msi рядом с остальными материалами.

Создание простого решения

Начнем работу с создания решения, включающего проект на основе Windows Forms и простейшую программу установки. Здесь я подробно опишу все необходимые шаги.

Запустим Visual Studio 2010. Создадим новое решение, выбрав пункт меню File –> New –> Project… В появившемся диалоге New Project выберем вариант Other Project Types –> Visual Studio Solutions –> Blank Solution. Укажем имя решения, например SimpleSolution, и укажем место его размещения. Остальные свойства изменять не будем. Нажмем кнопку OK.

Теперь мы имеем пустое решение. Добавим в него проект на основе Windows Forms. Для этого нажмем правой кнопкой мыши на решении в Solution Explorer и выберем Add –> New Project…, или воспользуемся пунктом меню File –> Add –> New Project… В диалоге New Project укажем тип проекта Windows Forms Application, назовем его SimpleApplication.

Единственная форма, Form1, будет содержать одну кнопку, закрывающую приложение. Кроме того, укажем пиктограмму для приложения в его свойствах (нажмем правой кнопкой мыши в Solution Explorer на имени проекта SimpleApplication, выберем пункт Properties). Нажмем кнопку с многоточием справа от раскрывающегося списка Icon. В стандартном диалоге выбора файла укажем любой файл с расширением .ico. Пиктограмма пригодится нам для демонстрации процесса создания ярлыков. Закроем свойства проекта.

Добавим в решение проект программы установки. Для этого вызовем диалог New Project и укажем тип проекта Windows Installer XML -> Setup Project. Укажем имя для проекта, SimpleSetup, и нажмем кнопку OK. Откроем файл сценария установки - Product.wxs, если он не открылся автоматически. Сразу добавим в пакет ссылку на добавленный ранее проект Windows Forms. Нажмем правой кнопкой мыши на названии проекта SimpleSetup и в контекстном меню выберем Add Reference… Перейдем на закладку Projects, выберем в списке единственный элемент SimpleApplication и нажмем кнопки Add, а затем OK. Теперь мы сможем использовать в сценарии установки ссылочные переменные.

Основные свойства проекта и пакета

Установим значения наиболее важных атрибутов, большинство из которых является обязательным для любого установочного пакета.

Элемент Product содержит атрибуты, описывающие устанавливаемый продукт:

- Id – GUID, уникальный идентификатор продукта;
- UpgradeCode – GUID, уникальный идентификатор, используется при выпуске обновлений. Хотя значение данного атрибута можно не указать, настоятельно рекомендую не пренебрегать им – чтобы не было поздно, если вдруг потребуется обновить уже установленный продукт;
- Name – отображаемое название продукта;
- Version – версия продукта;
- Manufacturer – название компании-разработчика устанавливаемой программы;
- Language – 1033 (английский язык, по умолчанию). Для поддержки русского языка следует использовать значение 1049;
- Codepage – используемая кодовая страница, для русского языка необходимо указать 1251. При отсутствии значения данного атрибута пользователю может не отображаться текст сообщений.

Элемент Package предназначен для указания свойств msi-пакета:

- SummaryCodepage – номер кодовой страницы для сводной информации о пакете. Устанавливается аналогично Product.Codepage;
- Description – описание пакета установки;
- Manufacturer – название компании-разработчика пакета;
- Comments – комментарий, отображается в свойствах пакета.

Не все атрибуты являются обязательными, но их наличие позволит придать даже простой программе установки законченный вид. Элементы Product и Package с заданными значениями приведены ниже.

```
   <Product Id="????????-5BFA-4743-A28B-59CA465AC591" Name="Демонстрационное приложение 1"
Language="1049" Version="1.0.1" Manufacturer="Производитель продукта" UpgradeCode="????????-
60D1-4131-B5B1-4FCCF8308DEA" Codepage="1251">
```

```
    <Package InstallerVersion="200" Compressed="yes" SummaryCodepage="1251"
Description="Пакет установки 1" Manufacturer="Производитель программы установки"
Comments="Комментарии к программе установки" />
```

Так как указанный язык отличается от того, который используется по умолчанию, необходимо сообщить компилятору о нашем выборе, иначе в процессе сборки мы получим ошибку. Для этого откроем свойства проекта, щелкнув правой кнопкой мыши на имени проекта в Solution Explorer и выбрав пункт Properties. Перейдем на закладку Build и в поле Cultures to build укажем необходимость поддержки русского языка, набрав значение «ru-RU», при этом регистр букв значения не имеет.

Определение структуры каталогов

Опишем структуру каталогов для устанавливаемой программы. Пусть по умолчанию она устанавливается в папку Program Files, в подпапку Company Name\Demo Application. Каждой очередной папке в иерархии соответствует свой элемент. Кроме того, добавим ссылку на рабочий стол для последующего помещения туда ярлыка.

Каталоги описываются элементом Directory, атрибут Id описывает идентификатор, а атрибут Name – название каталога. Ряд системных идентификаторов описывает пути к конкретным папкам: например, ProgramFilesFolder – путь к каталогу Program Files на целевой машине, а DesktopFolder – к рабочему столу. Подробно об указанных свойствах написано в главе 3, в разделе «Стандартные пути и их аналоги в управляемом коде». При использовании одного из стандартных идентификаторов атрибут Name не используется и его можно не указывать.

На верхнем уровне всегда должен находиться корневой элемент – виртуальный каталог с идентификатором TARGETDIR и именем SourceDir. Его отсутствие приведет к ошибке компиляции.

Так выглядит завершенное описание структуры каталогов для нашего примера:

```
    <!-- Виртуальный корневой каталог -->
    <Directory Id="TARGETDIR" Name="SourceDir">
      <!-- Каталог \Program Files\Company Name\Simple Application -->
      <Directory Id="ProgramFilesFolder">
        <Directory Id="CompanyNameFolder" Name="Company Name">
          <Directory Id="INSTALLLOCATION" Name="Simple Application">
          </Directory>
        </Directory>
      </Directory>
      <!-- Рабочий стол -->
      <Directory Id="DesktopFolder" />
    </Directory>
```

Теперь мы сможем размещать элементы в описанных каталогах.

Компоненты – контейнеры для файлов

Программа установки в нашем решении будет содержать единственный файл: исполняемый файл программы. Кроме того, мы создадим для него ярлык на рабочем столе. Для каждого файла используем отдельный элемент Component.

Для ярлыка нам потребуется пиктограмма, ее мы извлечем из исполняемого файла. Описание пиктограммы выполняется с использованием элемента Icon, для которого задается идентификатор и исходный файл – атрибуты Id и SourceFile соответственно. Замечу, что идентификатор пиктограммы обязательно должен иметь то же расширение, что и файл, из которого извлекается изображение, поэтому в нашем случае он будет иметь расширение .exe.

Далее поместим компоненты. В общем случае они могут располагаться непосредственно внутри тега Directory. Но я предпочитаю описывать компоненты отдельно, ссылаясь на каталоги с помощью элемента DirectoryRef, в атрибут Id которого помещается идентификатор каталога. На мой взгляд, это позволяет лучше структурировать содержимое файла сценария.

Внутрь DirectoryRef поместим элемент Component. Обязательными для него являются атрибуты Guid и Id. Первый, как было описано выше, позволяет контролировать установку, а второй – ссылаться на данный компонент по имени.

Вложенным элементом для Component в нашем случае будет элемент File. Для элемента File обязательными являются атрибуты Id – уникальный идентификатор, Name – имя файла без пути, Source – абсолютный путь к файлу. Атрибуты DiskId и KeyPath будут рассматриваться в главе 3, сейчас оставим их значения без изменений.

Последний элемент – Shortcut – вложен в нашем случае в элемент File (хотя может находиться и отдельно) и описывает ярлык для программы на рабочем столе. У него множество атрибутов, но наиболее важными являются Id, Name, Directory и Icon. Являясь достаточно сложным, подробно этот элемент описан в главе 3.

```
    <!-- Пиктограмма, извлекаемая из исполняемого (.exe) файла -->
    <Icon Id="ProgramIcon.exe" SourceFile="$(var.SimpleApplication.TargetPath)" />

    <!-- Содержимое для размещении в каталоге с идентификатором INSTALLLOCATION -->
    <DirectoryRef Id="INSTALLLOCATION">
      <Component Id="SimpleApplicationExeComponent" Guid="????????-532D-4535-A3F4-
503384846E64">
        <File Id="SimpleApplicationExe" Name="$(var.SimpleApplication.TargetFileName)"
Source="$(var.SimpleApplication.TargetPath)" DiskId="1" KeyPath="yes" >
          <!-- Ярлык для запуска программы -->
          <Shortcut Id="DesktopShortcut" Name="Simple Application" Description="Ярлык на
рабочем столе" Directory="DesktopFolder" Advertise="yes" Icon="ProgramIcon.exe"></Shortcut>
        </File>
      </Component>
    </DirectoryRef>
```

Таким образом, не рассматривая детали отдельных элементов, мы получили фрагмент сценария установки, описывающий простое содержимое – исполняемый файл и ярлык с пиктограммой.

Наборы компонентов - Features

Недостаточно описать компоненты – требуется разместить их по одному или более именованным наборам. Набор, являясь контейнером для компонентов – наименьшая единица, доступная для выбора в процессе установки.

Замечание: строго говоря, Feature в данном контексте переводится по-разному: чаще используются варианты «опция», «набор», а иногда термин не переводится совсем. Я предпочитаю использовать термин «наборы компонентов» или просто «наборы».

Наша программа установки очень проста – в ней будет единственный набор, состоящий из одного компонента.

```
    <!-- Наборы компонентов -->
    <Feature Id="Complete" Title="Simple Application" Description="Полная установка" Level="1"
ConfigurableDirectory="INSTALLLOCATION" >
      <ComponentRef Id="SimpleApplicationExeComponent" />
    </Feature>
```

Тег Feature позволяет организовать наборы в древовидную структуру, так как эти элементы допускается вкладывать один в другой. Значения атрибутов Title и Description используются для отображения в элементах интерфейса, а ConfigurableDirectory позволяет в процессе установки изменять каталог для копирования файлов.

Для добавления в набор компонентов используется элемент ComponentRef, в атрибут Id которого должен помещаться идентификатор компонента.

Добавление стандартного интерфейса пользователя

Если сейчас собрать наш проект, то, исправив ошибки, мы получим msi-сборку, которая позволит установить проект в каталог %ProgramFiles%\Company Name\Simple Application, одновременно создав ярлык на рабочем столе. В процессе установки мы увидим только немодальное окно с индикатором состояния, приведенное на рисунке 1.2 – это встроенный диалог базового интерфейса.

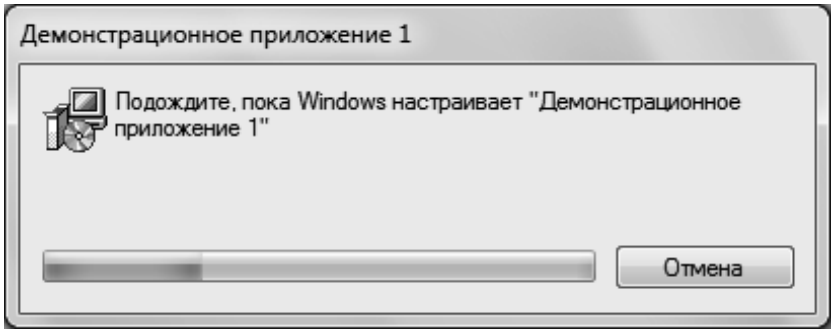

Рисунок 1.2 Встроенный диалог, отображающий ход установки.

Такой интерфейс практически равнозначен его отсутствию. Конечно, он применим в некоторых случаях, но, как правило, требуется большее участие пользователя в процессе установки. Приятно сознавать, что Windows Installer XML уже содержит несколько готовых вариантов интерфейса, каждый из которых содержит ряд связанных диалогов. Вставить один из этих вариантов в сценарий установки можно, добавив всего несколько строк текста.

```
    <!-- Интерфейс пользователя -->
    <UI Id="MyWixUI_Mondo">
      <UIRef Id="WixUI_Mondo" />
      <UIRef Id="WixUI_ErrorProgressText" />
    </UI>
```

Здесь мы добавили ссылку (элемент UIRef) на набор диалогов с названием WixUI_Mondo. Ссылка на фрагмент WixUI_ErrorProgressText дополнительно подключает локализованные ресурсы, отображающие информацию о возникающих в процессе установки ошибках.

Кроме того, библиотека, отвечающая за интерфейс пользователя, является расширением. Подробно об использовании расширений написано в главе 4, а о настройке и расширении интерфейса – в главе 6. Здесь мы только подключим это расширение.

Нажмем правой кнопкой мыши на проекте WiX в Solution Explorer, выберем пункт Add Reference… В открывшемся диалоге выберем закладку Browse, в списке укажем файл WixUIExtension.dll и нажмем кнопки Add, а затем OK. Это заставит компоновщик добавить к проекту библиотеку для отображения пользовательского интерфейса.

Результат

Выполнив описанные в этой главе действия, мы получим простой, но работоспособный дистрибутив для нашей программы. В процессе установки создаются каталоги в %ProgramFiles%, куда копируется исполняемый файл, а на рабочем столе создается ярлык для его запуска. Функционал, предоставляемый этим пакетом, крайне ограничен, но он позволяет продемонстрировать базовые принципы работы с Windows Installer XML. Все последующие примеры будут во многом использовать описанные здесь элементы.

Ниже приведен полный текст файла сценария установки.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
  <Product Id="????????-5BFA-4743-A28B-59CA465AC591" Name="Демонстрационное приложение 1"
Language="1049" Version="1.0.1" Manufacturer="Производитель продукта" UpgradeCode="????????-
60D1-4131-B5B1-4FCCF8308DEA" Codepage="1251">
    <Package InstallerVersion="200" Compressed="yes" SummaryCodepage="1251" Description="Пакет
установки 1" Manufacturer="Производитель программы установки" Comments="Комментарии к
программе установки" />
    <Media Id="1" Cabinet="media1.cab" EmbedCab="yes" />

    <!-- Виртуальный корневой каталог -->
    <Directory Id="TARGETDIR" Name="SourceDir">
```

```
      <!-- Каталог \Program Files\Company Name\Simple Application -->
      <Directory Id="ProgramFilesFolder">
        <Directory Id="CompanyNameFolder" Name="Company Name">
          <Directory Id="INSTALLLOCATION" Name="Simple Application">
          </Directory>
        </Directory>
      </Directory>
      <!-- Рабочий стол -->
      <Directory Id="DesktopFolder" />
    </Directory>

    <!-- Пиктограмма, извлекаемая из исполняемого (.exe) файла -->
    <Icon Id="ProgramIcon.exe" SourceFile="$(var.SimpleApplication.TargetPath)" />

    <!-- Содержимое для размещения в каталоге с идентификатором INSTALLLOCATION -->
    <DirectoryRef Id="INSTALLLOCATION">
      <Component Id="SimpleApplicationExeComponent" Guid="????????-532D-4535-A3F4-
503384846E64">
        <File Id="SimpleApplicationExe" Name="$(var.SimpleApplication.TargetFileName)"
Source="$(var.SimpleApplication.TargetPath)" DiskId="1" KeyPath="yes" >
          <!-- Ярлык для запуска программы -->
          <Shortcut Id="DesktopShortcut" Name="Simple Application" Description="Ярлык на
рабочем столе" Directory="DesktopFolder" Advertise="yes" Icon="ProgramIcon.exe"></Shortcut>
        </File>
      </Component>
    </DirectoryRef>

    <!-- Именованные наборы компонентов -->
    <Feature Id="Complete" Title="Simple Application" Description="Полная установка" Level="1"
ConfigurableDirectory="INSTALLLOCATION" >
      <ComponentRef Id="SimpleApplicationExeComponent" />
    </Feature>

    <!-- Интерфейс пользователя -->
    <UI Id="MyWixUI_Mondo">
      <UIRef Id="WixUI_Mondo" />
      <UIRef Id="WixUI_ErrorProgressText" />
    </UI>
  </Product>
</Wix>
```

Глава 2. Интеграция в Visual Studio

Я предпочитаю создавать решения с использованием привычных инструментов, важнейшим из которых для меня является Visual Studio; разработка программ установки не является исключением. Хотя применение среды разработки не является обязательным, оно может существенно упростить процесс разработки, делая его более продуктивным. Добавление функционала в знакомый инструмент позволяет ускорить обучение и сэкономить время, необходимое для достижения результата. В этой главе рассматриваются преимущества использования Windows Installer XML из Visual Studio.

Какие же плюсы несет в себе интеграция? Во-первых, после установки WiX в Visual Studio появляются новые типы проектов, которые можно добавлять в решения. Одновременно конфигурируются утилиты, используемые для сборки данных типов проектов, а возможность подключения ссылок избавляет нас от необходимости явно указывать пути к содержимому. Кроме того, не прилагая дополнительных усилий, мы получаем все плюсы от использования систем контроля версий (в случае наличия настроенных Visual Source Safe или Team Foundation Version Control). Также очень привлекательно выглядят возможности IDE Visual Studio по работе с XML.

Теперь давайте подробнее рассмотрим эти и некоторые другие вопросы.

Шаблоны основных типов проектов

Windows Installer XML добавляет в Visual Studio одноименную группу шаблонов, в которой содержатся три основных типа проектов, а также по одному типу проекта Custom Action Project для каждого поддерживаемого языка (в версии 3.5 поддерживаются C#, C++ и Visual Basic).

- Setup Project – наиболее часто используемый шаблон для создания msi-сборок;
- Merge Module Project – шаблон для создания MSM-пакетов – сборок для установки отдельных компонентов. Эти компоненты не могут быть использованы отдельно, а добавляются в msi-пакеты. Так могут распространяться общие библиотеки;
- Setup Library Project – заготовка для проекта wixlib – расширения с возможностью последующего использования из WiX.
- C#/C++/VB Custom Action Project – если для какой-либо задачи нет готового доступного решения, всегда можно написать собственную библиотеку и добавить ее в проект.

При выборе того или иного шаблона создается новый проект, в котором присутствуют заготовки необходимых файлов, добавлены наиболее часто используемые зависимости, а также установлены общие свойства проекта.

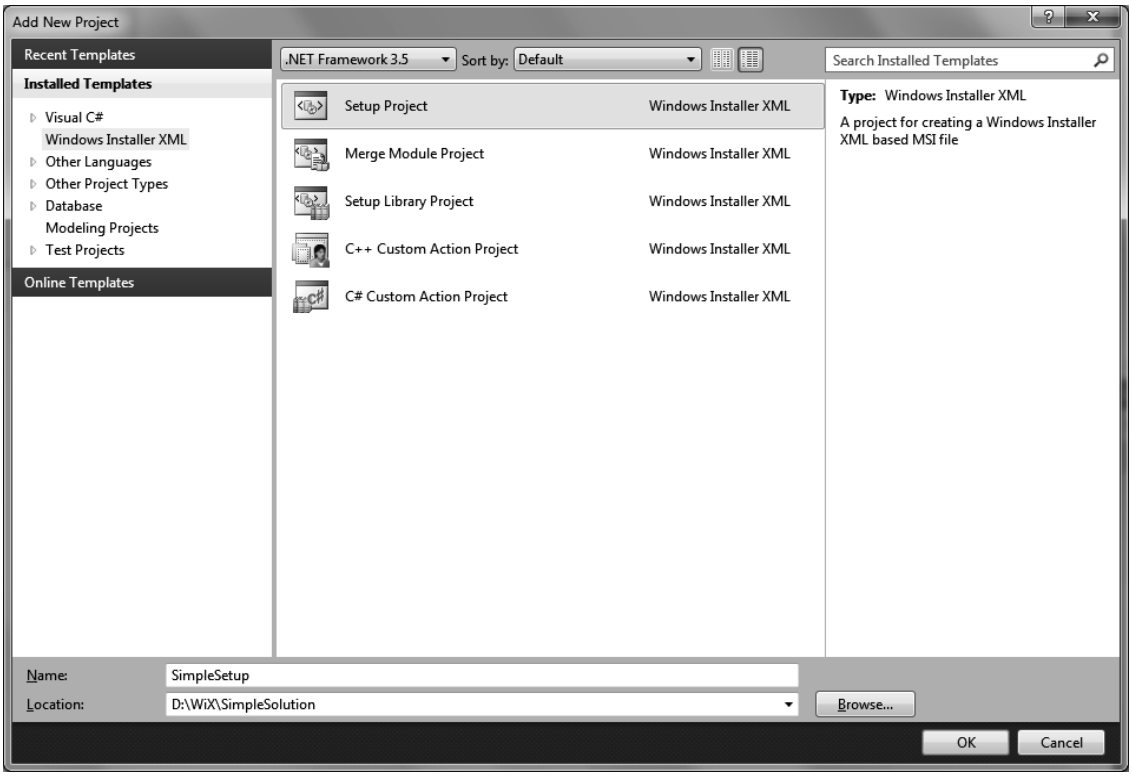

Рисунок 2.1 Типы проектов Windows Installer XML.

Мне пришлось столкнуться с тем, что в Visual Studio иногда отсутствуют шаблоны проектов Custom Action Project. Это происходило на тех машинах, где одновременно были установлены несколько различных версий Visual Studio. Данные шаблоны успешно устанавливались и отображались для более ранней версии, например, для Visual Studio 2008, но вместе с тем они отсутствовали в Visual Studio 2010. В аналогичной ситуации надо сделать следующее:

1. Найти в каталоге с той версией Visual Studio, где шаблоны установлены успешно, файлы с названиями CustomActionCS.zip, CustomActionCPP.zip, CustomActionVB.zip для языков C#, C++ и VB соответственно.
2. Из той версии Visual Studio, где отсутствуют необходимые шаблоны, надо выбрать пункт меню Tools -> Options… В диалоговом окне Options в списке слева указать Projects and Solutions, после чего справа найти путь с названием User project templates location (путь к устанавливаемым пользователем шаблонам).
3. Скопировать найденные (недостающие) файлы в папку с пользовательскими шаблонами.

Добавление ссылок на проекты и библиотеки.

В проект типа Setup Project мы можем добавлять ссылки на другие проекты решения, что позволяет быстро подключать сборки, внешние файлы и библиотеки расширения.

Добавление ссылки на проект производится привычным для Visual Studio способом: можно воспользоваться меню Project -> Add Reference… или нажать правой кнопкой мыши на проекте WiX из Solution Explorer и выбрать пункт Add Reference… Появившееся диалоговое окно (рисунок 2.2) несколько отличается от стандартного. В нем три закладки; первая, Projects, позволяет

добавить ссылку на один из проектов в решении. Вторая, Browse, позволяет выбрать элемент из файловой системы, в том числе подключить библиотеку расширений. Третья закладка, Recent, хранит перечень недавно использовавшихся элементов.

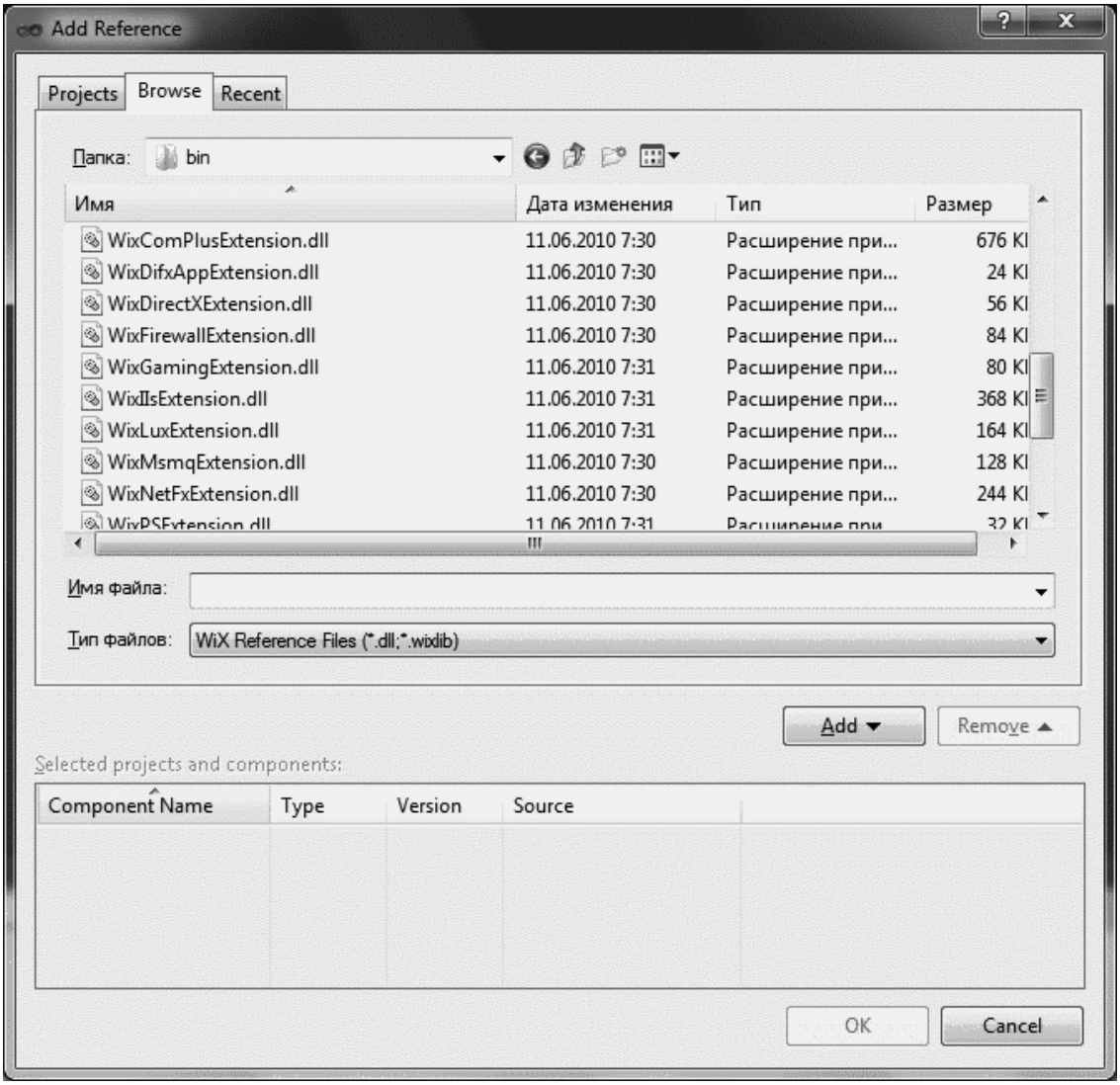

Рисунок 2.2 Диалог добавления ссылки на проект.

Ссылочные переменные

Добавив в проект программы установки ссылку на проект, мы получаем возможность использования ряда ссылочных переменных (project reference variables). Это дает возможность отказаться от необходимости указания путей к файлам и при компиляции всегда использовать последние версии сборок.

Например, так можно определить пиктограмму, извлекаемую из основного исполняемого файла в проекте DemoApplication, где переменная TargetPath возвращает нам полный путь к файлу:

```
    <Icon Id="ProgramIcon.exe" SourceFile="$(var.DemoApplication.TargetPath)" />
```

Таблица 2.1. Ссылочные переменные.

Название переменной	Пример использования	Пример возвращаемого значения
var.ProjectName.Configuration	$(var.MyProject.Configuration)	Debug или Release
var.ProjectName. FullConfiguration	$(var.MyProject.FullConfiguration)	Debug | AnyCPU
var.ProjectName.Platform	$(var.MyProject.Platform)	AnyCPU, Win32, x64 или ia64
var.ProjectName.ProjectDir	$(var.MyProject.ProjectDir)	C:\users\<пользователь>\Documents\Visual Studio 2010\Projects\MyProject\
var.ProjectName.ProjectExt	$(var.MyProject.ProjectExt)	.csproj
var.ProjectName. ProjectFileName	$(var.MyProject.ProjectFileName)	MyProject.csproj
var.ProjectName.ProjectName	$(var.MyProject.ProjectName)	MyProject
var.ProjectName.ProjectPath	$(var.MyProject.ProjectPath)	C:\users\<пользователь>\Documents\Visual Studio 2010\Projects\MyProject\MyApp.csproj
var.ProjectName.TargetDir	$(var.MyProject.TargetDir)	C:\users\<пользователь>\Documents\Visual Studio 2010\Projects\MyProject\obj\Debug\
var.ProjectName.TargetExt	$(var.MyProject.TargetExt)	.exe
var.ProjectName. TargetFileName	$(var.MyProject.TargetFileName)	MyProject.exe
var.ProjectName.TargetName	$(var.MyProject.TargetName)	MyProject
var.ProjectName.TargetPath	$(var.MyProject.TargetPath)	C:\users\<пользователь>\Documents\Visual Studio 2010\Projects\MyProject\obj\Debug\MyProject.exe
var.ProjectName.Culture .TargetPath	$(var.MyProject.en-US.TargetPath)	C:\users\<пользователь>\Documents\Visual Studio 2010\Projects\MyProject\obj\Debug\en-US\MyProject.msm
var.SolutionDir	$(var.SolutionDir)	C:\users\<пользователь>\Documents\Visual Studio 2010\Projects\MySolution\
var.SolutionExt	$(var.SolutionExt)	.sln
var.SolutionFileName	$(var.SolutionFileName)	MySolution.sln
var.SolutionName	$(var.SolutionName)	MySolution
var.SolutionPath	$(var.SolutionPath)	C:\users\<пользователь>\Documents\Visual Studio 2010\Projects\MySolution\MySolution.sln

Подключение библиотек расширения

Воспользовавшись закладкой Browse или Recent, мы также можем подключить к проекту внешнюю библиотеку. В первую очередь, это упрощает использование библиотек расширения. Так, например, для работы с интерфейсом пользователя нам потребуется библиотека WixUIExtension.dll. Конечно, ее можно подключить, указав дополнительные свойства для компоновщика, но добавление ссылки на библиотеку, пожалуй, несколько проще.

Свойства проекта

Свойства проекта сосредоточены по пяти закладкам. В верхней части каждой закладки находятся два раскрывающихся списка: Configuration и Platform. Доступны они на закладках Build (сборка) и Tool Settings (настройки инструментов) и позволяют, при необходимости, отдельно указывать настройки для различных типов сборки (Активная, Debug, Release, Все) и поддерживаемых платформ.

Закладка «Installer»

На данной странице устанавливаются два параметра:

Output name – имя результирующего пакета без расширения – аналог поля Assembly name для, например, приложения Windows Forms.

Output type – раскрывающийся список, позволяющий указать тип результирующего пакета. Доступно три варианта выбора: Windows Installer Package (.msi), Merge Module (.msm) и WiX Library (.wixlib), каждый из которых соответствует своему типу проекта, как описано выше.

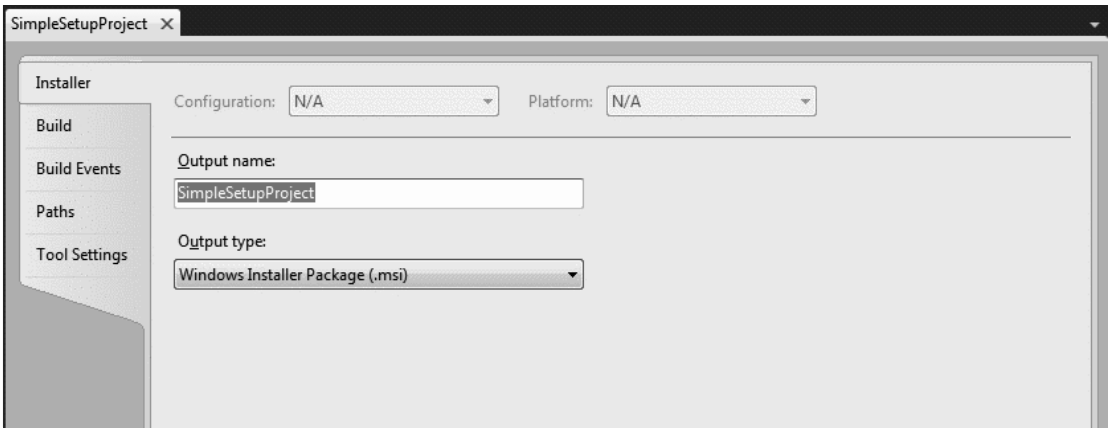

Рисунок 2.3 Свойства проекта, закладка «Installer».

Закладка «Build»

Элементы на данной странице размещены по трем группам: General, Messages, Output. Рассмотрим назначение элементов, двигаясь сверху вниз.

Флажок Define ‘Debug’ preprocessor variable (объявить переменную препроцессора Debug) установлен по умолчанию для отладочной сборки, а поле Define preprocessor variables (объявить переменные препроцессора) позволяет объявить ваши собственные именованные переменные. Значения переменных не должны содержать двойных кавычек. Подробнее переменные препроцессора рассматриваются в главе 3, в разделе «Использование свойств и переменных».

Поле Define variables (объявить переменные) позволяет объявить переменные аналогично переменным препроцессора. Данный вопрос также рассматривается в главе 3, в разделе «Использование свойств и переменных».

Поле Cultures to build (культуры, используемые в процессе сборки) используется для указания перечня необходимых локализованных ресурсов. Чуть ниже данного поля есть примечание – Leave blank to build all cultures (оставьте пустым для сборки всех локализованных ресурсов). Однако для успешного подключения русского языка необходимо явно указывать в данном поле значение «ru-RU» (регистр неважен), иначе мы получим ошибку компоновки. Если указать в качестве значения несколько культур (например, ru-RU;en-US), то для каждой из них будет создана своя сборка в подкаталоге вывода (для нашего случая, Debug\ru-RU, Debug\en-US). Кроме того, существуют случаи, когда необходимо использовать для единственного вывода несколько культур. Например, для некоторых элементов библиотеки WixUtilExtension отсутствуют

русскоязычные ресурсы, поэтому необходимо использовать одновременно ресурсы для русского и английского языков. В этом случае необходимо воспользоваться дополнительными параметрами компоновщика, расположенными на закладке Tool Settings.

Группа Messages начинается с раскрывающегося списка Warning Level, по умолчанию установленного в Normal. Если указать уровень Pedantic, то компилятор откажется принимать, например, смешанный стиль указания GUID, потребовав его перевода в верхний регистр.

Следующее поле – Suppress specific warnings – позволяет игнорировать предупреждения с указанными номерами.

Флажок Treat warnings as errors (рассматривать предупреждения как ошибки) – трактовать любые предупреждения как ошибки.

Флажок Verbose output (подробный вывод) – включает наиболее информативное комментирование процесса сборки.

Группа свойств Output начинается с поля Output path (путь для помещения результатов). Вы можете изменить значение по умолчанию, если это вам необходимо.

Флажок Do not delete temporary files позволяет отключить удаление временных файлов.

Флажок Suppress output of the wixpdb files подавляет дополнительный отладочный вывод.

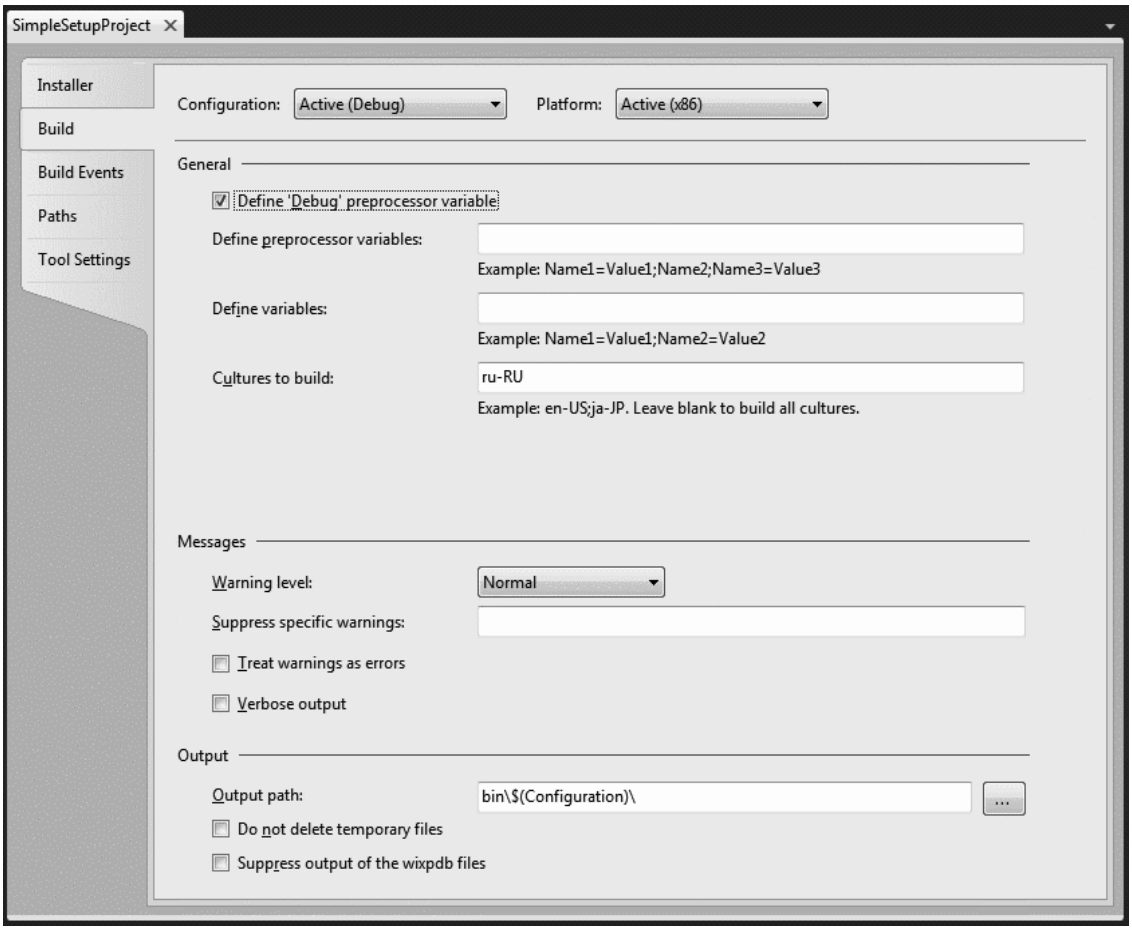

Рисунок 2.4 Свойства проекта, закладка «Build».

Build Events

Данная страница является для Visual Studio стандартной и присутствует для всех типов проектов. Здесь можно указать команды, которые будут выполнены как до начала сборки, так и после ее окончания. Команды в поле Pre-build Event Command Line (команды для выполнения до начала сборки) выполняются всегда, если указанное поле не пусто. Выполнение команд в поле Post-build Event Command Line (команды для выполнения после окончания сборки) зависит от значения раскрывающегося списка Run the post-build event (запуск события об окончании сборки), которое может принимать следующие значения:

- On successful build – после успешной сборки (по умолчанию);
- Always – всегда;
- When the build updates the project output – когда результат сборки обновляет предыдущий (отличается от него) – в проект были внесены изменения.

Выполняемые команды могут содержать макросы, значения которых вычисляются для конкретного проекта. Чтобы не запоминать их, можно воспользоваться кнопками Edit Pre-build и Edit Post-build, отображающие одно и то же диалоговое окно, где можно выбрать один или несколько из доступных макросов. Перечень макросов содержит все стандартные, а также два специфических для WiX: TargetPdbName и TargetPdbPath. Первый возвращает только имя отладочной сборки (без пути), а второй – полный путь (с именем) к ней же.

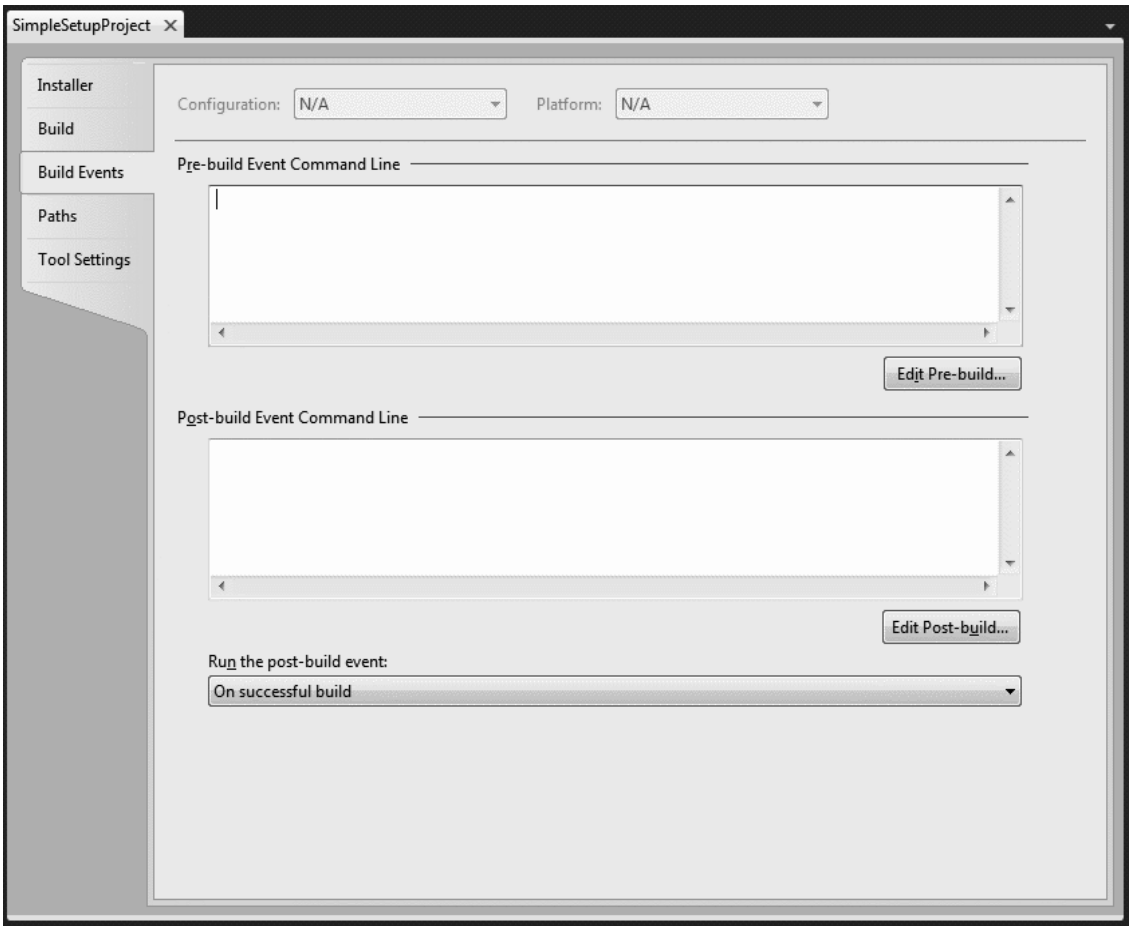

Рисунок 2.5 Свойства проекта, закладка «Build Events».

Paths

Страница Paths позволяет указать пути, используемые в процессе сборки проектов. Здесь присутствуют две аналогичные группы элементов: Reference Paths и Include Paths. Первый используется для поиска библиотек расширений и файлов .wixlib. Пути, указанные в группе Include Paths, используются для файлов, подключаемых с помощью директивы include. Процесс выбора каталогов стандартный – необходимо нажать на кнопку с многоточием напротив поля Folder и указать папку или впечатать ее название в само поле. После нажатия кнопки Add Folder путь к ней будет добавлен в список ниже. Чтобы удалить строку из списка, следует воспользоваться кнопкой с изображением крестика. Чтобы заменить существующую строку – выделить ее и нажать кнопку Update.

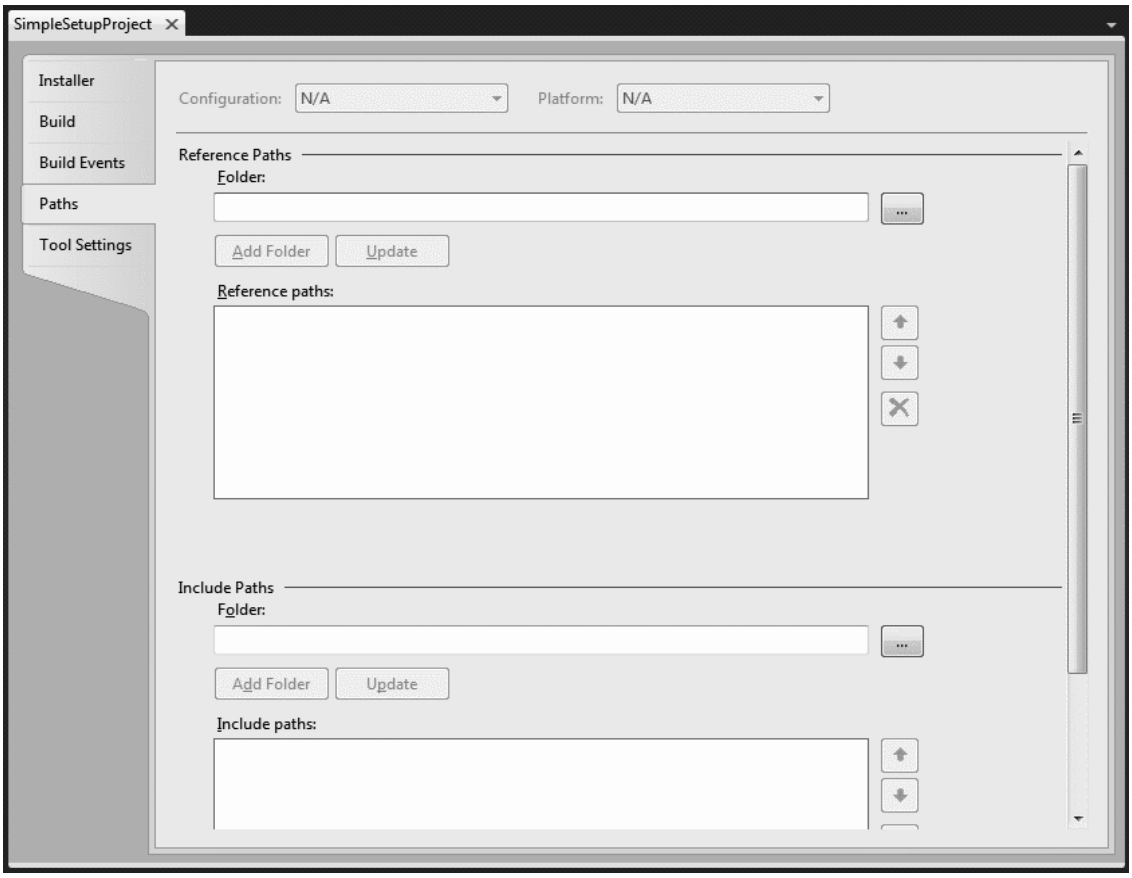

Рисунок 2.6 Свойства проекта, закладка «Paths».

Tool Settings

Закладка Tool Settings может быть использована для настройки дополнительных параметров инструментов.

Здесь указываются дополнительные параметры для компилятора (утилита candle.exe) и компоновщика (утилита light.exe). Это может потребоваться как при подключении расширений, так и при локализации для подключения языковых пакетов. Отдельные параметры отделяются друг от друга пробелами. Данный подход является стандартным при непосредственной работе с утилитами WiX из командной строки, но в случае работы из Visual Studio удобнее воспользоваться привычным добавлением ссылок на необходимые библиотеки.

Например, чтобы подключить языковой пакет для русского языка, следует установить для компоновщика (поле Linker) дополнительный параметр:

-cultures:ru-RU

Замечание: удобнее подключать языковые пакеты на странице Build, указывая необходимую культуру в поле Cultures to build. Единственное исключение – случай, когда для сборки необходимо одновременное использование нескольких культур. При этом, как правило, необходимо кроме языка локализации – например, русского – указать одновременно и английский:

-cultures:ru-RU;en-US

Чтобы отображать интерфейс пользователя, потребуется подключить соответствующее расширение:

-ext WixUIExtension

Замечание: подключать расширения проще, добавляя ссылки на необходимые библиотеки через меню Add Reference…, приведенное на рисунке 2.2.

На этой же странице можно отключать проверки отдельных или всех валидаторов совместимости (ICE – Internal Consistency Evaluator), которые используются службами Windows Installer для контроля msi-пакетов на наличие возможных проблем.

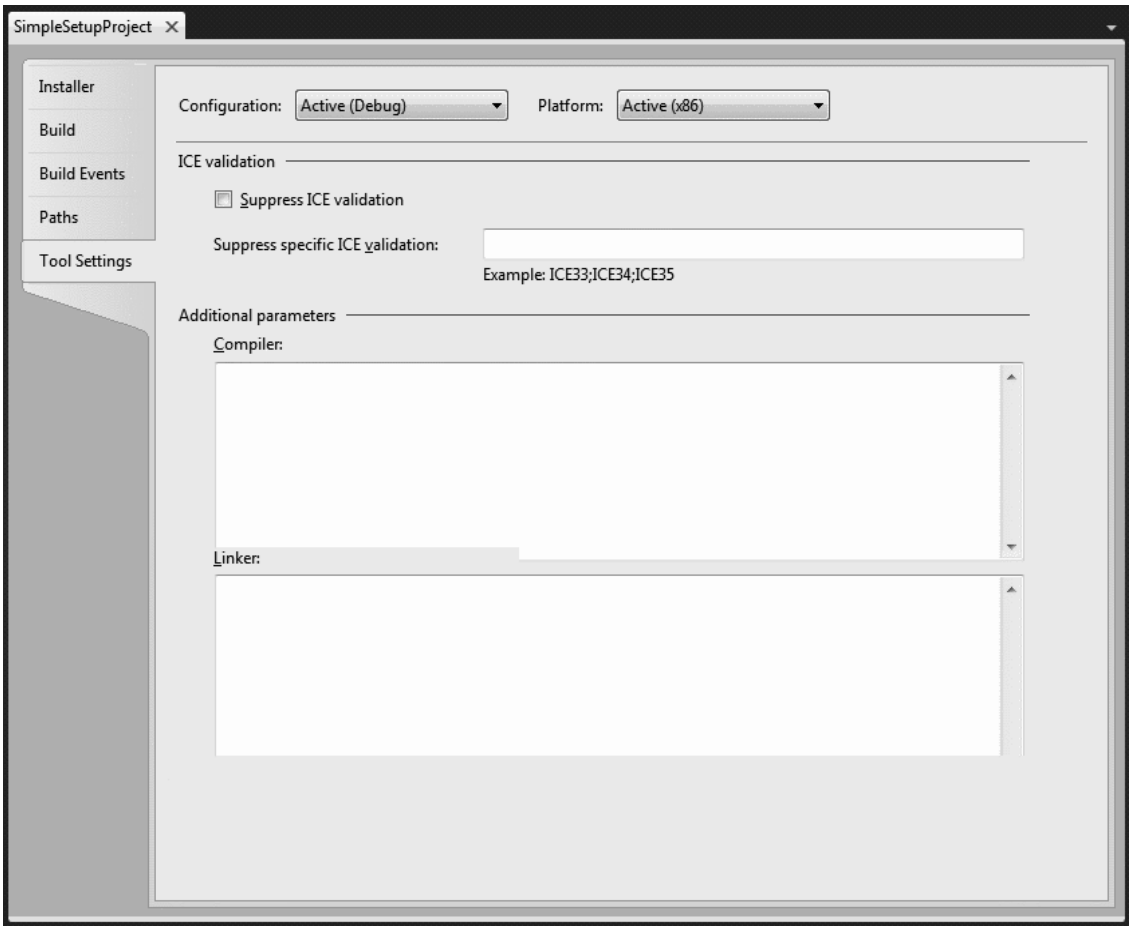

Рисунок 2.7 Свойства проекта, закладка «Tool Settings».

Возможности редактора по работе с XML

Редактор среды Visual Studio обладает рядом возможностей, которые делают работу с XML значительно удобнее:

- анализ XSD-схемы при импорте пространства имен и использование IntelliSence в процессе ввода. Данная возможность доступна при редактировании любых XML-файлов, но от этого не становится менее ценной;
- контроль наличия парных тегов и выделение проблемных мест;
- подсветка конструкций – раскрашивание элементов, атрибутов и значений различными цветами;
- возможность свертывания узлов, упрощающая работу с большими файлами.

Сборка проектов WiX в Team Foundation Server

Одной из возможностей, которые предоставляет Team Foundation Server, является обеспечение непрерывной интеграции решений при их создании. Если встроить проект программы установки в этот процесс, команда тестирования всегда сможет получить последнюю версию пакета, а разработчики – отклики уже на ранних стадиях построения решения.

Существует два способа заставить собираться проекты WiX на сервере TFS: простой и не очень. Первый способ прост технически, но требует наличия соответствующих прав доступа. Администратор устанавливает Windows Installer XML на сервер TFS. И все работает.

Но, допустим, мы столкнулись с тем случаем, когда установка на сервере дополнительных пакетов невозможна. Это может быть запрещено действующими в организации политиками безопасности. Чтобы обойти это ограничение, добавим все необходимые для сборки инструменты непосредственно в проект и продолжим работу. Как это сделать?

Пусть у нас уже создан командный проект. Для этого проекта есть соответствующий узел в системе контроля версий, ассоциированный с локальным каталогом. И создано хотя бы одно определение сборки. Как видно из рисунка 2.8, мы имеем решение, связанный командный проект и одно определение сборки RequestBuild, которое активирует сборку решения по требованию.

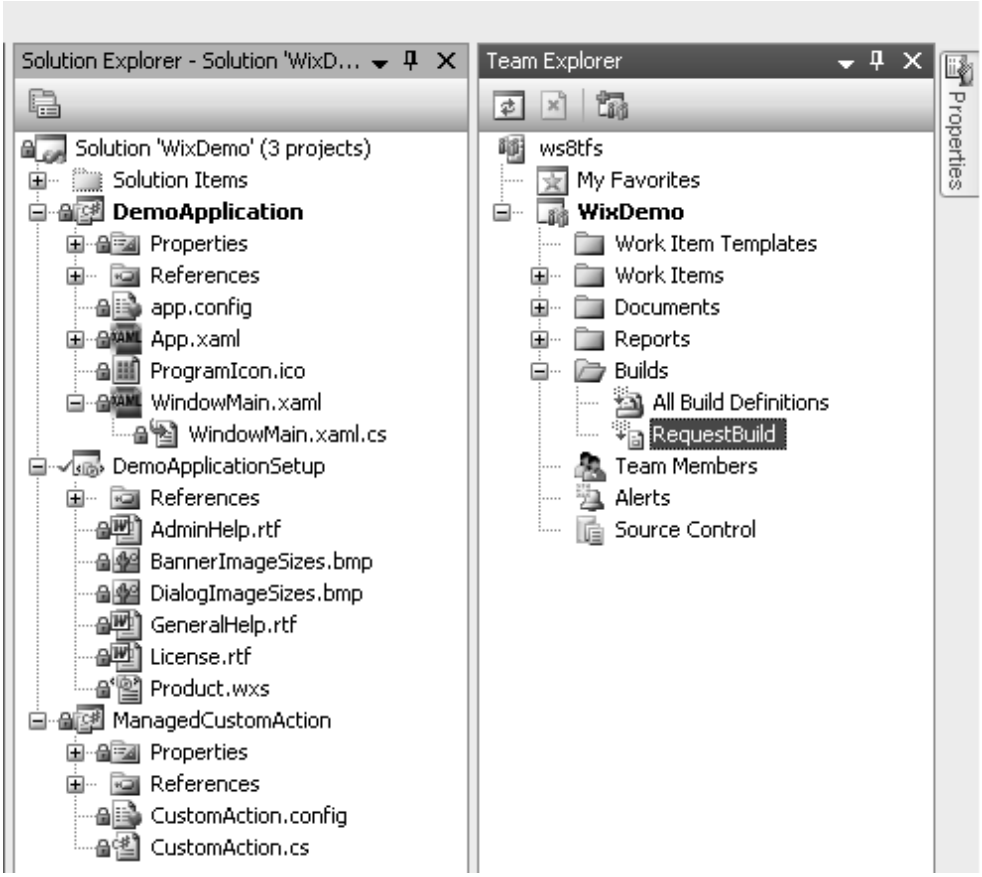

Рисунок 2.8 Командный проект и структура связанного решения.

Сборка решения на локальной машине проходит успешно, ведь все зависимые элементы доступны. А вот активация RequestBuild приводит к ошибкам при сборке, ведь на сервере TFS отсутствуют необходимые библиотеки Windows Installer XML. Мы видим ошибку, описание которой аналогично приведенной ниже:

---

error MSB4019: The imported project "C:\Program Files\MSBuild\Microsoft\WiX\v3.5\Wix. targets" was not found. Confirm that the path in the <Import> declaration is correct, and that the file exists on disk.

Прежде всего, на одном уровне с каталогами проектов добавим каталог для двоичных файлов. Назовем его WixBinaries. Если решение находится в каталоге D:\WixDemo, а проект WiX – в папке D:\WixDemo\DemoApplicationSetup, то каталог с двоичными файлами будет называться D:\WixDemo\WixBinaries. Скопируем в указанный каталог содержимое двух папок: C:\Program Files\Windows Installer XML v3.5\bin\ и C:\Program Files\MSBuild\Microsoft\WiX\v3.5\. Пути на вашей машине могут несколько отличаться; данный пример приведен для WiX версии 3.5, который установлен в предлагаемый по умолчанию каталог.

Добавим в решение файлы. Щелкнем правой кнопкой мыши в Solution Explorer на названии решения, выберем пункты меню Add -> Existing Item. В открывшемся диалоговом окне откроем каталог WixBinaries, выберем все файлы и нажмем Add. Файлы будут добавлены в решение и помечены для добавления в систему контроля версий.

Извлечем на редактирование проекты Windows Installer XML, воспользовавшись командой Check Out For Edit… Теперь выгрузим решение из Visual Studio и откроем в любом текстовом редакторе файл проекта «Имя проекта сценария».wixproj. Добавим в содержимое файла прямо перед элементом Import следующие строки:

```
<PropertyGroup>
  <WixToolPath>..\WixBinaries</WixToolPath>
  <WixTargetsPath>$(WixToolPath)\Wix.targets</WixTargetsPath>
  <WixTasksPath>$(WixToolPath)\wixtasks.dll</WixTasksPath>
</PropertyGroup>
```

Отмечу, что проект на сервере собирается только с отключенной в свойствах проекта проверкой ICE.

Откроем решение в Visual Studio. При открытии среда предупредит нас о том, что открытие измененного содержимого может быть опасным (рисунок 2.9). Разрешим открытие проекта, установив переключатель в положение Load project normally. Необходимо добиться отсутствия ошибок при открытии решения и последующей локальной сборке.

Теперь выполним команду Check-In, подтвердив все внесенные изменения. При этом в систему контроля версий также будут скопированы файлы из каталога WixBinaries. Активируем сборку RequestBuild, в результате которой мы должны получить успешно собранное решение.

Рисунок 2.9 Предупреждение при загрузке потенциально опасного содержимого.

Глава 3. Базовая функциональность

Эта глава посвящена рассмотрению основных элементов, без которых невозможна полноценная работа с Windows Installer XML. Рассматриваемые здесь вопросы используются при создании любого проекта установки.

Решение для демонстрации возможностей

Для этой и последующих глав мы будем использовать заранее подготовленное приложение на основе WPF, а также несколько вспомогательных файлов. Таким образом, мы имеем аналог полноценного программного продукта, установка которого позволит рассмотреть наиболее востребованные сценарии развертывания.

Результатом сборки проекта DemoApplication будут два основных файла: исполняемый DemoApplication.exe и конфигурационный DemoApplication.exe.config, а также один вспомогательный – файл отладочных символов DemoApplication.pdb, который мы тоже включим в установочный пакет. Кроме того, у нас есть файлы справки GeneralHelp.rtf, AdminHelp.rtf и файл с пиктограммой Symbol-Help.ico.

Создадим в решении новый проект GeneralSetup, для которого добавим ссылку на проект DemoApplication, на библиотеку WixUIExtension (для демонстрации будем использовать один из стандартных наборов диалогов) а в свойствах проекта, кроме того, укажем на необходимость подключения локализованных ресурсов для русского языка. Подробно эти действия описаны в главе 1, и здесь я постараюсь не повторяться.

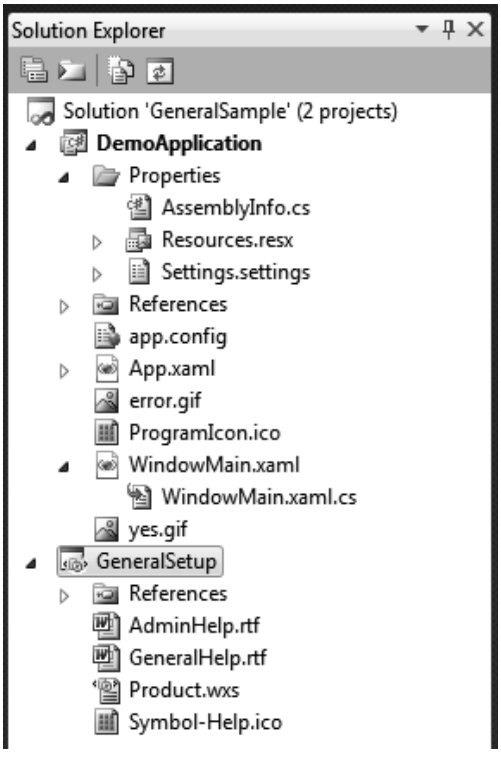

Рисунок 3.1 Демонстрационное решение.

Скопируем в каталог с проектом установки файлы GeneralHelp.rtf, AdminHelp.rtf, Symbol-Help.ico и добавим их в проект. Для этого в Solution Explorer нажмем правой кнопкой мыши на проекте GeneralSetup, укажем пункты Add –> Existing Item…, а затем выберем файлы и нажмем кнопку Add.

Вид из Solution Explorer будет аналогичен тому, который приведен на рисунке 3.1.

Пример

В данном разделе мы создадим базовый пример, который сможет выступать в качестве основы для проверки функциональности всех последующих глав. В этом примере будет создана приближенная к реальной структура каталогов, а в пакет будут включены все имеющиеся в решении файлы.

Таблица 3.1 Структура каталогов демонстрационного решения.

Каталог	Файл	Примечание
%ProgramFiles%		
CompanyName		
Demo Application		Необходимые компоненты (обязательно)
	DemoApplication.exe	Основной исполняемый файл
	DemoApplication.exe.config	Конфигурационный файл
	DemoApplication.pdb	Отладочные символы (опционально)
Doc		Файлы справки
	GeneralHelp.rtf	Руководство пользователя (опционально или для запуска через сеть/с установочного диска)
	AdminHelp.rtf	Руководство администратора (опционально или при первом обращении)
Пуск – Программы		
Demo Application		
	Demo Application	Ярлык для запуска программы
	Удалить Demo Application	Ярлык для удаления программы
	Руководство пользователя	Ярлык для файла GeneralHelp.rtf (при наличии)
	Руководство администратора	Ярлык для файла AdminHelp.rtf (при наличии)
Рабочий стол		
	Demo Application	Ярлык для запуска программы

В процессе установки пользователь сможет выбрать наборы для установки из списка, при этом ему будут доступны следующие наборы компонентов:

- необходимые компоненты;
- отладочные символы;
- справочная информация:
  - руководство пользователя;
  - руководство администратора.

Это уже достаточно функциональный пример и его полное описание занимает почти три страницы текста. Элементы Product и Package описывают общие свойства программы установки. Элемент Media описывает файл-контейнер для содержимого. Группа элементов Directory предназначена для формирования структуры каталогов. DirectoryRef и Component описывают все устанавливаемые компоненты: файлы, ярлыки, записи в реестр. Элементы Feature собирают

подготовленные компоненты в наборы и определяют способы их установки. Для описания интерфейса используются элементы UI, кроме того, для подключения пиктограмм используются элементы Icon. Готовое содержимое файла сценария установки имеет следующий вид:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi" >
  <Product Id="????????-DEF7-4A8A-A456-8C218B0B25D2" Name="Demo Application" Language="1049"
Codepage="1251" Version="1.0.10" Manufacturer="Evgeniy Vodnev" UpgradeCode="????????-B8B7-
43C7-B517-ED7B290750D2">
    <Package InstallerVersion="200" Compressed="yes" SummaryCodepage="1251"
Description="Программа установки Demo Application" Manufacturer="Evgeniy Vodnev"
Comments="Пакет установки демонстрационного приложения" />
    <Media Id="1" Cabinet="demoApp.cab" EmbedCab="yes" />
    <!-- Виртуальный каталог -->
    <Directory Id="TARGETDIR" Name="SourceDir">
      <!-- Подкаталог Program Files\CompanyName\Demo Application -->
      <Directory Id="ProgramFilesFolder">
        <Directory Id="ManufacturerFolder" Name="CompanyName">
          <Directory Id="INSTALLLOCATION" Name="Demo Application" >
            <!-- Подкаталог Doc -->
            <Directory Id="HelpFilesFolder" Name="Doc" />
          </Directory>
        </Directory>
      </Directory>
      <!-- Пуск\Программы\Demo Application -->
      <Directory Id="ProgramMenuFolder">
        <Directory Id="ProgramMenuDir" Name="Demo Application" />
      </Directory>
      <!-- Рабочий стол -->
      <Directory Id="DesktopFolder" />
    </Directory>
    <!-- Элементы в каталоге с программой -->
    <DirectoryRef Id="INSTALLLOCATION">
      <!-- Исполняемый файл и ярлыки -->
      <Component Id="MainExecutable" Guid="????????-C12A-4221-883C-70464A370AB4">
        <File Id="DemoApplicationExe" Name="$(var.DemoApplication.TargetFileName)"
Source="$(var.DemoApplication.TargetPath)" DiskId="1" KeyPath="yes" >
          <!-- Ярлыки для запуска программы -->
          <Shortcut Id="DesktopShortcut" Name="Demo Application" Description="Ярлык на рабочем
столе" WorkingDirectory="INSTALLLOCATION" Directory="DesktopFolder" Advertise="yes"
Icon="ProgramIcon.exe"></Shortcut>
```

```
          <Shortcut Id="ProgramsMenuShortcut" Name="Запустить Demo Application"
Description="Запускает программу" WorkingDirectory="INSTALLLOCATION"
Directory="ProgramMenuDir" Advertise="yes" Icon="ProgramIcon.exe"></Shortcut>
        </File>
      </Component>
      <!-- Конфигурационный файл -->
      <Component Id="MainExecutableConfig" Guid="????????-44F0-B23B-D3D550DBF0F7">
        <File Id="DemoApplicationExeConfig" Name="DemoApplication.exe.config"
Source="$(var.DemoApplication.TargetDir)" DiskId="1" KeyPath="yes" />
      </Component>
      <!-- Отладочные символы -->
      <Component Id="MainExecutablePdb" Guid="????????-4CB4-4F19-A368-7476C4DC2412">
        <File Id="DemoApplicationPdb" Name="DemoApplication.pdb"
Source="$(var.DemoApplication.TargetDir)" DiskId="1" KeyPath="yes" />
      </Component>
    </DirectoryRef>
    <!-- Каталог в меню "Пуск". Удаление при деинсталляции -->
    <DirectoryRef Id="ProgramMenuDir">
      <Component Id="ProgramMenuDir" Guid="????????-B68E-401A-8625-2EA138EEA114">
        <RemoveFolder Id="ProgramMenuDir" On="uninstall" />
        <RegistryValue Root="HKCU" Key="Software\[Manufacturer]\[ProductName]" Type="string"
Value="" KeyPath="yes" />
      </Component>
    </DirectoryRef>
    <DirectoryRef Id="HelpFilesFolder">
      <!-- Файл руководства пользователя -->
      <Component Id="GeneralHelp" Guid="????????-7389-4265-8F3A-4C328656BB5C"
Location="either">
        <File Id="GeneralHelpRtf" Name="GeneralHelp.rtf" DiskId="1" KeyPath="yes"
Compressed="no">
          <Shortcut Id="GeneralHelpShortcut" Name="Руководство пользователя"
Description="Основное руководство пользователя" WorkingDirectory="HelpFilesFolder"
Directory="ProgramMenuDir" Advertise="yes" Icon="HelpIcon.ico" />
        </File>
      </Component>
      <!-- Файл руководства администратора -->
      <Component Id="AdminHelp" Guid="????????-8757-49D3-8C5F-B210BD3D00C5">
        <File Id="AdminHelpRtf" Name="AdminHelp.rtf" DiskId="1" KeyPath="yes" >
          <Shortcut Id="AdminHelpShortcut" Name="Руководство администратора"
Description="Руководство администратора программы" WorkingDirectory="HelpFilesFolder"
Directory="ProgramMenuDir" Advertise="yes" Icon="HelpIcon.ico" />
        </File>
```

```
      </Component>
    </DirectoryRef>
    <!-- Элементы в главном меню -->
    <DirectoryRef Id="ProgramMenuDir">
      <!-- Ярлык для удаления программы -->
      <Component Id="ProgramsMenuShortcut" Guid="????????-ABF5-470A-AE4A-8F0EA13A514C">
        <Shortcut Id="UninstallProduct" Name="Удалить Demo Application"
Target="[System64Folder]msiexec.exe" Arguments="/x [ProductCode]" Description="Удаляет Demo
Application с данного компьютера" />
        <RegistryValue Root="HKCU" Key="Software\[Manufacturer]\[ProductName]"
Name="installed" Type="integer" Value="1" KeyPath="yes" />
      </Component>
    </DirectoryRef>
    <!-- Наборы компонентов -->
    <Feature Id="Complete" Title="Demo Application" Description="Полная установка"
Display="expand" Level="1" ConfigurableDirectory="INSTALLLOCATION" AllowAdvertise="no"
Absent="disallow" InstallDefault="local">
      <Feature Id="RequiredComponents" Title="Необходимые компоненты" Description="Важные
компоненты" Level="1" AllowAdvertise="no" Absent="disallow" InstallDefault="local">
        <ComponentRef Id="MainExecutable" />
        <ComponentRef Id="MainExecutableConfig" />
        <ComponentRef Id="ProgramMenuDir" />
        <ComponentRef Id="ProgramsMenuShortcut" />
      </Feature>
      <Feature Id="ExecutableSymbols" Title="Отладочные символы" Description="Дополнительные
библиотеки для отладки." Level="2" AllowAdvertise="no" Absent="allow" InstallDefault="local">
        <ComponentRef Id="MainExecutablePdb" />
      </Feature>
      <Feature Id="HelpFiles" Title="Справочная информация" Description="Справочная информация
о программе" Level="2" AllowAdvertise="no" Absent="disallow" InstallDefault="source">
        <Feature Id="GeneralHelpFiles" Title="Руководство пользователя"
Description="Руководство пользователя программы" Level="2" AllowAdvertise="no"
Absent="disallow" InstallDefault="source">
          <ComponentRef Id="GeneralHelp" />
        </Feature>
        <Feature Id="AdditionalHelpFiles" Title="Руководство администратора"
Description="Руководство администратора программы" Level="2" AllowAdvertise="system"
Absent="allow" InstallDefault="local">
          <ComponentRef Id="AdminHelp" />
        </Feature>
      </Feature>
    </Feature>
```

```
     <!-- Пиктограммы -->
    <Icon Id="ProgramIcon.exe" SourceFile="$(var.DemoApplication.TargetPath)" />
    <Icon Id="HelpIcon.ico" SourceFile="Symbol-Help.ico" />
    <!-- Интерфейс пользователя -->
    <UI Id="MyWixUI_Mondo">
      <UIRef Id="WixUI_Mondo" />
      <UIRef Id="WixUI_ErrorProgressText" />
    </UI>
  </Product>
</Wix>
```

Свойства программы и пакета

Свойства программы указываются в элементе Product, в то время как в Package описываются параметры установочного пакета. Оба эти элемента являются обязательными, назначение их основных атрибутов рассматривается в главе 1.

Кроме того, Package предоставляет ряд дополнительных атрибутов, наиболее важными из которых являются:

- InstallerVersion – номер версии Windows Installer, требуемой для установки данного пакета. Записывается как [старший номер версии] * 100 + [младший номер версии]. Например, для Windows Installer 4.5 мы получим значение «405». Указание данного атрибута имеет значение в тех случаях, когда в пакете используются функции, недоступные в ранних версиях установщика;
- InstallPrivileges – используется в Windows Vista и старше. Принимает значения elevated и limited – соответственно, требуются или нет повышенные привилегии для установки данного пакета. Значение по умолчанию – elevated;
- InstallScope – установка программы для всех пользователей – perMachine или только для текущего – perUser;
- Platform – поддерживаемая аппаратная платформа. Допустимыми значениями являются x86, x64 и ia64.

Компонент – контейнер для ресурсов

Все ресурсы, участвующие в развертывании, должны быть описаны внутри элементов Component - компонентов, которые, в свою очередь, размещаются в каталогах. Прежде всего, это файлы, копируемые программой установки. Однако компоненты также используются и для размещения части операций. Например, элемент RemoveFolder, используемый для удаления каталога, может находиться только внутри компонента. Компонент должен иметь ссылку на каталог. Этого можно достичь тремя методами: вложить его в Directory, поместить внутрь элемента DirectoryRef или поместить в элемент Product, установив для тега Component значение атрибута Directory. Первый способ я, как правило, не использую, стараясь лучше структурировать пакет. Другие два подхода приведены ниже, а получаемый результат одинаков.

```
    <Component Id="EmptyComponent1" Guid="????????-8966-428A-96C9-1DDC019E8296"
Directory="INSTALLLOCATION">
      <!-- Здесь должно быть содержимое -->
    </Component>

    <DirectoryRef Id="INSTALLLOCATION">
      <Component Id="EmptyComponent2" Guid="????????-DF5C-48B7-A16D-2DA2511B1D54" >
        <!-- Здесь должно быть содержимое -->
      </Component>
    </DirectoryRef>
```

Компонент является минимальной единицей развертывания; рекомендуется создавать по одному компоненту на один копируемый файл. Факт установки компонента контролируется по наличию или отсутствию данного файла, который в данном случае является ключевым. При наличии в компоненте нескольких сущностей только одна из них будет ключевой, для чего необходимо установить для нее в yes значение атрибута KeyPath.

В случае, когда компонент устанавливается в папку, специфичную для пользователя (например, в меню Пуск –> Программы) или не содержит файлов, для контроля необходимо использовать запись в пользовательском ключе реестра.

```
        <RegistryValue Root="HKCU" Key="Software\[Manufacturer]\[ProductName]"
Name="installed" Type="integer" Value="1" KeyPath="yes" />
```

Такая форма записи используется для создания ярлыков. Подробно этот вопрос будет рассмотрен в разделе, посвященном работе с реестром.

Элемент Component имеет ряд дополнительных атрибутов; к числу наиболее важных следует отнести:

- Permanent – установка в yes запрещает удаление содержимого компонента при деинсталляции;
- NeverOverwrite – присваивание значения yes заставляет установщик не обновлять содержимое компонента в тех случаях, когда ключевой файл или запись реестра уже существует;
- Location – необходим, если содержащийся внутри компонента файл должен запускаться с диска или через сеть. Допустимыми значениями являются: source – запуск только с источника; either – запуск с источника или локальная установка. Значение local – только возможность локальной установки – указывать не обязательно, применяется по умолчанию. Подробно использование описано в разделе, посвященном работе с наборами;
- Win64 – применяется в случае, когда один пакет устанавливает одновременно как 32-х, так и 64-битные компоненты. Для последних необходимо установить данный атрибут в yes.

Работа с каталогами

Определение структуры каталогов для устанавливаемой программы является одной из первоочередных задач. Для успешного копирования файлов необходимо указать все каталоги, задействованные при установке, в том числе, при необходимости, добавить ссылки на рабочий стол, главное меню и другие стандартные папки. Каталоги описываются элементом Directory, в атрибут Id помещается идентификатор, а в атрибут Name – название каталога. Идентификаторы каталогов должны быть уникальны в пределах пакета.

Первым вложенным в Product элементом всегда должен являться виртуальный корневой каталог с идентификатором TARGETDIR и именем SourceDir. Все физические каталоги описываются внутри этого элемента:

```
<!-- Виртуальный каталог -->
<Directory Id="TARGETDIR" Name="SourceDir">
</Directory>
```

Если необходимо создать иерархию каталогов, элементы Directory следует вкладывать друг в друга. Идентификатор каталога, куда будут непосредственно копироваться файлы, следует указывать в верхнем регистре. Это делает переменную открытой и позволяет изменять ее значение, например, в элементах интерфейса пользователя.

```
<!-- Виртуальный каталог -->
<Directory Id="TARGETDIR" Name="SourceDir">
  <!-- Подкаталог Program Files\CompanyName\Demo Application -->
  <Directory Id="ProgramFilesFolder">
    <Directory Id="ManufacturerFolder" Name="CompanyName">
      <Directory Id="INSTALLLOCATION" Name="Demo Application" />
    </Directory>
  </Directory>
  <!-- Пуск\Программы\Demo Application -->
  <Directory Id="ProgramMenuFolder">
    <Directory Id="ProgramMenuDir" Name="Demo Application" />
  </Directory>

  <!-- Рабочий стол -->
  <Directory Id="DesktopFolder" />
</Directory>
```

Существует множество системных идентификаторов, каждый из которых описывает пути к конкретным папкам. Например, идентификатор ProgramMenuFolder описывает путь к меню Пуск - Программы, а WindowsFolder – к системному каталогу Windows. При использовании системного идентификатора атрибут Name не используется и его можно не указывать. Подробнее данный вопрос рассматривается ниже в этой главе, в разделе «Стандартные пути и их аналоги в управляемом коде», а исчерпывающая информация на английском языке содержится в библиотеке MSDN – ссылка на соответствующий раздел есть в приложении.

Добавление каталогов

Описанные каталоги создаются автоматически при копировании в них содержимого компонентов. Если никакие файлы не копируются, то и соответствующий каталог создан не будет. Для тех случаев, когда нам необходима пустая папка, предназначен элемент CreateFolder. Опишем компонент с единственным элементом CreateFolder внутри. При установке он будет создавать в каталоге INSTALLOCATION папку NewFolder:

```
    <DirectoryRef Id="INSTALLLOCATION">
      <Directory Id="NewFolderDirectory" Name="NewFolder">
        <Component Id="CreateNewFolder" Guid="????????-1E81-476E-B818-A3115FC14635">
          <CreateFolder />
        </Component>
      </Directory>
    </DirectoryRef>
```

Ссылку на созданный компонент необходимо добавить в один из наборов. Если мы забудем это сделать, в процессе сборки проекта будет выдано сообщение об ошибке.

Удаление каталогов

Каталоги, создаваемые в процессе установки в общих разделах файловой системы (например, в папке Program Files), автоматически удаляются при деинсталляции. Чуть большей заботы требуют папки, создаваемые компонентами в пользовательских каталогах. Кроме необходимости контролировать их наличие на основании ключей в реестре, их также необходимо явно удалять. Для этих целей создается компонент, фактически наполненный только логикой удаления:

```
    <DirectoryRef Id="ProgramMenuDir">
      <Component Id="ProgramMenuDir" Guid="????????-B68E-401A-8625-2EA138EEA114">
        <RemoveFolder Id="ProgramMenuDir" On="uninstall" />
        <RegistryValue Root="HKCU" Key="Software\[Manufacturer]\[ProductName]" Type="string"
Value="" KeyPath="yes" />
      </Component>
    </DirectoryRef>
```

Некоторые каталоги создаются программами уже во время работы, например, для хранения в них файлов журналов. Их удаление также необходимо предусмотреть, для чего служит элемент RemoveFolder. Атрибут On предоставляет возможность выбора, в каких случаях следует произвести удаление: при установке (install), при удалении (uninstall) или в обоих случаях (both). Пример «чистого» удаления каталогов вместе с файлами рассматривается ниже, в разделе «Полное удаление файлов».

Поиск каталогов

Чтобы найти каталог, следует использовать элемент DirectorySearch. Полученное значение может быть использовано вложенными элементами для поиска, например, файлов. Если необходимо присвоить найденное значение свойству, следует установить значение атрибута AssignToProperty в

yes. Значение атрибута Depth определяет глубину вложенности поиска. Если значение не указано, оно приравнивается нулю, что определяет необходимость поиска только в текущей папке.

Например, так можно определить наличие каталога %ProgramFiles%\Common Files:

```
    <Property Id="COMMONFILESDIRECTORYPATH">
      <DirectorySearch Id="DirectorySearch1" Path="[ProgramFilesFolder]">
        <DirectorySearch Id="DirectorySearch2" Path="Common Files" Depth="0"
AssignToProperty="yes" />
      </DirectorySearch>
    </Property>
```

В случае наличия в файловой системе данного каталога переменная COMMONFILESDIRECTORYPATH получит значение вида C:\Program Files\Common Files.

Стандартные пути и их аналоги в управляемом коде

Существует ряд системных каталогов, физические пути к которым неодинаковы в различных версиях операционных систем. В Windows Installer определены предустановленные свойства, возвращающие пути к указанным каталогам. В управляемом коде для этих целей служит метод System.Environment.GetFolderPath, в качестве аргумента для которого выступает элемент перечисления System.Environment.SpecialFolder. Например, в Windows 7 приведенный ниже фрагмент кода возвращает путь C:\Users\<имя_пользователя>\AppData\Local, а в Windows XP этот же код возвращает путь C:\Documents and Settings\<имя_пользователя>\Local Settings\Application Data:

```
String localApplicationDataFolder =
Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
```

Чтобы получить путь к этому же каталогу в Windows Installer, следует воспользоваться переменной LocalAppDataFolder. Сложность заключается в том, что взаимное соответствие переменных не всегда очевидно, а в некоторых случаях аналоги вообще отсутствуют.

Когда это важно? Допустим, наше приложение записывает информацию о ходе функционирования в файл журнала, находящийся в каталоге, соответствующем элементу Environment.SpecialFolder.LocalApplicationData. Это связано с тем, что по умолчанию в Windows Vista и Windows 7 приложение запускается с правами Standard User и не имеет прав на запись в некоторые каталоги, в том числе в Program Files. Конечно, можно воспользоваться возможностями записи в изолированное хранилище – Isolated Storage или положиться на файловую виртуализацию, го в таком случае мы теряем возможность вычисления физического пути к данным файлам (например, для открытия файлов журнала с помощью внешнего приложения). В этом случае можно воспользоваться общими каталогами, доступными для записи без запроса на превышение полномочий. Данные каталоги и файлы должны быть доступны программе установки для выполнения полной деинсталляции и, возможно, каких-либо других действий. В этом случае может пригодится приведенная ниже информация о соответствии переменных.

Таблица 3.2 Стандартные пути Windows Installer и их аналоги в управляемом коде.

Переменная	Элемент	Физический путь для Windows 7 (32-х битная)

Windows Installer	перечисления Environment. SpecialFolder	
AdminToolsFolder	нет	
AppDataFolder	ApplicationData	C:\Users\<имя_пользователя>\AppData\Roaming
CommonAppDataFolder	CommonApplicationData	C:\ProgramData
CommonFiles64Folder	нет	
CommonFilesFolder	CommonProgramFiles	C:\Program Files\Common Files
DesktopFolder	Desktop или DesktopDirectory	C:\Users\<имя_пользователя>\Desktop
FavoritesFolder	Favorites	C:\Users\<имя_пользователя>\Favorites
FontsFolder	нет	
LocalAppDataFolder	LocalApplicationData	C:\Users\<имя_пользователя>\AppData\Local
MyPicturesFolder	MyPictures	C:\Users\<имя_пользователя>\Pictures
PersonalFolder	Personal или MyDocuments	C:\Users\<имя_пользователя>\Documents
ProgramFiles64Folder	нет	
ProgramFilesFolder	ProgramFiles	C:\Program Files
ProgramMenuFolder	Programs	C:\Users\<имя_пользователя>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs
SendToFolder	SendTo	C:\Users\<имя_пользователя>\AppData\Roaming\Microsoft\Windows\SendTo
StartMenuFolder	StartMenu	C:\Users\<имя_пользователя>\AppData\Roaming\Microsoft\Windows\Start Menu
StartupFolder	Startup	C:\Users\<имя_пользователя>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup
System16Folder	нет	
System64Folder	нет	
SystemFolder	System	C:\Windows\system32
TempFolder	нет	
TemplateFolder	Templates	C:\Users\<имя_пользователя>\AppData\Roaming\Microsoft\Windows\Templates
WindowsFolder	нет	
WindowsVolume	нет	
нет	Cookies	C:\Users\<имя_пользователя>\AppData\Roaming\Microsoft\Windows\Cookies
нет	History	C:\Users\<имя_пользователя>\AppData\Local\Microsoft\Windows\History
нет	InternetCache	C:\Users\<имя_пользователя>\AppData\Local\Microsoft\Windows\Temporary Internet Files
нет	MyMusic	C:\Users\<имя_пользователя>\Music
нет	Recent	C:\Users\<имя_пользователя>\AppData\Roaming\Microsoft\Windows\Recent

Работа с файлами

Без копирования и удаления файлов не обходится, пожалуй, ни одна программа установки. Для этих целей служит элемент File, размещаемый внутри компонентов, по одному на каждый копируемый файл. Обязательным элементом каждого файла является ссылка на контейнер,

внутрь которого он будет помещаться в процессе сборки пакета, поэтому в начале раздела рассмотрим назначение элемента Media.

Элемент Media – контейнер для содержимого

В каждом сценарии установки должен присутствовать хотя бы один элемент Media. Он описывает файл-контейнер, хранящий содержимое программы установки.

```
    <!-- Файл-контейнер для содержимого -->
    <Media Id="1" Cabinet="demoApp.cab" EmbedCab="yes" />
```

В приведенном выше фрагменте создается файл demoApp.cab, который помещается внутрь пакета msi (атрибуту EmbedCab присвоено значение yes). Далее в файле сценария в каждой паре компонент/файл хотя бы один из них должен иметь атрибут DiskId, значение которого устанавливается равным идентификатору существующего элемента Media. Если значение DiskId установлено и для компонента, и для файла, то используется значение в элементе File. В случае, когда в установочном пакете описан единственный контейнер, значение атрибута DiskId можно не указывать.

```
        <File Id="DemoApplicationPdb" Name="DemoApplication.pdb"
Source="$(var.DemoApplication.TargetDir)" DiskId="2" KeyPath="yes" />
```

Описание двух и более элементов Media необходимо тогда, когда ваша программа установки не помещается на одном носителе. Тогда их описание выглядит так:

```
    <Media Id="1" Cabinet="media1.cab" EmbedCab="yes" />
    <Media Id="2" DiskPrompt="Demo Application - диск #2" VolumeLabel="JOB2"
Cabinet="media2.cab" EmbedCab="no" />
    <Property Id="DiskPrompt" Value="[1]" />
```

Для контейнера media2.cab атрибут EmbedCab установлен в no, что отключает его внедрение внутрь msi-пакета; также для него указано значение атрибута DiskPrompt. Одновременно мы объявляем переменную DiskPrompt. Когда процесс копирования достигнет файлов, хранящихся на недоступном в настоящий момент носителе, пользователю будет отображен соответствующий запрос. Для этого будет использован текст, хранящийся в переменной DiskPrompt, где вместо [1] будет подставлено значение атрибута DiskPrompt требуемого элемента Media. Также необходимо указать значение атрибута VolumeLabel – метки диска, которая будет использоваться для контроля вставленного носителя: дискеты, CD или DVD–диска или USB-накопителя.

Замечание: проверка наличия следующего носителя будет производиться только в том случае, если исходный msi-пакет запускается с внешнего носителя. В случае запуска с жесткого диска и отсутствии дополнительных cab-файлов установка будет прервана с сообщением об ошибке.

Копирование файлов

Для копирования файлов их следует заключить внутрь тега File. Как правило, удобнее использовать последовательность DirectoryRef – Component – File, размещая ее внутри элемента Product:

```
    <DirectoryRef Id="INSTALLLOCATION">
```

```
      <Component Id="MainExecutablePdbComponent" Guid="????????-4CB4-4F19-A368-7476C4DC2412">
        <File Id="DemoApplicationPdb" Name="DemoApplication.pdb"
Source="$(var.DemoApplication.TargetDir)" DiskId="1" KeyPath="yes" />
      </Component>
    </DirectoryRef>
```

Здесь сначала указывается ссылка на идентификатор определенного ранее каталога, для чего используется атрибут Id элемента DirectoryRef. Внутри создается компонент и в него вкладывается элемент File. Для каждого файла необходимо указывать произвольный уникальный идентификатор, значения атрибутов Name (имя файла) и Source (источник файла) В таблице 3.3 приведены рекомендуемые значения атрибутов Name и Source для различных файлов. Вместо DemoApplication следует использовать имя проекта, ссылка на который добавлена в проект установки.

Таблица 3.3 Значения атрибутов Name и Source.

Файл	Значение атрибута Name	Значение атрибута Source
Основной файл сборки в добавленном проекте, например DemoApplication.exe	$(var.DemoApplication.TargetFileName)	$(var.DemoApplication.TargetPath)
Все остальные файлы в добавленном проекте, например DemoApplication.exe.config	Имя файла без пути (DemoApplication.exe.config)	$(var.DemoApplication.TargetDir)
Файлы, добавляемые в проект установочного пакета, например AdminHelp.rtf	Указание атрибута необязательно	Имя файла без пути (AdminHelp.rtf)

Ниже описаны основные атрибуты, которые могут использоваться в различных случаях:

- DiskId – используется для указания идентификатора элемента Media; если в проекте единственный элемент Media, то этот атрибут можно не указывать;
- KeyPath – обязателен в том случае, если родительский компонент содержит более одного вложенного элемента. Он используется для контроля факта установки компонента в системе, при этом для ключевого элемента KeyPath устанавливается в yes, а для всех остальных – в no.
- ReadOnly, Hidden, System – если установить в yes какие-либо из этих атрибутов, скопированный файл будет помечен как «только для чтения», «скрытый» или «системный» соответственно;
- Assembly – установка для данного атрибута значения .net помечает файл как .NET сборку, которую необходимо поместить в Global Assembly Cache. Следует помнить, что для этого сборка должна быть подписана.

Создание ярлыков и пиктограмм

Создание ярлыков является неотъемлемой частью практически любой программы установки. И действительно, все мы привыкли ожидать, что пиктограмма установленной программы будет найдена в меню «Пуск» и, возможно, на рабочем столе.

В приведенном ниже примере ярлыки, описываемые элементом Shortcut, вложены в элемент File. Целесообразно придерживаться такого подхода, когда это применимо, но ничто не мешает описывать ярлыки отдельно. Для ярлыков указываются следующие атрибуты:

- Id – уникальный идентификатор элемента;
- Name – отображаемое имя;
- Directory – ссылка на идентификатор каталога, где будет располагаться ярлык;
- IconIndex – если в файле несколько пиктограмм, то для указания необходимой используется данный атрибут. По умолчанию его значение равно нулю и оно указывает на первое изображение в списке;
- Description – необязательное описание, отображаемое во всплывающей подсказке при наведении курсора мыши.

Рассмотрим два наиболее часто встречающихся случая: пиктограмма встроена в исполняемый файл или представлена отдельным файлом. Если пиктограмма встроена в исполняемый файл, то ее достаточно подключить следующим образом:

```
    <Icon Id="ProgramIcon.exe" SourceFile="$(var.DemoApplication.TargetPath)" />
```

Атрибут SourceFile указывает на файл, содержащий пиктограмму. Следует обратить внимание, что в идентификаторе пиктограммы должно содержаться расширение, такое же, как и у файла с пиктограммой. Так как в данном случае изображение находится внутри .exe файла, это же расширение получит идентификатор ProgramIcon.exe. Чтобы использовать эту пиктограмму, достаточно установить значение атрибута Icon соответствующего элемента в ProgramIcon.exe.

Если же пиктограмма представлена отдельным файлом, то сначала ее необходимо добавить в проект через элементы меню Add – Existing Item, а затем подключить, явно указав имя файла:

```
    <Icon Id="HelpIcon.ico" SourceFile="Symbol-Help.ico" />
```

Теперь этой пиктограммой также можно пользоваться.

Отдельно следует рассмотреть атрибут Advertise. Если его значение установить в yes, то ярлык становится неотъемлемым от объекта-владельца. При этом он более не является простым указателем на файл, наследует часть его свойств и управляется службами Windows Installer. Такие ярлыки необходимы для установки наборов по требованию или для запуска содержимого с установочного диска. Фактически, такой ярлык не является указателем на физический файл, а представляет собой интерфейс для доступа к объекту. В дальнейшем такие ярлыки мы будем называть управляемыми.

```
      <Component Id="MainExecutable" Guid="????????-C12A-4221-883C-70464A370AB4">
        <File Id="DemoApplicationExe" Name="$(var.DemoApplication.TargetFileName)"
Source="$(var.DemoApplication.TargetPath)" DiskId="1" KeyPath="yes" >
          <!-- Ярлыки для запуска программы -->
          <Shortcut Id="DesktopShortcut" Name="Demo Application" Description="Ярлык на рабочем
столе" Directory="DesktopFolder" Advertise="yes" Icon="ProgramIcon.exe"></Shortcut>
          <Shortcut Id="ProgramsMenuShortcut" Name="Запустить Demo Application"
Description="Запускает программу" WorkingDirectory="INSTALLLOCATION"
Directory="ProgramMenuDir" Advertise="yes" Icon="ProgramIcon.exe"></Shortcut>
```

```
        </File>
```

Если же мы создаем независимый ярлык, то для него Advertise устанавливается в no. При этом необходимо указать значение атрибута WorkingDirectory – рабочий каталог, где находится ссылаемый объект и значение атрибута Target, в котором размещается ссылка на целевой файл. Если указать WorkingDirectory, но опустить Target, то при выборе этого ярлыка будет открываться проводник, отображающий перечень содержимого в каталоге WorkingDirectory.

```
      <Component Id="ProgramsMenuShortcut" Guid="????????-ABF5-470A-AE4A-8F0EA13A514C">
        <RegistryValue Root="HKCU" Key="Software\[Manufacturer]\[ProductName]"
Name="installed" Type="integer" Value="1" KeyPath="yes" />
        <Shortcut Id="AdminHelpShortcut" Name="Руководство администратора"
Description="Руководство администратора программы" WorkingDirectory="HelpFilesFolder"
Directory="ProgramMenuDir" Advertise="no" Icon="HelpIcon.ico"
Target="[HelpFilesFolder]\AdminHelp.rtf" />
      </Component>
```

Копирование .NET сборок в GAC

Установка .NET сборок в глобальный кэш сборок (Global Assembly Cache) позволяет разделять общий функционал между различными приложениями, одновременно решая проблему несовместимости различных версий. Это достигается введением требования наличия у библиотеки так называемого строгого имени, без которого ее размещение в GAC невозможно.

Для установки предварительно подписанной библиотеки в кэш сборок достаточно пометить соответствующий файл атрибутом Assembly со значением .net, а также установить в yes атрибут KeyPath.

```
      <Component Id="ProductComponent" Guid="????????-72CE-42EE-A6C1-70236E3340E6">
        <File Id="StrongNamedLibraryDll" Name="StrongNamedLibrary.dll"
Source="$(var.StrongNamedLibrary.TargetDir)" Assembly=".net" KeyPath="yes" />
      </Component>
```

Несмотря на то, что компонент всегда привязан к тому или иному каталогу, в данном случае фактическое копирование в него не будет выполнено – файл будет установлен в GAC.

Установка шрифтов

Установка как True Type, так и Open Type шрифтов с помощью Windows Installer XML выполняется очень просто. Прежде всего, в описание структуры папок следует добавить ссылку на каталог FontsFolder и поместить внутрь него компонент с файлом шрифта. Затем необходимо пометить указанный файл атрибутом TrueType со значением yes.

```
    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="FontsFolder" />
    </Directory>
...
    <DirectoryRef Id="FontsFolder">
      <Component Id="Font1Ttf" Guid="????????-7516-49B8-AC08-3143616FF354">
```

```
        <File Id="Font1Ttf" Source="font1.ttf" TrueType="yes" />
      </Component>
    </DirectoryRef>
```

В данном случае предполагается, что файл font1.ttf был добавлен в проект программы установки через меню Add -> Existing Item. Соответственно, при деинсталляции пакета шрифт будет удален из системы.

Принудительная перезапись файлов

В некоторых случаях устанавливаемое содержимое должно перезаписывать уже находящиеся на диске файлы. Если не указано иное, то файлы перезаписываются только в тех случаях, когда версия устанавливаемого файла старше версии имеющегося. Чтобы принудительно перезаписать содержимое, следует изменить значение стандартной переменной REINSTALLMODE.

```
    <Property Id="REINSTALLMODE" Value="amus" />
```

Каждая буква в значении имеет свой смысл. Значение по умолчанию – omus. Остальные варианты используются существенно реже, поэтому мы их рассматривать не будем – подробно о них написано в библиотеке MSDN.

Поиск файлов и каталогов

Иногда нам необходимо извлечь полный путь к файлу или к каталогу, содержащему указанный файл. Для этого используется сочетание элементов DirectorySearch и FileSearch. Результат поиска присваивается свойству, внутрь которого помещены элементы. Если файл не найден, никакое значение свойству не присваивается. Свойство, в которое помещается результат поиска, должно быть открытым – public, для этого его название пишется заглавными буквами.

В приведенном ниже примере производится поиск каталога со стандартным идентификатором SystemFolder (чаще это «C:\Windows\System32\»). Атрибут Depth устанавливает необходимый уровень вложенности поиска. Нулевое значение диктует необходимость поиска только в указанном каталоге. Затем в найденной папке ищется файл с именем compmgmt.msc. Результат поиска присваивается переменной COMPMGMTPATH.

```
    <Property Id="COMPMGMTPATH">
      <DirectorySearch Id="DirectorySearchInstallFolder" Path="[SystemFolder]" Depth="0">
        <FileSearch Id="FileSearchComputerManagementMsc" Name="compmgmt.msc" />
      </DirectorySearch>
    </Property>
```

Если нас интересует не полное имя файла, а только путь к нему, тогда для элемента DirectorySearch следует установить в yes значение атрибута AssignToProperty. В этом случае вложенный элемент может отсутствовать. При необходимости можно вкладывать элементы DirectorySearch друг в друга.

Полное удаление файлов

Реализация «чистой» деинсталляции является одним из требований для получения логотипа «Compatible with Windows 7», поэтому следует помнить о том, что в процессе работы программа может создавать различные файлы – файлы журналов и данных. Поскольку Windows Installer ничего не знает об этих файлах, то и удалены при деинсталляции они не будут. Их удаление организуется аналогично удалению каталогов. Можно вынести эту логику в отдельный компонент или поместить в один из существующих.

```
          <RemoveFile Id="InformationLog" On="uninstall" Name="Information.log"
Directory="LogFilesLocation" />
```

Атрибут On принимает одно из трех возможных значений: install, uninstall или both – при установке, при удалении или в обоих случаях. Атрибут Name содержит имя удаляемого файла, Directory – ссылку на идентификатор каталога. В имени могут быть использованы подстановочные символы.

Пусть наша программа создает в процессе работы подкаталог с идентификатором LogFilesLocation, куда помещает файлы журналов. Имена файлов генерируются по внутренним правилам и их общее количество нам неизвестно, поэтому немаловажна возможность использования символов-заменителей в именах файлов. Так, приведенный ниже фрагмент позволяет при деинсталляции удалить все файлы из каталога LogFilesFolder, а затем и сам каталог. Элемент CreateFolder необходим, так как в качестве ключевого элемента компонента должен выступать существующий элемент.

```
<Component Id="ClearUninstallation" Guid="????????-1ABA-4149-AE18-C79714851777">
  <CreateFolder Directory="INSTALLLOCATION" />
  <RemoveFile Id="RemoveLogFiles" On="uninstall" Name="*.*" Directory="LogFilesFolder" />
  <RemoveFolder Id="RemoveLogFilesLocation" On="uninstall" Directory="LogFilesFolder" />
</Component>
```

Работа с INI-файлами

Хотя в настоящее время конфигурационная информация размещается преимущественно в XML-файлах, все еще очень большое количество программ использует для служебных целей INI-файлы. Поэтому было бы неправильным обойти вниманием способы их обработки в WiX.

INI-файл представляет собой текстовый файл с расширением ini. Информация в нем располагается в одной или более секциях, а в каждой секции построчно располагаются записи вида «Имя_параметра»=«Значение_параметра»:

```
[Section1Name]
Parameter1=Value1
Parameter2=123
[Section2Name]
OtherParameter=0.9000
…
```

Сохраним приведенный выше фрагмент в виде текстового файла с именем TestIniFile.ini.

Извлечение данных

Извлечение данных выполняется практически аналогично поиску файлов. Следует отметить, что можно извлекать содержимое только тех файлов, которые находятся в каталоге %WINDIR%. Атрибут Name указывает имя файла, Section – имя секции в файле, Key – имя ключа. Атрибут Type указывает тип извлекаемого значения: directory, file или raw. В первом случае извлекается путь к каталогу, во втором – к файлу, а в последнем – текстовое значение.

```
    <Property Id="INIFILEVALUE">
      <IniFileSearch Id="IniFileSearch1" Name="TestIniFile.ini" Type="raw"
Section="Section2Name" Key="OtherParameter" />
    </Property>
```

Пример выше извлекает запись и помещает ее значение в свойство INIFILEVALUE. В файле журнала мы можем увидеть результат вычисления:

```
Property(C): INIFILEVALUE = 0.9000
```

Запись INI-файлов

Мы можем также изменять содержимое INI-файлов, используя элемент IniFile. Как и в случае извлечения данных, обрабатывается только содержимое каталога %WINDIR%. Наиболее важный атрибут – Action, принимающий следующие значения:

- addline – добавляет или обновляет запись;
- addTag – при отсутствии записи работает аналогично addLine, при наличии добавляет новое значение через запятую;
- createLine – создает запись в случае, если она не существует, иначе не делает ничего;
- removeLine – удаляет запись целиком;
- removeTag – удаляет только значение.

Если удаляется единственное значение в секции, то удаляется и секция. Для всех Action кроме removeLine необходимо также указывать значение атрибута Value. Вместе с последним значением будет удален и сам INI-файл.

В примере ниже в файл TestIniFile.ini в секцию Section1Name добавляется ключ Parameter3 со значением Value3, а из секции Section2Name удаляется запись с ключом OtherParameter.

```
      <Component Id="IniFileComponent" Guid="????????-1FE8-4336-B5C6-36825DACAE96">
        <IniFile Id="IniFileWrite1" Action="addLine" Key="Parameter3" Name="TestIniFile.ini"
Section="Section1Name" Value="Value3" />
        <IniFile Id="IniFileWrite2" Action="removeLine" Key="OtherParameter"
Name="TestIniFile.ini" Section="Section2Name" />
      </Component>
```

Работа с реестром

На сегодняшний день реестр является основным местом хранения служебной информации. Windows Installer XML предоставляет для обработки этих данных множество функций – о них рассказывается в данном разделе.

Чтение ключей реестра

Необходимость поиска существующих ключей и извлечения их значений возникает достаточно часто. Для поиска предназначен элемент RegistrySearch, а его использование во многом похоже на поиск файлов. Он вкладывается внутрь свойства, которому и присваивается результат поиска. В примере ниже поиск производится в ветви HKEY_LOCAL_MACHINE – атрибуту Root присвоено значение HKLM. Также допустимы значения HKCR, HKCU и HKU – HKEY_CLASSES_ROOT, HKEY_CURRENT_USER и HKEY_USERS соответственно. Ищется ключ, указываемый в атрибуте Key, имя параметра указывается в атрибуте Name. Если ищется значение параметра по умолчанию, то атрибут Name не указывается.

Атрибут Type принимает значения directory, file или raw. В первом случае ожидается, что ключ содержит адрес каталога, во втором – полный путь к файлу. Тип raw используется для любого другого случая, при этом к строке результата добавляются префиксы, позволяющие определить фактический тип значения. Значение типа DWORD начинается с символа «#», REG_BINARY – начинается с символов «#x», эта же последовательность идет перед каждой шестнадцатеричной цифрой. Тип REG_EXPAND_SZ начинается со строки «#%», REG_MULTI_SZ начинается и заканчивается тильдой «~». К строкам типа REG_SZ префикс не добавляется, за исключением случаев, когда значение начинается с символа «#». В этом случае указанный символ удваивается.

Пример ниже считывает используемую системой оболочку по умолчанию и для систем со стандартными настройками присваивает свойству REGISTRYSEARCHSHELLVALUE значение Explorer.exe

```
    <Property Id="REGISTRYSEARCHSHELLVALUE">
      <RegistrySearch Id="RegistrySearchShellValue" Type="raw" Root="HKLM"
Key="Software\Microsoft\Windows NT\CurrentVersion\Winlogon" Name="Shell" />
    </Property>
```

Пример организации сложного поиска

Периодически возникает необходимость выполнения сложного поиска по ключам реестра. Рассмотрим несколько надуманный, но демонстрирующий такую возможность пример. Допустим, мы хотим получить путь к каталогу, где размещаются надстройки .XLA и .XLAM для MS Excel. Конкретный путь зависит от используемой версии Excel. Это один из подкаталогов папки Microsoft Office: Office10\Library для MS Office 2000, OFFICE11\Library для MS Office XP и OFFICE12\Library для MS Office 2007. Таким образом, задача раскладывается на две: найти путь к запускаемому файлу MS Excel, а затем добавить к полученному значению подкаталог Library.

```
    <!-- Поиск в реестре каталога с Excel-->
    <Property Id="EXCELGUID">
```

```
      <RegistrySearch Id="RegistrySearchExcelApplication" Type="raw" Root="HKCR"
Key="Excel.Application\CLSID" />
    </Property>
    <!-- Извлечение пути к приложению EXCEL.EXE -->
    <Property Id="EXCELPATH">
      <RegistrySearch Id="RegistrySearchExcelFilePath" Type="raw" Root="HKCR"
Key="CLSID\[EXCELGUID]\LocalServer" >
        <DirectorySearch Id="DirectorySearchExcelFolderPath" AssignToProperty="yes" />
      </RegistrySearch>
    </Property>
    <!-- Комбинирование полученного пути -->
    <Property Id="INSTALLLOCATION2">
      <DirectorySearch Id="DirectorySearchExcelLibraryFolderPath" AssignToProperty="yes"
Path="[EXCELPATH]Library" />
    </Property>
```

Как мы видим, решение находится в три действия. Сначала в ветви HKCR мы находим значение ключа Excel.Application\CLSID, получая значение GUID приложения. Строго говоря, следует дополнительно проверить, присвоено ли значение данной переменной. Если переменная не инициализирована, указанный ключ реестра в системе не найден и MS Excel на машине не установлен. Получив идентификатор приложения, мы производим второй поиск, подставляя найденное значение в качестве части ключа (CLSID\[EXCELGUID]\LocalServer). В данном ключе содержится значение вида «C:\PROGRA~1\MICROS~3\Office12\EXCEL.EXE /automation», содержащее в том числе путь к искомому каталогу, поэтому в данный элемент мы вкладываем элемент DirectorySearch. Указанный элемент находит в передаваемом значении путь к каталогу и присваивает результат переменной EXCELPATH. На третьем шаге значение комбинируется и с помощью элемента DirectorySearch присваивается переменной INSTALLLOCATION2.

Если открыть файл журнала Windows Installer, можно увидеть значения переменных после окончания процесса установки. Этот пример выполнялся на машине с установленным MS Excel 2007.

```
Property(C): EXCELGUID = {00024500-0000-0000-C000-000000000046}
Property(C): EXCELPATH = C:\PROGRA~1\MICROS~3\Office12\
Property(C): INSTALLLOCATION2 = C:\PROGRA~1\MICROS~3\Office12\Library\
```

Добавление ключей

Для добавления ключей предназначен элемент RegistryValue. Пример ниже добавляет в ветвь HKEY_USERS новый раздел, при этом значения свойств Manufacturer и ProductName берутся из атрибутов Manufacturer и Name элемента Product. В раздел добавляется строковый параметр NewPropertyName, которому присваивается значение NewPropertyValue. Атрибут Type указывает тип создаваемого параметра, принимая значения string, integer, binary, expandable или multiString.

```
      <Component Id="ManageRegistryKeys" Guid="????????-90CE-4CD3-B76B-49B6EBDED921">
```

```
        <RegistryValue Id="RegistryValueCreateNew" Root="HKCU"
Key="Software\[Manufacturer]\[ProductName]" Type="string" Value="NewPropertyValue"
Name="NewPropertyName" KeyPath="yes" />
      </Component>
```

Указывать значение атрибута Id в данном случае необязательно. При отсутствии он будет сгенерирован из значений идентификатора компонента, а также атрибутов Root, Key и Name.

Еще один атрибут – Action, устанавливаемое для которого по умолчанию значение write необходимо менять только в том случае, когда записывается многострочный параметр. В этом случае Action устанавливается в append или prepend, тем самым очередной параметр будет добавляться, соответственно, в конец или в начало имеющегося содержимого.

```
        <RegistryValue Id="RegistryValueCreateMultiString" Root="HKCU"
Key="Software\[Manufacturer]\[ProductName]" Type="multiString" Name="MultiStringProperty"
Action="append" KeyPath="yes">
          <MultiStringValue>Line 1</MultiStringValue>
          <MultiStringValue>Line 2</MultiStringValue>
          <MultiStringValue>Line 3</MultiStringValue>
        </RegistryValue>
```

Если необходимо создать сразу несколько параметров внутри одного ключа, можно воспользоваться элементом RegistryKey. Атрибуты Root и Key устанавливаются аналогично RegistryValue, идентификатор также необязателен. Атрибут Action принимает следующие значения:

- none - значение по умолчанию, ключ не создается;
- create - ключ создается, но не удаляется при деинсталляции;
- createAndRemoveOnUninstall - ключ создается при установке и удаляется при деинсталляции.

Пример ниже взят из справочной системы Windows Installer XML. В нем в ветви для текущего пользователя создается ключ Software\Microsoft\MyApplicationName, внутри которого создаются два параметра: параметр по умолчанию со значением «Default Value» и параметр типа REG_DWORD со значением 1.

```
        <RegistryKey Root="HKCU" Key="Software\Microsoft\MyApplicationName"
Action="createAndRemoveOnUninstall">
          <RegistryValue Type="integer" Name="SomeIntegerValue" Value="1" KeyPath="yes" />
          <RegistryValue Type="string" Value="Default Value" />
        </RegistryKey>
```

Удаление ключей

Для удаления ключей реестра используются элементы RemoveRegistryValue и RemoveRegistryKey, помещаемые внутрь компонентов. Тег RemoveRegistryValue позволяет удалять отдельные параметры, но только при установке продукта. Атрибут Root указывает ветвь, Key – ключ, а Name – имя параметра. Если атрибут Name не установлен, удаляется значение параметра по умолчанию.

```
        <RemoveRegistryValue Id="RemoveRegistryValue1" Root="HKLM"
Key="Software\CompanyName\ApplicationName" Name="PropertyName" />
```

Если необходимо выполнять действия по очистке реестра при деинсталляции, то следует воспользоваться элементом RemoveRegistryKey, удаляющим ключ целиком.

```
        <RemoveRegistryKey Id="RemoveRegistryKey1" Root="HKLM"
Key="Software\CompanyName\ApplicationName" Action="removeOnUninstall" />
```

Элемент практически идентичен элементу RemoveRegistryValue. У него отсутствует атрибут Name, но добавлен атрибут Action, принимающий значения removeOnUninstall или removeOnInstall – удаление ключа при удалении или установке программы соответственно.

Регистрация расширений файлов

Если ваше приложение является обработчиком файлов определенных расширений, его надо зарегистрировать в системе и настроить ассоциации с данным типом файлов. Элемент ProgId служит для регистрации программы; для регистрации расширения файла в него вкладывается дочерний элемент Extension, в который, в свою очередь, добавляется элемент Verb, регистрирующий связанное с расширением действие. При этом в реестре создается ряд записей, используемых для ассоциации расширения с данным типом файла.

Элемент ProgId, регистрирующий приложение-обработчик, следует размещать в том же компоненте, где находится регистрируемый в качестве обработчика файл. Важно установить значения атрибутов Id и Description. Первый определяет уникальный идентификатор, второй – описание, которое будет отображаться в свойствах файлов зарегистрированного типа. Для связывания типа файла с пиктограммой в атрибуте Icon указывается идентификатор содержащего значок файла, а в IconIndex – порядковый номер пиктограммы. Элемент Extension регистрирует непосредственно расширение, при этом его идентификатором должно быть само расширение. В атрибут ContentType, при необходимости, помещается MIME-тип. И последний вложенный элемент – Verb – описывает действие, связываемое с данным типом файла. Значение атрибута Command отображается в контекстном меню для данного типа файла, TargetFile содержит идентификатор файла-обработчика, а Argument – передаваемые файлу-обработчику аргументы. В нашем примере Argument содержит значение ‘”%1”’. Это означает, что обработчику будет передано полное имя с путем к выбранному файлу. Двойные кавычки используются для корректной передачи имен файлов, возможно содержащих пробелы.

```
      <Component Id="MainExecutable" Guid="????????-C12A-4221-883C-70464A370AB4">
        <File Id="DemoApplicationExe" Name="$(var.DemoApplication.TargetFileName)"
Source="$(var.DemoApplication.TargetPath)" DiskId="1" KeyPath="yes" />

        <ProgId Id="DemoApplication.daefile" Description="Файл Demo Application Extension
(.dae)" Icon="DemoApplicationExe" IconIndex="0">
          <Extension Id="dae">
            <Verb Id="OpenDataFile" Command="Обработать" TargetFile="DemoApplicationExe"
Argument='"%1"' />
          </Extension>
```

```
        </ProgId>
      </Component>
```

Выборочная установка наборов компонентов

После того, как описаны все компоненты, их следует разбить по наборам (Feature). Для описания наборов используются элементы Feature, а для ссылок на компоненты – ComponentRef. В любой программе установки должен быть хотя бы один набор, в этом случае он будет содержать ссылки на все компоненты. В случае наличия единственного набора он описывается очень просто: единственный элемент Feature с несколькими атрибутами, внутри которого размещаются элементы ComponetRef, значение атрибута Id каждого из которых ссылается на идентификатор описанного ранее компонента. Для Feature в данном случае важно указать значения нескольких атрибутов:

- Id – уникальный идентификатор;
- Title и Description – заголовок и описание набора, они отображаются в соответствующем диалоговом окне.
- ConfigurableDirectory – содержит идентификатор каталога, путь к которому будет предложен по умолчанию, но у пользователя будет возможность изменить его.

```
    <!-- Именованные наборы компонентов -->
    <Feature Id="Complete" Title="Simple Application" Description="Полная установка" Level="1"
ConfigurableDirectory="INSTALLLOCATION" >
      <ComponentRef Id="SimpleApplicationExeComponent" />
    </Feature>
```

Атрибут Level связан со стандартным свойством INSTALLLEVEL, которое управляет доступностью и выбором набора:

```
    <Property Id="INSTALLLEVEL" Value="1" />
```

Данное свойство по умолчанию имеет значение 1, явно указывать его нет необходимости. Ненулевые значения атрибута Level, меньшие или равные INSTALLLEVEL, отображают набор в списке как выбранный для установки. Большие значения также отображают набор в списке, но по умолчанию он не выбран для установки – пользователь должен явно сделать выбор. Нулевое значение атрибута скрывает набор из списка. Наличие последнего варианта позволяет добавить в программу установки логику, управляющую доступностью тех или иных наборов.

Для большинства программ установки, как правило, наличие единственного набора является недостаточным. Пользователи выбирают те части продукта, которые их интересуют: указывают, следует ли устанавливать документацию, дополнительные инструменты, языковые пакеты и другие компоненты.

В случае описания более чем одного набора структура несколько усложняется. Иерархия наборов описывается за счет вложения элементов Feature друг в друга. Также возникает необходимость использования дополнительных атрибутов. Пример с использованием нескольких наборов приведен в начале данной главы, здесь мы рассмотрим его подробно. Чтобы сократить описание, здесь удалены атрибуты Description.

Корневой набор с идентификатором Complete является контейнером для всех остальных. В нем располагаются три дочерних набора: RequiredComponents, ExecutableSymbols и HelpFiles. В свою очередь, HelpFiles содержит наборы GeneralHelpFIles и AdditionalHelpFiles.

Атрибуты Id, Title, Description, Level и ConfigurableDirectory описаны выше, возвращаться к ним не будем. Атрибут Display управляет отображением набора в соответствующем диалоге. Он принимает значения collapse (по умолчанию), expand или hidden. Collapse отображает содержимое набора свернутым, expand – развернутым, а hidden скрывает набор.

Атрибут Absent может принимать два значения: allow и disallow. В первом случае пользователь получит возможность не устанавливать набор совсем; во втором – набор будет устанавливаться в любом случае – так следует помечать необходимые для работы компоненты.

```
    <Feature Id="Complete" Title="Demo Application" Display="expand" Level="1"
ConfigurableDirectory="INSTALLFOLDER" AllowAdvertise="no" Absent="disallow"
InstallDefault="local">
      <Feature Id="RequiredComponents" Title="Необходимые компоненты" Level="1"
AllowAdvertise="no" Absent="disallow" InstallDefault="local">
        <ComponentRef Id="MainExecutable" />
        <ComponentRef Id="MainExecutableConfig" />
        <ComponentRef Id="ProgramMenuDir" />
        <ComponentRef Id="ProgramsMenuShortcut"></ComponentRef>
      </Feature>

      <Feature Id="ExecutableSymbols" Title="Отладочные символы" Level="2" AllowAdvertise="no"
Absent="allow" InstallDefault="local">
        <ComponentRef Id="MainExecutablePdb" />
      </Feature>

      <Feature Id="HelpFiles" Title="Справочная информация" Level="2" AllowAdvertise="no"
Absent="disallow" InstallDefault="source">
        <Feature Id="GeneralHelpFiles" Title="Руководство пользователя" Level="2"
AllowAdvertise="no" Absent="disallow" InstallDefault="source">
          <ComponentRef Id="GeneralHelp" />
        </Feature>
        <Feature Id="AdditionalHelpFiles" Title="Руководство администратора" Level="2"
AllowAdvertise="system" Absent="allow" InstallDefault="local">
          <ComponentRef Id="AdminHelp" />
        </Feature>

      </Feature>

    </Feature>
```

Установка наборов по требованию

Кроме непосредственного копирования файлов на целевую машину Windows Installer позволяет устанавливать компоненты по первому к ним обращению – данная возможность называется Installation-On-Demand. Для настройки этого поведения служит атрибут AllowAdvertise. Значение yes включает установку при обращении, значение no – отключает. Вместо yes целесообразно использовать значение system – установка по требованию применяется только в том случае, когда она поддерживается операционной системой.

В примере выше по требованию устанавливается руководство администратора в составе набора AdditionalHelpFiles. Обратим внимание: чтобы иметь возможность обратиться к указанному файлу, необходимо предварительно описать управляемый ярлык, для которого атрибут Advertise установлен в yes:

```
      <Component Id="AdminHelp" Guid="????????-8757-49D3-8C5F-B210BD3D00C5">
        <File Id="AdminHelpRtf" Name="AdminHelp.rtf" DiskId="1" KeyPath="yes" >
          <Shortcut Id="AdminHelpShortcut" Name="Руководство администратора"
Description="Руководство администратора программы" WorkingDirectory="HelpFilesFolder"
Directory="ProgramMenuDir" Advertise="yes" Icon="HelpIcon.ico" />
        </File>
      </Component>
```

Запуск содержимого с источника

В ряде случаев возникает необходимость предоставить пользователю возможность не устанавливать файлы, а запускать их с установочного диска. Например, если идет речь о редко используемых справочных файлах большого размера. Windows Installer поддерживает такое поведение. Прежде всего, для содержащих необходимый файл компонентов следует установить значение атрибута Location в either или source. В первом случае компонент может устанавливаться на машину или запускаться с носителя; во втором – только запускаться с носителя. Кроме того, файл должен находиться в неупакованном виде, то есть не входить в состав CAB-файлов. Как правило, по умолчанию сжатие устанавливается на уровне пакета, поэтому проще отключить его для конкретного файла: атрибут Compressed устанавливается в no. Как и в предыдущем случае, физический объект на диске отсутствует, поэтому для доступа к нему необходим управляемый ярлык:

```
      <Component Id="GeneralHelp" Guid="????????-7389-4265-8F3A-4C328656BB5C"
Location="either">
        <File Id="GeneralHelpRtf" Name="GeneralHelp.rtf" DiskId="1" KeyPath="yes"
Compressed="no">
          <Shortcut Id="GeneralHelpShortcut" Name="Руководство пользователя"
Description="Основное руководство по использованию программы"
WorkingDirectory="HelpFilesFolder" Directory="ProgramMenuDir" Advertise="yes"
Icon="HelpIcon.ico" />
        </File>
      </Component>
```

И в последнюю очередь, атрибуту InstallDefault соответствующего элемента Feature следует присвоить значение source. Другие допустимые значения, local и followParent, первое из которых допускает только установку на компьютер пользователя, а второе неявно присваивает атрибуту такое же значение, как у родительского элемента Feature.

Замечание: если для компонента и файла не установлены значения необходимых атрибутов, сборка пакета пройдет успешно, но в списке наборов будет отсутствовать вариант запуска с источника.

Использование свойств и переменных

Свойства используются для хранения и передачи значений между элементами, для чего предназначен элемент Property. Открытые свойства могут использоваться для передачи значений за пределы пакета, а также их установки при запуске.

В предыдущих разделах неоднократно использовались свойства, что сделало возможным взаимодействие элементов для описания сложной логики. Свойствам присваиваются результаты поиска файлов и папок, значения ключей реестра. Если цель поиска найдена не была, никакое значение свойству не присваивается. В простейшем случае для описания свойства достаточно указать его идентификатор и, при необходимости, начальное значение:

```
<Property Id="TestProperty" Value="TestPropertyValue" />
```

Имена свойств чувствительны к регистру. Чаще всего используются открытые свойства, так как только их можно использовать для взаимодействия с интерфейсом пользователя и для присваивания значений при запуске пакета из командной строки. Только открытые свойства могут выступать в качестве параметров расширенных операций (Custom Action), в них же помещаются результаты любого поиска. Для создания открытого свойства его идентификатор должен содержать только буквы в верхнем регистре и цифры. Открытое свойство выглядит так:

```
<Property Id="PUBLICPROPERTY" />
```

Значения инициализированных свойств попадают в журнал Windows Installer. Если какое-либо свойство не должно отображаться в журнале, например, лицензионный ключ продукта, для него необходимо установить значение атрибута Hidden в yes.

Стандартные свойства Windows Installer

Команда разработчиков Windows Installer предоставила в наше распоряжение большое количество предопределенных свойств. Некоторые из них автоматически вычисляются для текущего пакета; к ним можно отнести, например, ProductCode – уникальный идентификатор продукта и Manufacturer – производителя продукта. Другие используются для воздействия на процесс установки наборов и файлов. Так, уже описанная выше переменная REINSTALLMODE позволяет разрешать конфликты при копировании и замене файлов. Ряд свойств возвращает характеристики операционной системы, а также основные параметры оборудования. Стандартные свойства также содержат пути к системным каталогам, как описано выше.

Стандартных свойств достаточно много, без учета идентификаторов каталогов их более 170. Лучшим местом для получения информации о них является соответствующий раздел библиотеки

MSDN, адреса конкретных страниц приведены в приложении. Все свойства разнесены по одиннадцати категориям:

- местоположения компонентов (Component Location Properties) – служебные свойства, явно не используются или используются в специальных местах;
- свойства конфигурации (Configuration Properties) – свойства продукта, отображаемые на панели управления в разделе удаления программ, управление различными свойствами процесса установки;
- дата и время (Date, Time Properties) – данная группа свойств содержит всего две переменных, позволяющих получить текущую дату и время;
- установка наборов и файлов (Feature Installation Options Properties) – свойства из этой группы позволяют управлять и контролировать то, как будут устанавливаться наборы и файлы;
- аппаратные свойства (Hardware Properties) – архитектура используемого процессора, объем оперативной памяти, параметры экрана;
- свойства состояния установки (Installation Status Properties) – свойства, позволяющие получить дополнительную информацию о ходе установки, например, об окончании свободного места на диске;
- свойства операционной системы (Operating System Properties) – проверка версии операционной системы, пакетов обновлений и наличия наиболее важных служб;
- информация о продукте (Product Information Properties) – основные параметры устанавливаемого пакета: указав единожды, в дальнейшем к ним можно обращаться через свойство;
- Summary Information Update Properties – свойства, обновляющие сводную информацию – данная группа свойств используется только файлами патчей (*.msp) при обновлении административного установочного образа;
- идентификаторы каталогов (System Folder Properties) – описанные ранее свойства, позволяющие получить пути к системным каталогам;
- информация о пользователе (User Information Properties) – основные сведения о текущем пользователе.

Передача значений свойств в параметрах командной строки

Существует возможность указывать значения свойств, передавая их в качестве параметров при запуске msi-пакета. Это позволяет создавать сценарии, производящие автоматическую установку пакета без вмешательства администратора, сокращая необходимое для развертывания время.

Для передачи значения свойства его необходимо передать при запуске в формате [Название свойства]=[Значение свойства]. Строковые значения можно заключить в двойные кавычки.

```
PS D:\Temp> .\TestSetupProject.msi INSTALLLEVEL=5
```

Подготовив файл сценария, мы можем передать пакету значения всех необходимых свойств без взаимодействия с интерфейсом.

Теперь встроим в пакет дополнительную проверку. Допустим, мы планируем запретить непосредственный запуск msi-пакета кроме как с использованием инициирующего загрузчика (Bootstrapper), предварительно выполняющего ряд необходимых дополнительных операций. Для этого можно настроить загрузчик таким образом, чтобы он передавал пакету дополнительное свойство, например SETUP_EXE с известным значением. А при запуске пакета следует проверять наличие данного свойства, выдавая ошибку в случае его отсутствия:

```
        <Condition Message="Для начала установки запустите файл Setup.exe">Installed or
(SETUP_EXE="yes")</Condition>
```

Переменные препроцессора и переменные WiX

Windows Installer XML позволяет определять переменные препроцессора, чьи значения вычисляются в процессе работы компилятора. Также существуют переменные, инициализируемые компоновщиком.

Объявление переменной препроцессора компилятора выполняется следующим образом:

```
    <?define PropertyName="PropertyValue" ?>
```

После директивы define идет имя переменной, которому, при необходимости, присваивается значение.

Объявление данной переменной (или любой другой) может формировать какую-то сложную логику при сборке вашего пакета. Проверить факт объявления переменной можно, например, так:

```
    <?ifndef Debug ?>
      <?error Переменная Debug должна быть объявлена ?>
    <?endif ?>
```

Если добавить указанный выше текст в файл сценария, то пакет будет собран только при наличии объявленной переменной Debug. Чтобы использовать значение переменной, надо обратиться к ней с использованием конструкции $(var.PropertyName). Такая переменная может пригодиться в случаях, когда некоторые значения формируются на основе других переменных препроцессора и должны быть использованы в атрибутах элементов:

```
    <!-- Определить переменную препроцессора для ссылки на каталог SharedBinaries -->
    <?define SharedBinariesFolder="$(var.SolutionDir)..\SharedBinaries\" ?>
…
      <Component Id="DevExpressDatav8.3Dll" Guid="????????-5298-48FC-BCFC-4EB537DCEB9B" >
        <File Id="DevExpress.Data.v8.3.dll" Name="DevExpress.Data.v8.3.dll"
Source="$(var.SharedBinariesFolder)">
        </File>
      </Component>
```

В примере выше значение переменной SharedBinariesFolder формируется на основании значения переменной SolutionDir для ее последующего использования в атрибуте Source элемента File. При этом никакие другие типы переменных не применимы, так как их значения еще не известны на этапе компиляции.

Также в Windows Installer XML существуют переменные, значения которых устанавливаются компоновщиком. Для объявления такой переменной используется элемент WixVariable:

```
<WixVariable Id="WixVar" Value="WixVarValue" />
```

А для использования значения переменной внутри пакета предназначена конструкция !(wix.WixVar).

Замечание: переменные препроцессора компилятора и компоновщика также можно объявлять на странице Build свойств проекта.

Форматированные строки

Чтобы использовать значение свойства его идентификатор следует поместить в квадратные скобки. Есть возможность объединять в одной строке значения нескольких свойств. Если какое-либо из свойств не инициализировано, вместо него будет подставлена пустая строка. При выполнении содержащего приведенный ниже фрагмент пакета переменная TestProperty4 получит значение «TestValue1##», так как переменные TestPropery2 и TestProperty3 не определены.

```
    <Property Id="TestProperty1" Value="TestValue1" />
    <Property Id="TestProperty2" />
    <SetProperty Id="TestProperty4" Before="FileCost" Sequence="ui"
Value="[TestProperty1]#[TestProperty2]#[TestProperty3]" />
```

Возможности форматирования позволяют также извлечь полный путь к файлу или путь установки компонента по идентификатору:

```
    <!-- C:\Program Files\DemoApp\DemoApplication.exe -->
    <SetProperty Id="TestProperty5" After="CostFinalize" Sequence="ui"
Value="[#DemoApplication]" />
    <!-- C:\Program Files\DemoApp\ -->
    <SetProperty Id="TestProperty6" After="CostFinalize" Sequence="ui"
Value="[$DemoApplicationExe]" />
```

Для вычисления путей необходимо запланировать присваивание после операции CostFinalize.

Проверка условий

Существует множество случаев, когда поведение программы установки зависит от выполнения того или иного условия. Это может быть наличие или отсутствие на компьютере какого-либо файла, установленного продукта, записи в реестре. Взаимодействие пользователя с элементами управления также может отражаться на ходе установки.

Для проверки условного выражения предназначен элемент Condition. В зависимости от места использования, он может описывать условие запуска программы установки, устанавливать доступность наборов и компонентов, а также управлять свойствами элементов управления.

Условие истинно, если логическое выражение внутри него истинно, либо указанная переменная инициализирована значением, либо внутри него имеется отличное от нуля числовое значение:

```
    <Condition Message="Error Message 1">1</Condition>
```

```
    <Property Id="InitializedProperty" Value="SomeValue" />
    <Condition Message="Error Message 2">InitializedProperty = "SomeValue"</Condition>

    <Condition Message="Error Message 3">NOT UninitializedProperty</Condition>
```

Во всех трех приведенных выше примерах результат вычисления условия возвращает истину: в первом случае используется отличная от нуля константа, во втором случае проверяется значение инициализированной переменной, а в третьем мы имеем отрицание ложного значения, так как свойство UninitializedProperty не инициализировано.

Проверка условий при запуске

Программы, кроме самых простых, практически всегда обрастают массой зависимостей. Чтобы избежать неприятного для пользователя разочарования и не испортить первого впечатления о продукте, информировать клиента о наличии неразрешенных зависимостей следует заранее. Наличие вопросов такого рода приводит нас к необходимости проверки определенных условий при запуске. Интерес может представлять текущая версия операционной системы, объем оперативной памяти, наличие установленных продуктов и служб.

Хорошим примером является проверка версии операционной системы, но она уже неоднократно описана. Чтобы не повторяться, воспользуемся свойством PhysicalMemory и проверим, достаточно ли на нашей машине оперативной памяти. Обязательным в данном случае является указание значения атрибута Message: в случае вычисления условия как ложного это значение будет использовано в качестве сообщения об ошибке.

```
    <Condition Message="Необходимо не менее 4Гб ОЗУ">PhysicalMemory >= 4096</Condition>
```

При запуске, как правило, проверяются значения свойств, относящихся к аппаратным – Hardware Properties – и свойства операционной системы – Operating System Properties. Для проверки установленной версии .NET Framework необходимо использовать расширение WixNetFxExtension, описанное в главе 4.

Управление доступностью компонентов и наборов

Условия позволяют отключать компоненты при необходимости. В этом случае достаточно добавить условие внутрь элемента Component. Если условие истинно, компонент доступен для установки, в противном случае он устанавливаться не будет. Устанавливать значения каких-либо атрибутов в данном случае не требуется:

```
      <Component Id="MainExecutableConfig" Guid="????????-A713-44F0-B23B-D3D550DBF0F7">
        <Condition>1</Condition>
        <File Id="DemoApplicationExeConfig" Name="DemoApplication.exe.config"
Source="$(var.DemoApplication.TargetDir)" DiskId="2" KeyPath="yes" />
      </Component>
```

Если в качестве аргумента условия выступает единица, его значение всегда истинно.

Управление наборами осуществляется похожим образом. Отличие заключается в том, что в этих случаях необходимо указывать значения атрибута Level. В случае истинного условия это значение присваивается атрибуту Level родительского набора, включая его, отключая, или делая недоступным:

```
    <Property Id="FeatureStateDisabled" Value="1" />
    <Feature ...>
      <Feature Id="ExecutableSymbols" Title="Отладочные символы" Level="2" >
        <ComponentRef Id="MainExecutablePdb" />
        <Condition Level="0">FeatureStateDisabled</Condition>
      </Feature>
    </Feature>
```

В примере выше объявляется свойство FeatureStateDisabled, значение которого устанавливается равным единице. Условие вычисляется как истинное, в результате чего значение атрибута Level элемента Condition присваивается атрибуту Level родительского по отношению к условию элемента Feature. Результат – набор с идентификатором ExecutableSymbols становится недоступным.

Свойства элементов управления

Условия позволяют управлять видимостью, доступностью и фокусом ввода элементов управления. В этих случаях необходимо указывать значение атрибута Action, принимающего следующие значения:

- default – назначает элементом «по умолчанию», внешне не проявляется;
- enable – делает элемент доступным;
- disable – делает элемент недоступным;
- hide – скрывает элемент управления;
- show – отображает элемент управления.

Например, конечный диалог в случае успешной установки может содержать переключатель, позволяющий запустить установленную программу. Когда программа уже установлена, указанный переключатель отображаться не должен:

```
        <Control Id="StartProgramCheckBox" Type="CheckBox" X="100" Y="290" Text="Запустить
программу после окончания установки" Width="250" Height="22" Property="StartProgram"
CheckBoxValue="0">
          <Condition Action="hide">Installed</Condition>
        </Control>
```

Элементы управления подробно рассматриваются в главе 5, посвященной разработке пользовательского интерфейса.

Глава 4. Использование расширений

Одной из наиболее важных особенностей WiX является возможность подключения расширений, что позволяет дополнить отсутствующую в технологии Windows Installer функциональность. Более того, в комплект поставки уже входит ряд библиотек, позволяющих решать широкий круг задач. В этой главе рассматриваются некоторые из существующих расширений. Ввиду того, что некоторые поставляемые с Windows Installer XML расширения находятся в состоянии разработки и практически не документированы, здесь они не описаны.

В тех случаях, когда существующего функционала становится недостаточно, у нас есть возможность написания собственной библиотеки, способной заполнить пробел. Вопросы создания расширений на управляемом коде, на языке C#, рассматриваются в главе 6.

Встроенные расширения

Комплект поставки WiX 3.5 содержит четырнадцать расширений. Для подключения каждого из них в проект необходимо добавить ссылку на одноименную библиотеку. Также для использования описанных в них элементов для всех расширений, кроме WixUIExtension, следует прописать псевдоним для пространства имен. Например, так будут выглядеть добавляемое по умолчанию стандартное и дополнительное, с псевдонимом «util» для расширения WixUtilExtension пространства имен:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:util="http://schemas.microsoft.com/wix/UtilExtension">
```

Кратко рассмотрим назначение и основные функции библиотек, а ниже продемонстрируем некоторые случаи их применения.

- **WixComPlusExtension**. Позволяет выполнять регистрацию COM+-компонентов в процессе установки.
- **WixDifxAppExtension**. Установка драйверов устройств с использованием технологии Driver Install Frameworks for Applications (DIFxApp).
- **WixDirectXExtension** Позволяет узнать текущие возможности видеокарты.
- **WixFirewallExtension**. Используется для добавления исключений в Windows Firewall.
- **WixGamingExtension**. Позволяет регистрировать игры в обозревателе игр.
- **WixIIsExtension**. Содержит набор действий, позволяющих решать задачи по управлению веб-сайтами, веб-приложениями, традиционно решаемые через оснастку Internet Information Services.
- **WixNetFxExtension**. Предоставляет свойства, с помощью которых можно проверить информацию об установленной версии .NET Framework. Также позволяет генерировать для .NET сборок образы в машинном коде вместо выполнения JIT-компиляции при запуске.
- **WixSqlExtension**. Предназначено для работы с базами данных SQL Server. Позволяет создавать и удалять базы, а также выполнять SQL-запросы к ним.

- **WixUIExtension**. Используется для создания интерфейсов программы установки. Подробно описывается в главе 5.
- **WixUtilExtension**. Предоставляет функции, позволяющие управлять учетными записями, назначать права доступа, редактировать XML-файлы и ряд других.

Использование других расширений – LuxExtension, MsmqExtension, PSExtension и VSExtension в данной книге не рассматривается.

Замечание: при использовании элементов, описанных в библиотеках расширения, вы можете получить ошибку вида:

The localization variable !(loc.<имя_переменной>) is unknown. Please ensure the variable is defined.

Причина ошибки – в отсутствии локализованных ресурсов для выбранного основного языка. В этом случае языковые ресурсы следует подключать не на закладке Build в свойствах проекта, а на закладке Tool Settings в поле дополнительных параметров компоновщика, например:

-cultures:ru-RU;en-US

Расширение WixComPlusExtension – регистрация COM+-компонентов

До появления технологии Windows Communication Foundation для создания компонентов уровня предприятия широко применялись службы на основе механизмов COM+. На тот момент только они одновременно поддерживали транзакции, в том числе распределенные, пулы потоков и объектов, активацию по запросу и безопасность на основе ролей. Создаваемые с помощью .NET Framework COM+-компоненты носят название обслуживаемых (Serviced Components) и способны использовать все преимущества данной технологии.

Чтобы создать простейший COM+-компонент с помощью Visual Studio, достаточно унаследовать новый открытый класс от типа System.EnterpriseServices.ServicedComponent и добавить в него открытый метод:

```
using System;

using System.EnterpriseServices;

namespace ComPlusComponent

{

  public class ServicedCom: ServicedComponent

  {

    public String GetCurrentTimeString() { return DateTime.Now.ToLongTimeString(); }

  }

}
```

Кроме того, сборку потребуется подписать строгим именем и установить для нее в true атрибут ComVisible. Данный COM+-компонент, не выполняя полезных функций, позволяет продемонстрировать процесс регистрации. В вызывающий проект необходимо добавить ссылку

на созданную библиотеку, после чего можно создать объект данного типа и использовать его методы:

```
private void buttonInvoke_Click(object sender, EventArgs e)
{
   ServicedCom servicedCom = new ServicedCom();
   String result = servicedCom.GetCurrentTimeString();
   MessageBox.Show(result);
}
```

При первом обращении компонент будет зарегистрирован в системе, что можно проверить с помощью соответствующей оснастки. Однако для выполнения регистрации необходимо запустить приложение от имени административной учетной записи, что недопустимо для большинства пользовательских приложений по соображениям безопасности.

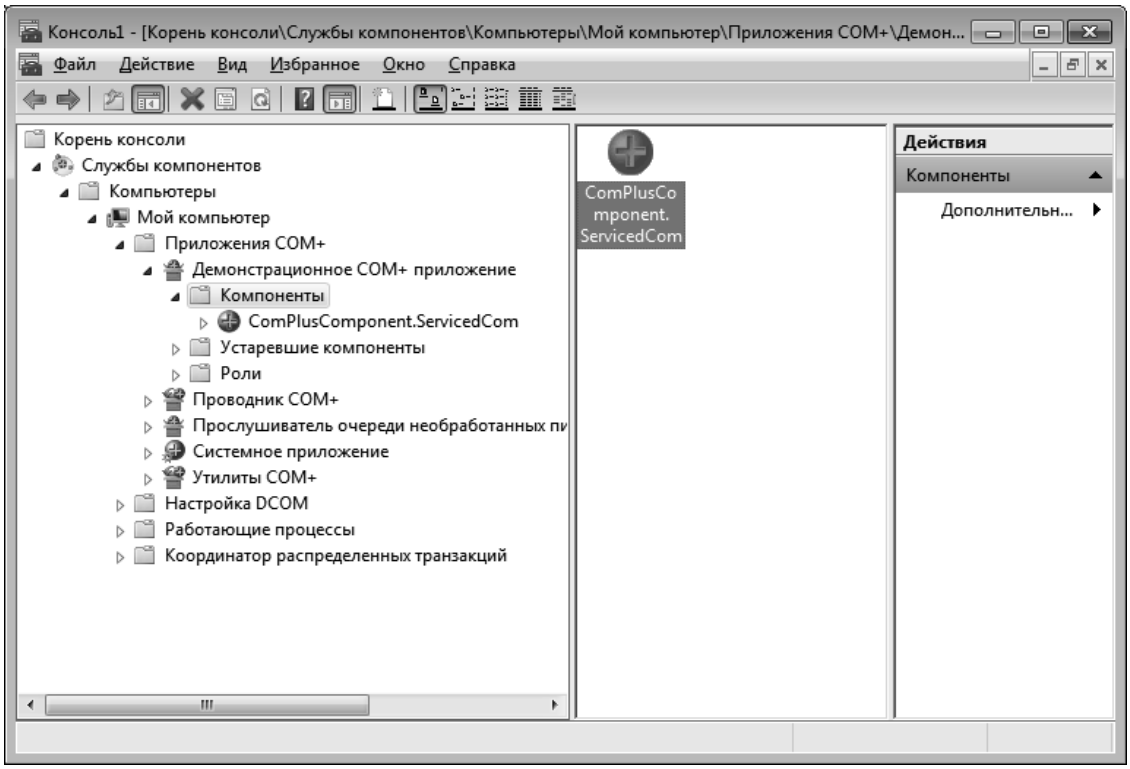

Рисунок 4.1 Зарегистрированный в системе COM+-компонент.

Чтобы не предоставлять пользователям административных полномочий, регистрацию компонентов следует выполнять при установке приложения, а отмену регистрации – при деинсталляции. Для решения этой задачи и предназначено расширение WixComPlusExtension.

Для использования расширения необходимо добавить в проект ссылку на данную библиотеку, а затем описать пространство имен «http://schemas.microsoft.com/wix/ComPlusExtension»:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:complus="http://schemas.microsoft.com/wix/ComPlusExtension">
```

Расширение предоставляет множество элементов, позволяющих конфигурировать различные аспекты поведения, но для регистрации отдельного компонента важнейшими являются ComPlusApplication, ComPlusAssembly и ComPlusComponent.

Можно выделить три случая: регистрация компонента из неуправляемой библиотеки, а также регистрация управляемой COM+-сборки с размещением библиотеки в GAC или без нее.

Замечание: при работе с данным расширением мне так и не удалось установить компонент, содержащийся в .NET-сборке при ее одновременной регистрации в GAC. Данный вопрос на сегодняшний день остается открытым.

Для регистрации компонента нам потребуется знать присваиваемый ему идентификатор класса. Самый простой способ получить его для управляемой сборки – зарегистрировать компонент с использованием утилиты regsvcs:

```
c:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\regsvcs.exe ComPlusComponent.dll
```

В примере выше регистрируется компонент, содержащийся в сборке ComPlusComponent.dll. Теперь запустим оснастку управления службами компонентов, приведенную на рисунке 4.1, и откроем свойства компонента.

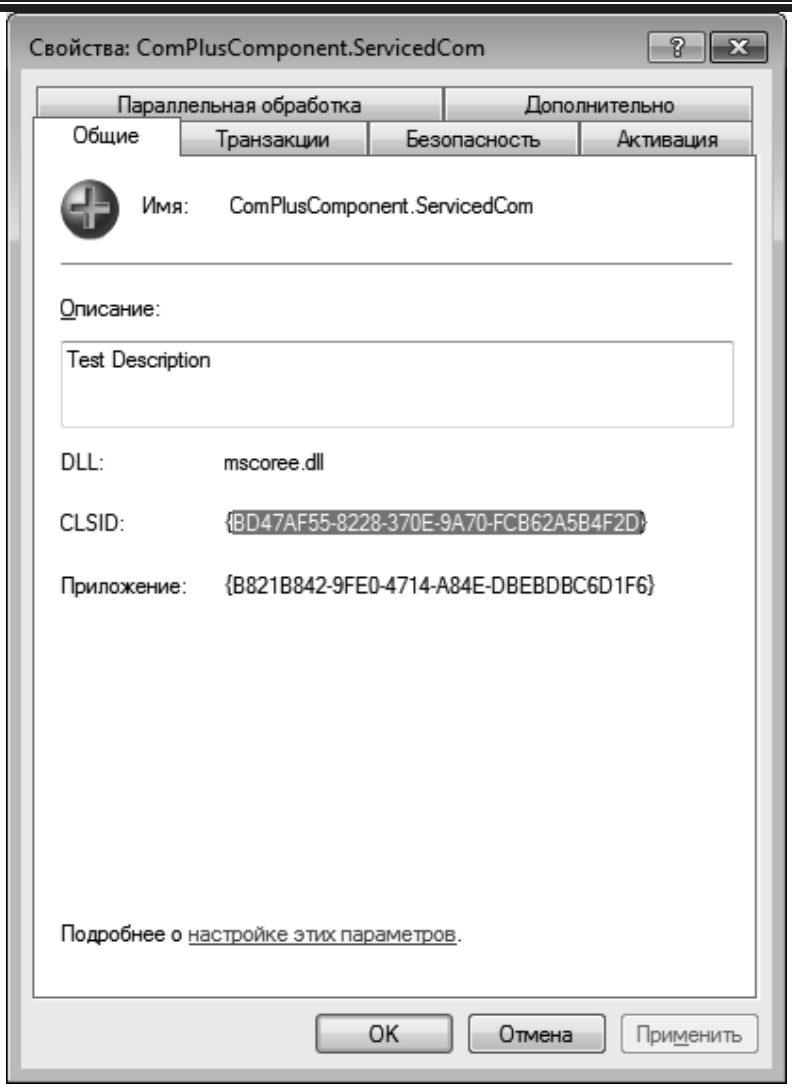

Рисунок 4.2 Свойства установленного COM+-компонента.

Значение поля CLSID на рисунке 4.2 и является интересующим нас значением, которое необходимо указать в атрибуте CLSID элемента ComPlusComponent. Для установки управляемого компонента нам потребуются два файла – собственно библиотека и файл библиотеки типов, создаваемые при сборке проекта. Файлы целесообразно разнести по разным компонентам. Регистрирующие элементы размещаются в том же компоненте, где и файл библиотеки. Первым идет элемент ComPlusApplication, описывающий приложение COM+. Пожалуй, наиболее важным атрибутом является Activation. Принимая значения inproc или local, он устанавливает тип активизации компонента – как серверного или библиотечного приложения соответственно. Остальные атрибуты практически один к одному дублируют элементы управления, присутствующие на закладках приложения в оснастке.

Дочерним для ComPlusApplication является элемент ComPlusAssembly. Он используется для указания имени и свойств библиотеки, содержащей компонент. Кроме идентификатора важными являются атрибуты Type, DllPath и, только для .NET-сборок, TlbPath. Type задает тип сборки – .net или native. Если тип установлен в .net, следует указать значения атрибутов DllPath и TlbPath. В первый помещается путь к библиотеке, а во второй – путь к TLB-файлу. Для передачи пути следует использовать форматированную строку вида [#FileId], где FileId является идентификатором

соответствующего элемента File. Для тех случаев, когда Type установлен в native, значение атрибута TlbPath не используется и не указывается.

И, наконец, внутрь ComPlusAssembly помещается элемент ComPlusComponent, задающий свойства компонента. Важнейшим здесь является правильное указание значения атрибута CLSID, получение которого было описано ранее, в противном случае установка не будет выполнена. Атрибуты данного элемента позволяют установить различные аспекты поведения добавленного компонента: безопасность, поддержку транзакций, синхронизацию и другие.

Простейший пример регистрации COM+-компонента из управляемой сборки приведен ниже:

```
    <DirectoryRef Id="INSTALLLOCATION">
      <!-- Библиотека -->
      <Component Id="ServicedCom" Guid="????????-BCA7-41EB-A9B7-F2142436FAA5">
        <File Id="ComPlusComponentDll" Name="ComPlusComponent.dll"
Source="$(var.ComPlusComponent.TargetDir)" KeyPath="yes" Assembly="no" />
        <complus:ComPlusApplication Id="DemoComPlusApplication" Name="Demo COM+">
          <complus:ComPlusAssembly Id="DemoComPlusComponentAssembly"
DllPath="[#ComPlusComponentDll]" TlbPath="[#ComPlusComponentTlb]" Type=".net" >
            <!-- Значение атрибута CLSID взято из надстройки, рисунок 4.2-->
            <complus:ComPlusComponent Id="DemoComPlusComponent" CLSID="BD47AF55-8228-370E-
9A70-FCB62A5B4F2D" Description="Test Description" />
          </complus:ComPlusAssembly>
        </complus:ComPlusApplication>
      </Component>
      <!-- TLB-файл -->
      <Component Id="ComPlusComponentTlb" Guid="????????-58EC-4FAE-8AA3-E9D92A75B2BA">
        <File Id="ComPlusComponentTlb" Name="ComPlusComponent.tlb"
Source="$(var.ComPlusComponent.TargetDir)" />
      </Component>
    </DirectoryRef>
```

При регистрации компонента из неуправляемой сборки несколько отличаются значения атрибутов для элемента ComPlusAssembly. Атрибут Type устанавливается в native. Кроме того, TLB-файл для такой сборки отсутствует, поэтому атрибут TlbPath также не используется.

```
    <!-- Native COM+ -->
    <DirectoryRef Id="INSTALLLOCATION">
      <Component Id="NativeCom" Guid="????????-DC6A-4D8A-8960-87198B7FACBB">
        <File Id="NativeComDll" Name="NativeCom.dll" Source="$(var.NativeCom.TargetDir)"
KeyPath="yes" />
        <complus:ComPlusApplication Id="NativeComApplication" Name="Demo Native COM+">
          <complus:ComPlusAssembly Id="NativeComComponentAssembly"
DllPath="[#ComPlusComponentDll]" Type="native" >
            <complus:ComPlusComponent Id="NativeComComponent" CLSID="6A671598-F914-47F3-B80A-
B4AA11C95D6F" Description="Test Description" />
          </complus:ComPlusAssembly>
        </complus:ComPlusApplication>
```

```
      </Component>
    </DirectoryRef>
```

Последнее, о чем следует сказать, это поддержка русского языка. По умолчанию данное расширение содержит файлы для английского, испанского и, почему-то, японского языков. Добавление файла локализации в проект описано в главе 5, а насчитывающий несколько десятков значений список подлежащих переводу строк может быть найден в исходных кодах Windows Installer XML.

```
  <String Id="RegisterComPlusAssemblies" Overridable="yes">Регистрация компонентов COM+
</String>
```

Таким образом, с некоторыми ограничениями, данное расширение позволяет регистрировать и настраивать COM+-компоненты.

Расширение WixDifxAppExtension – установка драйверов устройств

Расширение WixDifxAppExtension добавляет в Windows Installer XML поддержку DIFxApp - Driver Install Frameworks for Applications – решения, позволяющего устанавливать драйверы с использованием msi. Это позволяет производить установку драйверов устройств при работе программы установки.

Чтобы воспользоваться возможностями данного расширения, нам потребуется INF-файл, который должен иметь каждый драйвер. Это текстовый файл, известной структуры, в котором содержатся сведения о названии и местонахождении файла драйвера, версии драйвера и записях реестра. Нам также потребуется подписанный файл каталога (с расширением CAT) и непосредственно файл драйвера (SYS). Описание создания указанных файлов выходит за рамки данной книги.

Для использования в проект необходимо добавить ссылку на расширение DifxAppExtension. В корневом элементе следует описать псевдоним – в нашем случае difx – для пространства имен "http://schemas.microsoft.com/wix/DifxAppExtension". Также потребуется добавить в проект ссылку на файл difxapp_*.wixlib для поддерживаемой аппаратной платформы: difxapp_x86.wixlib, difxapp_x64.wixlib или difxapp_ia64.wixlib – x86, x64 или ia64 соответственно.

При установке драйвера лучше отказаться от создания отдельного компонента на каждый файл и рассматривать все составляющие как единое целое.

```
      <Component Id="DirFilterDriver" Guid="????????-A60E-4411-824B-583F3252E08B">
        <File Id="DirFilterSYS" Name="DirFilter.sys" DiskId="1" Source="DirFilter.sys"
KeyPath="yes" />
        <File Id="DirFilterINF" Name="DirFilter.inf" DiskId="1" Source="DirFilter.inf" />
        <File Id="DirFilterCAT" Name="DirFilter.cat" DiskId="1" Source="DirFilter.cat" />
        <difx:Driver Legacy="yes" />
      </Component>
```

Элемент Driver инициирует работу по установке драйвера в систему. При отсутствии подписанного CAT-файла установку можно произвести, установив в yes значение атрибута Legacy.

Дополнительно можно указать ряд опций, используя дополнительные атрибуты:

- AddRemovePrograms - создавать или нет запись в разделе Установка и удаление программ (в Windows Vista и старше - Программы и компоненты) - по умолчанию yes;
- DeleteFiles - следует ли при деинсталляции удалить двоичные файлы, копирующиеся в процессе установки драйвера - по умолчанию no;
- ForceInstall - значение yes принудительно устанавливает текущий драйвер даже при наличии в системе более нового;
- Legacy - значение yes позволяет устанавливать неподписанные драйверы и драйверы с отсутствующими файлами - по умолчанию no;
- PlugAndPlayPrompt - выдавать или нет запрос на подключение устройства Plug-and-Play в случае, если оно не подключено - по умолчанию yes;
- Sequence - целочисленное значение, позволяющее установить порядок в случае установки более чем одного драйвера, иначе порядок установки не определен.

Расширение WixFirewallExtension – настройка сетевого экрана

Расширение WixFirewallExtension позволяет настраивать брандмауэр, который поставляется вместе с операционными системами Windows XP и старше.

Функционал брандмауэра расширяется от версии к версии; так, поддержка блокировки исходящих соединений появилась только вместе с Windows Vista и Windows Server 2008. По умолчанию все входящие подключения блокируются, если для них не назначены разрешающие правила. Для работы программы может быть необходим как прием соединений извне, так и подключение к внешним ресурсам, для чего необходимо добавить соответствующее правило в список исключений брандмауэра.

Исключения могут назначаться как для конкретной программы, так и для порта. Исключение для программы позволяет ей выполнять соединения с использованием любых портов и протоколов. Исключения, назначаемые для порта, позволяют открыть порт с известным номером для протокола TCP или UDP. Оба типа исключений позволяют накладывать ограничения на местоположение вызывающей стороны:

- любые сети, в том числе Интернет;
- только локальная подсеть;
- конкретный IP-адрес.

Для конфигурирования брандмауэра с помощью WixFirewallExtension необходимо добавить в проект ссылку на данную библиотеку, а затем подключить соответствующее пространство имен «http://schemas.microsoft.com/wix/FirewallExtension»:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:fire="http://schemas.microsoft.com/wix/FirewallExtension">
```

Наиболее простой случай - создание исключения для программы. При этом достаточно добавить элемент FirewallException в качестве дочернего для исполняемого файла, указав значения атрибутов Id и Name. Значение атрибута Name используется как название правила и отображается в панели управления брандмауэра. Если элемент не является дочерним для File, следует

использовать атрибут File, где указывается идентификатор исполняемого файла. При создании исключения для программы неприменимы атрибуты Program, Port и Protocol.

```
    <Component Id="DemoApplicationExe" Guid="????????-789A-4001-A66C-9491BAAF529C">
      <File Id="DemoApplicationExe" Name="DemoApplication.exe"
Source="$(var.DemoApplication.TargetDir)" DiskId="2" KeyPath="yes" >
        <fire:FirewallException Id="FirewallException1" Name="Demo Application Exception"
Scope="any" />
      </File>
    </Component>
```

Если указать для элемента FirewallException атрибут Scope со значением localSubnet, то можно принимать подключения только из локальной подсети. Значение any не накладывает ограничений. Обязательным является либо задание значения атрибута Scope, либо вложение дочерних элементов RemoteAddress.

В случае, когда допускается прием подключений только от узлов с известными IP-адресами, их можно указать с помощью дочерних для FirewallException элементов RemoteAddress. В Windows Vista адрес может быть в формате IPv6, а также задаваться диапазоном значений.

```
      <Component Id="DemoApplicationExe2" Guid="????????-DB94-44d9-AF3E-4E2C7ADB25E0">
        <File Id="DemoApplicationExe" Name="DemoApplication.exe"
Source="$(var.DemoApplication.TargetDir)" DiskId="2" KeyPath="yes" >
          <fire:FirewallException Id="FirewallException2" Name="Demo Application Exception 2">
            <fire:RemoteAddress>127.0.0.1</fire:RemoteAddress>
            <fire:RemoteAddress>127.0.0.2</fire:RemoteAddress>
            <!-- Для Windows Vista и старше -->
            <fire:RemoteAddress>127.0.0.6-127.0.0.10</fire:RemoteAddress>
          </fire:FirewallException>
        </File>
      </Component>
```

Кроме того, для элемента FirewallException может быть указан атрибут Program, содержащий имя программы, для которой создается исключение. Это может быть полезным в случаях, когда исключение создается для продукта, развернутого другой программой установки. В этом случае также не используются атрибуты File, Port и Protocol.

Создание исключения для порта выполняется несколько иначе. В этом случае задаются значения атрибутов Port и Protocol, где первый хранит номер порта, а второй принимает значения tcp, udp или any, устанавливая допустимый протокол. Отмечу, что вариант Protocol="any" допустим только в Windows Vista и старше. Для Windows XP и Windows Server 2003 необходимо создавать два правила – по одному для каждого протокола.

```
          <Component Id="PortException" Guid="????????-B893-4B36-B91A-874D9F4684ED">
            <fire:FirewallException Id="FirewallException3" Name="TCP:9997" Port="9997"
Protocol="tcp" Scope="any" />
          </Component>
```

Дополнительно для элемента FirewallException можно установить в yes или no атрибут IgnoreFailure. Значение yes позволяет игнорировать любые ошибки в процессе создания правила; значение no, применяемое по умолчанию, в этом случае инициирует откат установки.

Расширение WixDirectXExtension – проверка возможностей видеокарты

Данное расширение позволяет узнать возможности видеокарты на клиентской машине. Наличие данной информации важно при установке игр и других программ, использующих возможности, предоставляемые библиотеками DirectX.

Добавив ссылку на расширение, мы получаем возможность подключить два свойства: WIX_DIRECTX_PIXELSHADERVERSION и WIX_DIRECTX_VERTEXSHADERVERSION. Первое свойство возвращает версию пиксельных шейдеров, поддерживаемых видеокартой, второе – вершинных. Каждое свойство в процессе вычисления получает значение, рассчитываемое как [старший номер версии] * 100 + [младший номер версии]. Таким образом, для версии Shader Model 3.0 свойство WIX_DIRECTX_PIXELSHADERVERSION получит значение 300.

Для использования свойств их необходимо подключить с помощью элемента PropertyRef. Значения свойств вычисляются не при запуске пакета, при вызове входящей в это же расширение операции WixQueryDirectXCaps, поэтому проверку версий шейдеров следует выполнять только после ее завершения. Для задания порядка выполнения изменяются последовательности InstallExecuteSequence и InstallUISequence, описываемые в главе 6. В случае, когда номер версии шейдеров ниже требуемого, вызывается расширенная операция – в примере ниже ей присвоен идентификатор CA_CheckPixelShaderVersion, выводящая сообщение об ошибке и завершающая процесс установки.

```
<PropertyRef Id="WIX_DIRECTX_PIXELSHADERVERSION" />

<CustomAction Id="CA_CheckPixelShaderVersion" Error="[ProductName] требует поддержки
видеокартой пиксельных шейдеров версии 3.0 или старше." />

<InstallExecuteSequence>
  <Custom Action="CA_CheckPixelShaderVersion" After="WixQueryDirectXCaps">
    <![CDATA[WIX_DIRECTX_PIXELSHADERVERSION < 300]]>
  </Custom>
</InstallExecuteSequence>

<InstallUISequence>
  <Custom Action="CA_CheckPixelShaderVersion" After="WixQueryDirectXCaps">
    <![CDATA[WIX_DIRECTX_PIXELSHADERVERSION < 300]]>
  </Custom>
</InstallUISequence>
```

В отличие от других свойств, указанные два по умолчанию инициализируются значениями NotSet. Если запрос параметров завершится неудачей, никакие ошибки генерироваться не будут, а свойства сохранят данные значения.

Расширение WixGamingExtension – регистрация игр

В Windows Vista впервые появился обозреватель игр – Game Explorer – специальный каталог, упрощающий доступ к играм и управление ими. В Windows 7 обозреватель был несколько доработан, а его поддержка в том числе необходима для получения логотипа Games for Windows. На сегодняшний день обозреватель предоставляет следующие возможности:

- единая точка доступа к играм;
- настраиваемые контекстные команды для каждой игры;
- отображение информации об игре от ее поставщика (заголовок, описание, требования к производительности системы, возрастной ценз и др.);
- периодическое получение новостей об игре;
- информация о наличии обновлений и возможность их установки;
- отображение статистики и уведомлений.

Рисунок 4.3 Зарегистрированная в обозревателе игр программа.

Реализация конкретных возможностей из числа перечисленных выше зависит от ее производителя, но регистрацию игры в обозревателе следует выполнять в программе установки. Для этого и предназначено расширение WixGamingExtension. На рисунке 4.3 приведен пример программы, зарегистрированной с помощью этого расширения.

Подготовка программы к регистрации в обозревателе игр

Для успешной регистрации игры в исполняемый файл необходимо встроить так называемый Game Definition File (GDF), создание которого выполняется с помощью графической утилиты GDFMaker.exe, входящей в комплект поставки DirectX SDK. Результатом использования утилиты являются файлы <имя_проекта>.rc и <имя_проекта>.h, а также подкаталоги по числу поддерживаемых языковых культур с файлами изображений и GDF-файлами <имя_проекта>.gdf.xml.

Полученные файлы используются для создания Win32-ресурсов (<имя_проекта>.res) и их последующего добавления в решение.

Замечание: используемые по умолчанию в .NET-проектах ресурсные файлы *.resx не поддерживаются. Для встраивания в .NET-сборку res-файла следует в свойствах проекта на закладке Application установить переключатель Resources в состояние Resource File и указать используемый файл.

Проверить корректность встроенного в приложение GDF-файла позволяет утилита GDFTrace.exe, также поставляемая с DirectX SDK.

Регистрация игры

Чтобы использовать элементы расширения, следует добавить в корневой элемент псевдоним для пространства имен «http://schemas.microsoft.com/wix/GamingExtension»:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:game="http://schemas.microsoft.com/wix/GamingExtension">
```

Для регистрации игры добавим дочерний элемент Game для исполняемого файла, установив уникальный GUID в качестве значения атрибута Id.

```
<Component Id="Executable" Guid="????????-A2E1-44E6-BB2D-F26182B13EA0">
  <File Id="GameDemoExe" Name="GameDemo.exe" Source="$(var.GameDemo.TargetPath)" >
    <game:Game Id="????????-5017-4173-AD38-8C3FC499A8B8" />
  </File>
</Component>
```

Элемент Game можно располагать отдельно, но тогда для него потребуется дополнительно указать идентификатор исполняемого файла в атрибуте ExecutableFile.

Замечание: в элементе Game присутствует атрибут GdfResourceFile для тех случаев, когда Game Definition File не является ресурсом исполняемого файла. В текущей версии расширения данная возможность не поддерживается, попытка зарегистрировать игру без встроенного GDF-файла завершится неудачей.

Создание задач для Windows Vista

Задачи в обозревателе игры выглядят как команды в контекстном меню, а фактически представляют ярлыки, сконфигурированные для выполнения различных операций. Задачи разделяются на игровые (Play tasks) и сопровождающие (Support tasks). Первые представляют собой ярлык для исполняемого файла с возможность указания аргументов командной строки;

вторые – ссылки на внешние ресурсы. При подготовке дистрибутива для Windows 7 нет необходимости явно регистрировать задачи – теперь они описываются в Game Definition File с использованием элемента ExtendedProperties, поэтому описанные ниже операции имеют смысл только для регистрации задач в Windows Vista.

```
              <game:Game Id="????????-5017-4173-AD38-8C3FC499A8B8" >
                <game:PlayTask Name="Запустить" />
                <game:PlayTask Name="Запустить в безопасном режиме" Arguments="-safeMode" />
                <game:SupportTask Name="Перейти на страницу игры"
Address="http://myaddress.ru" />
              </game:Game>
```

Поддержка сохраненных игр

В Windows Vista присутствует возможность предварительного просмотра файлов, содержащих информацию о сохраненных играх. Основная часть функций реализуется в самой игре, в программе установки необходимо только добавить атрибут IsRichSavedGame в элемент Extension при регистрации обработчика для расширения. В Windows 7 данная возможность более не поддерживается.

Регистрация обработчика рассматривается в главе 3 в разделе «регистрация расширений файлов».

Таким образом, регистрация программы в обозревателе игр является задачей, которая может эффективно решаться средствами Windows Installer XML, в то время как основная часть работы возлагается на разработчика игры.

Расширение WixIISExtension – установка веб-приложений

Задача создания программ установки для развертывания веб-приложений возникает относительно редко. Это связано с тем, что, в отличие от настольных приложений, они чаще существуют в количестве нескольких экземпляров, централизованно управляются и обновляются. В таких условиях необходимость подготовки msi-пакета не всегда оправдана. Тем не менее, Windows Installer XML предоставляет в распоряжение разработчика расширение, позволяющее развертывать веб-приложения и веб-службы, работающие под управлением Internet Information Services (IIS).

Замечание: предоставляемая Internet Information Services функциональность изменяется между версиями. Работоспособность приведенных ниже примеров проверена в Windows 7 для IIS 7.5.

Чтобы использовать расширение, добавим в проект ссылку на него и подключим необходимое пространство имен:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:iis="http://schemas.microsoft.com/wix/IIsExtension">
```

Начнем рассмотрение с описания создания сайта. Веб-сайт, описываемый элементом WebSite, является корневым узлом для веб-приложений, внутри него размещаются виртуальные каталоги. Расширением поддерживается как создание нового веб-сайта, так и использование существующего – при размещении элемента внутри компонента сайт будет создан в процессе установки и удален при деинсталляции. Если же узел WebSite помещается непосредственно

внутрь тега Product, будет использован существующий сайт с указанным именем – он не будет создаваться и удаляться установщиком. В последнем случае отсутствие сайта при установке прервет ее с сообщением об ошибке. Обязательным для объекта WebSite является наличие хотя бы одного дочернего элемента WebAddress, определяющего прослушиваемые порты и используемые IP-адреса.

```
<!-- Ссылка на существующий сайт -->
<iis:WebSite Id="Default_Web_Site" Description="Default Web Site">
  <!-- Протокол http:80, IP-адрес любой -->
  <iis:WebAddress Id="_80" Port="80" IP="*" />
</iis:WebSite>
```

Для элемента WebSite необходимо указать значения атрибутов Id и Description – первый используется внутри пакета, второй – идентифицирует узел внутри IIS. Для WebAddress обязательно указывается идентификатор и номер порта в атрибуте Port. Атрибут IP дополнительно позволяет указать IP-адрес, где символ «*» при создании соответствует значению «Все неназначенные» (All Unassigned), а при поиске – любому адресу вообще, включая все неназначенные. Установка в yes атрибута Secure добавляет требование использования протокола HTTPS:

```
<iis:WebAddress Id="_443" Port="443" Secure="yes" IP="*" />
```

Кроме описанных выше, WebSite позволяет указать ряд дополнительных атрибутов:

- ConfigureIfExists – значение yes указывает на необходимость конфигурирования уже существующего сайта при его наличии;
- ConnectionTimeout – устанавливает таймаут в секундах для подключений;
- StartOnInstall – установка в yes запускает сайт при установке;
- WebLog – ссылается на идентификатор элемента WebLog, устанавливая формат активного журнала событий.

Далее создается виртуальный каталог для размещения содержимого, для чего используется элемент WebVirtualDir. Виртуальный каталог может находиться в любом месте файловой системы и является корневым узлом создаваемых веб-приложений. Каждый из данных элементов может быть вложен внутрь WebSite, в противном случае в атрибут WebSite записывается идентификатор описанного ранее веб-сайта. Обязательно указывается идентификатор – атрибут Id. Для виртуального каталога задается значение атрибутов Directory и Alias – ссылка на каталог с содержимым и используемый в URL псевдоним. Обязательным для каталога является ссылка на элемент WebDirProperties через одноименный атрибут или размещение в виде дочернего узла. Данный элемент используется для задания множества свойств: аутентификации и прав доступа, шифрования и устаревания содержимого – фактически, большинством параметров, находящихся на странице свойств.

В настроенном каталоге при необходимости можно создать веб-приложение, для чего используется элемент WebApplication.

Рассмотрим простой пример. Сначала создадим структуру физических каталогов для размещения файлов библиотек:

```
<Directory Id="TARGETDIR" Name="SourceDir">
```

```
      <Directory Id="ProgramFilesFolder">
        <Directory Id="WebApplicationRoot" Name="WebApplicationRoot">
          <!-- Каталог для веб-приложения -->
          <Directory Id="TestWebApplication" Name="TestWebApplication">
            <Directory Id="BinariesFolder" Name="bin" />
          </Directory>
        </Directory>
      </Directory>
    </Directory>
```

Теперь скопируем в созданные каталоги необходимые для работы приложения файлы: web.config, Default.aspx и bin\TestWebApplication.dll. Описание создания используемых в примере файлов выходит за рамки данной книги.

```
    <!-- Файлы в рабочем каталоге и настройка веб-приложения -->
    <DirectoryRef Id="TestWebApplication">
      <!-- Конфигурирование веб-приложения -->
      <Component Id="DefaultAspx" Guid="????????-9463-4107-B1F1-A5F0DB1B1638">
        <File Id="DefaultAspx" Name="Default.aspx"
Source="$(var.TestWebApplication.TargetDir)..\" />
      </Component>
      <Component Id="WebConfig" Guid="????????-7E0C-4E47-99FA-C988BA2D5215">
        <File Id="WebCfg" Name="Web.config" Source="$(var.TestWebApplication.TargetDir)..\" />
      </Component>
    </DirectoryRef>

    <!-- Файлы библиотек в подкаталоге bin -->
    <DirectoryRef Id="BinariesFolder">
      <Component Id="TestWebApplicationDll" Guid="????????-867E-47F4-895E-63FFAFAE97D4">
        <File Id="TestWebApplicationDll" Name="TestWebApplication.dll"
Source="$(var.TestWebApplication.TargetDir)" />
      </Component>
    </DirectoryRef>
```

И, наконец, добавим компонент, в котором создается виртуальный каталог, устанавливаются свойства каталога, а затем конфигурируется веб-приложение.

```
      <!-- Конфигурирование веб-приложения -->
      <Component Id="WebAppConfiguration" Guid="????????-0036-4748-8CA7-83A2B706FD7E}">
        <CreateFolder Directory="TestWebApplication" />
        <iis:WebVirtualDir Id="RootVirtualDir" Alias="webAdmin" Directory="TestWebApplication"
WebSite="Default_Web_Site" >
          <iis:WebDirProperties Id="WebDefaultDir" DefaultDocuments="Default.aspx"
AnonymousAccess="yes" Read="yes" Write="no" Script="yes" Index="no"  />
          <iis:WebApplication Id="WebApplication" AllowSessions="yes" ClientDebugging="no"
DefaultScript="VBScript" Name="MyApp" ServerDebugging="no" />
        </iis:WebVirtualDir>
```

```
    </Component>
```

Созданное программой установки веб-приложение приведено на рисунке 4.4.

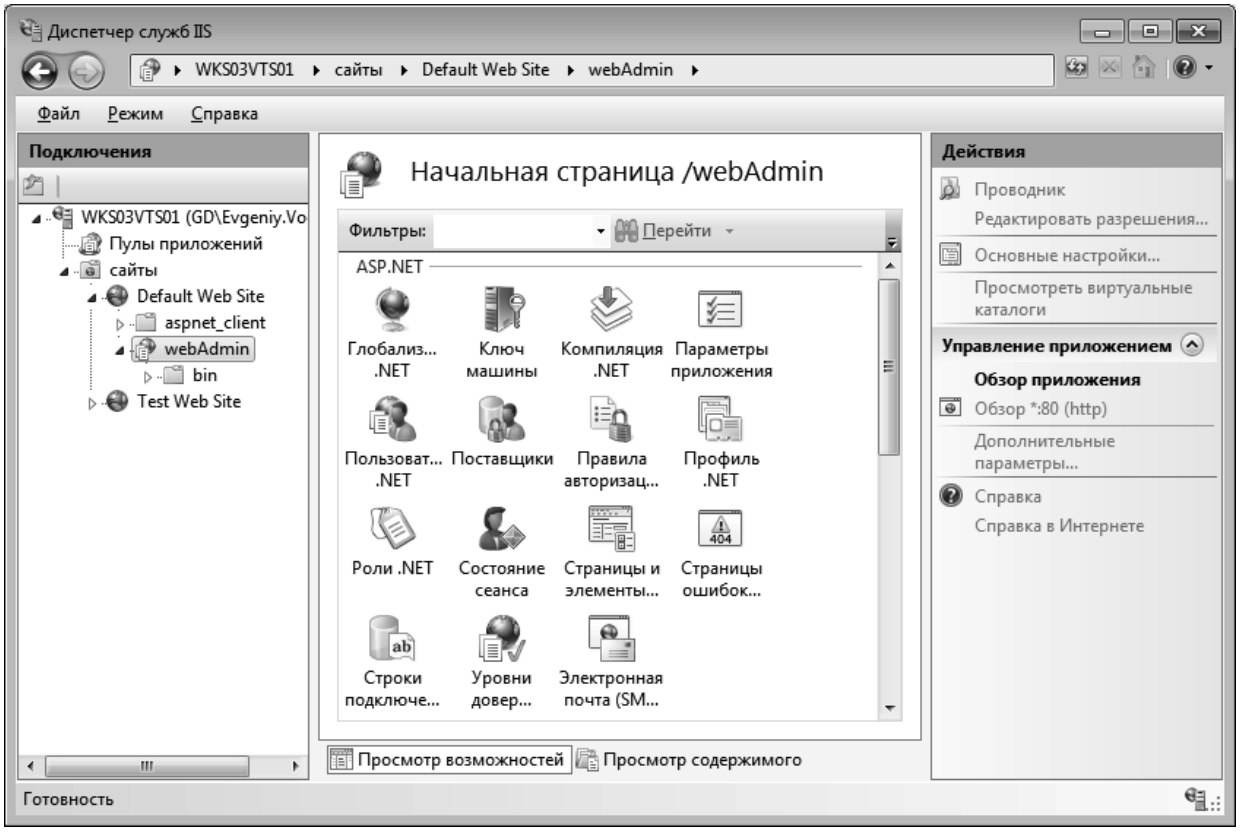

Рисунок 4.4 Зарегистрированное программой установки веб-приложение.

Создание пула приложений в IIS 6

При развертывании веб-приложений под IIS версии 6 есть возможность описания собственного пула приложений, для чего используются элементы WebAppPool и RecycleTime.

```
          <Component Id="CustomPoolComponent" Guid="????????-C975-439F-9FD9-D33BCDA2576A">
            <iis:WebAppPool Id="CustomPool" Name="Custom Pool" CpuAction="none"
MaxCpuUsage="90" RefreshCpu="5" Identity="localService" IdleTimeout="3"
MaxWorkerProcesses="10" PrivateMemory="1000000" QueueLimit="10" RecycleMinutes="3"
RecycleRequests="20" VirtualMemory="2000000" >
              <iis:RecycleTime Value="03:30" />
              <iis:RecycleTime Value="05:55" />
            </iis:WebAppPool>
          </Component>
```

Опираясь на описанное выше, расширение WixIIsExtension можно использовать для развертывания веб-приложений и служб, функционирующих под управлением Internet Information Services.

Расширение WixUtilExtension – полезные возможности

Данное расширение предоставляет множество полезных функций, отсутствующих в стандартном пространстве имен. Не решая какой-либо единственной задачи, оно может пригодиться в различных ситуациях, среди которых:

- получение дополнительной информации о параметрах операционной системы;
- управление учетными записями пользователей;
- создание общих каталогов;
- обработка содержимого XML файлов;
- назначение разрешений для доступа к объектам;
- регистрация счетчиков производительности;
- создание ссылок на Web-ресурсы;
- расширенное конфигурирование служб Windows – описывается в главе 7 в посвященном службам разделе.

Для подключения необходимо добавить ссылку на библиотеку WixUtilExtension, а в заголовке документа - прописать пространство имен «http://schemas.microsoft.com/wix/UtilExtension»:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:util="http://schemas.microsoft.com/wix/UtilExtension">
```

Получение дополнительной информации об операционной системе

Расширение содержит свыше пятидесяти свойств, позволяющих получить различную информацию об операционной системе. Они дополняют уже имеющиеся в Windows Installer стандартные свойства и разнесены по четырем категориям:

- WixQueryOsInfo – информация о редакции операционной системы, например WIX_SUITE_STARTER – редакция Starter Edition, WIX_SUITE_SERVERR2 – Windows 2003 Server R2. Если проверяемое переменной условие истинно, в качестве значения ей будет присвоена единица, в противном случае переменная останется неинициализированной;
- WixQueryOsDirs – содержит наибольшее число свойств, позволяющих получить пути к различным системным каталогам. Так, WIX_DIR_INTERNET_CACHE возвращает путь к каталогу кэша браузера;
- WixQueryOsWellKnownSID – названия наиболее часто используемых участников безопасности. Переменная WIX_ACCOUNT_ADMINISTRATORS получает значение «BUILTIN\Administrators»;
- WixQueryOsDriverInfo – используются в Windows Vista и старше, возвращая дополнительную информацию о характеристиках установленных драйверов. Свойство WIX_WDDM_DRIVER_PRESENT устанавливается в 1, если драйвер видеокарты реализует Windows Display Driver Model, номер версии не возвращается. Свойство WIX_DWM_COMPOSITION_ENABLED устанавливается в 1 в случае, если используется оконный менеджер DWM.

Полный список доступных переменных приведен в справочной системе по Windows Installer XML, найти его можно по ключевому слову «OSInfo». Для использования переменной ее достаточно подключить с помощью элемента PropertyRef:

```
    <PropertyRef Id="WIX_ACCOUNT_ADMINISTRATORS" />
```

Управление учетными данными пользователей

Для управления учетными записями пользователей предназначен элемент User. С его помощью можно создать новую или добавить ссылку на существующую учетную запись. Также возможно использование некоторых зарезервированных системных идентификаторов: Everyone, Administrator, Guest, AuthenticatedUser, LocalSystem, NetworkService. Чтобы подключить учетную запись «Все», достаточно указать ее имя в атрибуте Name и идентификатор для ссылок:

```
    <util:User Id="EveryoneUser" Name="Everyone" />
```

Существующие учетные записи подключаются аналогичным образом. Так, в примере ниже мы получаем ссылку на существующего пользователя Evgeniy.Vodnev:

```
    <util:User Id="EV" Name="Evgeniy.Vodnev" />
```

При работе с базой данных SQL Server для подключения могут потребоваться реквизиты пользователя sa, при этом создавать его нет необходимости:

```
            <util:User Id="user_sa" Name="sa" CreateUser="no" FailIfExists="no"
Password="abc*$123456" />
```

При создании пользователя применимо большее число атрибутов, при этом размещаться он должен внутри компонента:

```
          <Component Id="CreateUserComponent" Guid="????????-5DDB-4D1A-B88D-AD83EDD0A918">
            <CreateFolder Directory="INSTALLLOCATION" />
            <util:User Id="NewUser" CreateUser="yes" Name="NewUserName" FailIfExists="no"
UpdateIfExists="yes" Password="Pa$$w0rd1234" CanNotChangePassword="yes" Disabled="no"
LogonAsService="no" PasswordExpired="no" PasswordNeverExpires="yes" RemoveOnUninstall="yes" />
          </Component>
```

Значения почти всех атрибутов понятны и повторяют элементы диалогового окна, отображаемого в свойствах пользователя:

- Name – имя пользователя;
- Password – пароль;
- Domain – домен

Остальные атрибуты принимают значения yes или no. В первом случае описываемое условие выполняется, во втором – нет:

- CreateUser – создавать нового пользователя;
- FailIfExists – прерывать установку при наличии учетной записи;
- UpdateIfExists – обновить учетную запись, если она уже существует;
- CanNotChangePassword – пользователь не может самостоятельно изменить пароль;
- Disabled – учетная запись отключена;

- LogonAsService – вход в систему в качестве службы;
- PasswordExpired – пользователь должен сменить пароль при первом входе;
- PasswordNeverExpires – пароль никогда не устаревает;
- RemoveOnUninstall – удалить пользователя при деинсталляции.

Для добавления пользователя в одну или более существующую группу используются элементы Group и GroupRef. Описывающий группу элемент Group размещается в Product:

```
    <util:Group Id="AdminGroup" Name="Администраторы" />
```

Сложность заключается в том, что ссылка на группу разрешается по имени, которое может отличаться, например, в русской и английской версиях операционной системы. Если группа не найдена, попытка добавления в нее пользователя и весь процесс установки завершатся неудачей. Какого-либо универсального решения данной проблемы на текущий момент не существует. После завершения описания групп в них с использованием элемента GroupRef добавляются пользователи:

```
            <util:User Id="NewUser" CreateUser="yes" CanNotChangePassword="yes" Disabled="no"
Name="NewUser" Password="Pa$$w0rd123" PasswordExpired="no" PasswordNeverExpires="yes"
RemoveOnUninstall="yes" UpdateIfExists="no">
              <util:GroupRef Id="AdminGroup" />
              <util:GroupRef Id="UserGroup" />
            </util:User>
```

Некоторые элементы, например, Permission, в качестве значения атрибута User оперируют строковым значением, а не ссылкой, что усложняет и сужает область их применения. Строго говоря, работа с пользователями и группами в Windows Installer является, пожалуй, наиболее сложным и неоднозначным процессом, требующим внимания и тщательного тестирования.

Создание общего каталога

Создание общей папки является достаточно распространенным действием. Пусть при установке нашего приложения нам необходимо создать локальную папку Shared и сделать ее доступной пользователям под именем Temp.

```
            <Directory Id="SharedFolder" Name="Shared" >
              <Component Id="SharedFolderComponent" Guid="????????-9F9E-48AD-B564-
0FDC5BA92FC0">
                <CreateFolder/>
                <util:FileShare Id="SharedFolderTemp" Name="Temp">
                  <util:FileSharePermission User="Everyone" GenericAll="yes" />
                </util:FileShare>
              </Component>
            </Directory>
...
    <util:User Id="EveryoneUser" Name="Everyone" CreateUser="no" />
```

Сначала мы описываем каталог, затем явно создаем его – для этого используются уже рассмотренные нами ранее стандартные элементы. После делаем полученный каталог общим,

воспользовавшись элементами FileShare – для создания непосредственно общей папки и FileSharePermission, чтобы задать разрешения на доступ. Несмотря на обилие атрибутов в элементе FileSharePermission, влияние на систему оказывают только GenericAll – устанавливает все виды доступа – и GenericRead – устанавливает флажок «Чтение». Атрибут GenericRead должен устанавливаться совместно с каким-либо другим флагом, в качестве которого я использую Read.

Редактирование XML-файла

Хранение значений в XML-файлах является нормой для разработчиков управляемых приложений, где в них, прежде всего, хранятся настройки. И для целей редактирования таких файлов идеально подходит элемент XmlFile.

Пусть мы имеем привычный для .NET приложений конфигурационный файл:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <!-- Часть содержимого опущена за ненадобностью -->
  <userSettings>
    <DemoApplication.Properties.Settings>
      <setting name="ExternalApplication" serializeAs="String">
        <value>notepad.exe</value>
      </setting>
      <setting name="SecondParameter" serializeAs="String">
        <value>SecondParameterValue</value>
      </setting>
    </DemoApplication.Properties.Settings>
  </userSettings>
</configuration>
```

Как видно, в этом файле описываются два параметра, ExternalApplication и SecondParameter. Единственная сложность заключается в том, что имя искомого параметра размещается в атрибуте name элемента setting, поэтому для доступа к нужному значению необходимо указать правильное выражение. Приведенный ниже пример устанавливает значение параметра ExternalApplication равным текущему значению свойства APPLICATIONTOOPENLOG:

```
    <Component Id="MainExecutableConfig" Guid="????????-A713-44F0-B23B-D3D550DBF0F7">
      <File Id="DemoApplicationExeConfig" Name="DemoApplication.exe.config"
Source="$(var.DemoApplication.TargetDir)" DiskId="1" KeyPath="yes" />
        <!-- Установка значения элемента -->
        <util:XmlFile Id="XmlSettingsSetExternalApplication"
File="[INSTALLLOCATION]DemoApplication.exe.config" Action="setValue"
ElementPath="//setting[\[]@name='ExternalApplication'[\]]/value"
Value="[APPLICATIONTOOPENLOG]" />
    </Component>
```

Для элемента XmlFile указываются уникальный идентификатор в атрибуте Id и полный путь к целевому файлу в атрибуте File. Атрибут Action может принимать следующие значения:

- setValue – если не указан атрибут Name, то устанавливает текст элемента, иначе устанавливает значение атрибута, указанного в Name;
- deleteValue – если не указан атрибут Name, то удаляет текст элемента, иначе удаляет атрибут, указанный в Name;
- createElement – создает новый элемент. При этом указание атрибута Name является обязательным, а Value – нет;
- bulkSetValue – работает аналогично setValue, но устанавливает значения всех элементов в файле, удовлетворяющих выражению в ElementPath.

И, наконец, атрибут ElementPath позволяет задать выражение для поиска необходимого элемента. В нашем случае это простейшее XPath-выражение, позволяющее извлечь все узлы /setting/value, причем значение атрибута name для элемента setting должно быть равно ExternalApplication.

Если же, допустим, в XML-файле следует изменить значение не элемента, а атрибута, нам потребуется только немного изменить путь. Пусть мы хотим обработать элемент add в приведенном ниже файле.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MyPath" value="C:\Temp\" />
  </appSettings>
</configuration>
```

Тогда для изменения значения атрибута value наш элемент примет следующий вид:

```
        <util:XmlFile Id="XmlSettingsSetExternalApplication"
File="[INSTALLLOCATION]DemoApplication.exe.config" Action="setValue"
ElementPath="//configuration/appSettings/add[\[]@key='MyPath'[\]]/@value" Value="someValue" />
```

Для создания нового XML-файла, следует воспользоваться элементом XmlConfig. Его же следует применять в тех случаях, когда необходимо удалить существующий узел.

В качестве примера удаления рассмотрим более сложный случай из реального проекта. Серверное приложение является хостом для WCF-службы, при этом фрагмент его конфигурационного файла может выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="TestService">
        <endpoint address="net.tcp://localhost:1234/TestService" binding="netTcpBinding"
contract="TestAssembly.ITestService" />
        <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
      </service>
    </services>
  </system.serviceModel>
```

```
</configuration>
```

При установке из конфигурации необходимо удалить реализующую контракт IMetadataExchange конечную точку, позволяющую запрашивать метаданные службы. Такое поведение реализуется, как показано в примере ниже:

```
        <util:XmlConfig Id="RemoveEndpoint" File="[INSTALLLOCATION]Server.exe.config"
On="install" Action="delete" Node="element"
ElementPath="//configuration/system.serviceModel/services/service[\[]@name='TestService'[\]]"
VerifyPath="//configuration/system.serviceModel/services/service[\[]@name='TestService'[\]]/en
dpoint[\[]@binding='mexHttpBinding'[\]]" />
```

Здесь следует объяснить назначение атрибутов Action, Node, ElementPath, VerifyPath:

- Action – принимает значения create или delete, соответственно создает или удаляет узел;
- Node – принимает одно из значений element, value или document, указывая обрабатываемое содержимое – элемент, значение или документ полностью;
- ElementPath – XPath-выражение, указывающее на родительский для изменяемого элемент;
- VerifyPath – XPath-выражение, указывающее на изменяемый элемент.

Проверка отсутствия запущенного процесса, закрытие работающего процесса

В тех случаях, когда программа установки обновляет файлы, никакие процессы не должны удерживать их в момент перезаписи. В подавляющей части случаев такая ситуация корректно обрабатывается установщиком и выдается запрос на закрытие соответствующих приложений. Однако могут возникнуть ситуации, в которых существующие процессы необходимо остановить явно. Для этого нам потребуется элемент CloseApplication:

```
    <util:CloseApplication Id="CalcCloseAction" CloseMessage="yes" Target="calc.exe"
RebootPrompt="no" />
```

Установленный в yes атрибут CloseMessage позволяет отправить приложению запрос на завершение. Атрибут RebootPrompt запрашивает перезагрузку в случае, если процесс останется в работающем состоянии.

С использованием этого же элемента можно проверить наличие запущенного процесса, не принимая попыток остановить его. Для этого атрибуты CloseMessage и RebootPrompt устанавливаются в no – попытка останова процесса не выполняется, перезагрузка не требуется. При наличии активного процесса инициализируется свойство IsExcelRunning, указанное в атрибуте Property, после чего его значение может использоваться в других местах программы:

```
    <!-- Проверка, запущен ли процесс excel.exe -->
    <util:CloseApplication Id="ExcelCloseAction" CloseMessage="no" Target="excel.exe"
RebootPrompt="no" Property="IsExcelRunning" />
```

Чтобы воспользоваться значением свойства, проверку необходимо выполнять сразу после завершения операции WixCloseApplications. Так, в приведенном ниже примере наличие запущенного процесса excel.exe приводит к немедленному завершению программы установки:

```
    <InstallExecuteSequence>
      <Custom After="WixCloseApplications" Action="StopInstallation">IsExcelRunning</Custom>
    </InstallExecuteSequence>
    <!-- Прерывание установки, если Excel запущен -->
    <CustomAction Id="StopInstallation" Error="До начала установки надстройки необходимо
закрыть программу MS Office Excel." />
```

Установка разрешений на доступ к объектам

Правильное назначение разрешений на доступ к объектам является крайне важным шагом. Так, операционных системах Windows Vista и старше по умолчанию отсутствует доступ к содержимому каталога Program Files, поэтому вопрос настройки прав приобретает особую остроту. Описанный в расширении WixUtilExtension элемент PermissionEx позволяет управлять разрешениями для каталогов, файлов, ключей реестра и устанавливаемых служб. Рассмотрим данную возможность на примере задания разрешений для каталогов и файлов, как показано на рисунке 4.5.

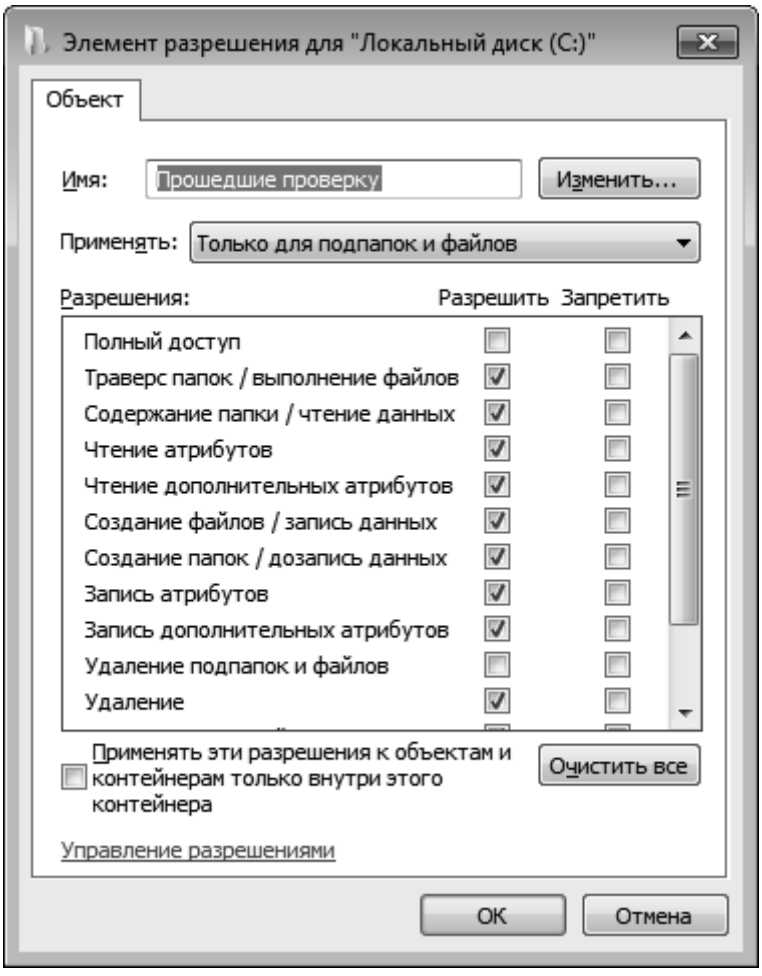

Рисунок 4.5 Разрешения на доступ к каталогам и файлам.

Разрешения для каталогов и файлов и устанавливающие их атрибуты сведены в таблице 4.1. Для краткости пришлось ввести сокращения для названий разрешений:

- полный доступ – ПД;
- траверс (обзор) папок/выполнение файлов – ОП;
- содержание папки/чтение данных – ЧД;
- чтение атрибутов – ЧА;
- чтение дополнительных атрибутов – ЧДА;
- создание файлов/запись данных – СФ;
- создание папок/дозапись данных – СП;
- запись атрибутов – ЗА;
- запись дополнительных атрибутов – ЗДА;
- удаление подпапок и файлов – УП;
- удаление – У;
- чтение разрешений – ЧР;
- смена разрешений – СР;
- смена владельца – СВ.

Таблица 4.1 Атрибуты элемента PermissionEx и устанавливаемые разрешения для доступа к каталогам и файлам.

Разрешение / Атрибут	ПД	ОП	ЧД	ЧА	ЧДА	СФ	СП	ЗА	ЗДА	УП	У	ЧР	СР	СВ
ChangePermission													●	
CreateChild							●							
CreateFile						●								
Delete											●			
DeleteChild										●				
GenericAll	●	●	●	●	●	●	●	●	●	●	●	●	●	●
GenericExecute		●		●								●		
GenericRead + Read			●	●	●							●		
GenericWrite						●	●	●	●			●		
Read			●											
ReadAttributes				●										
ReadExtendedAttributes					●									
ReadPermission												●		
TakeOwnership														●
Traverse		●												
WriteAttributes								●						
WriteExtendedAttributes									●					

Набор разрешений для разделов реестра отличается от разрешений для каталогов и приведен на рисунке 4.6.

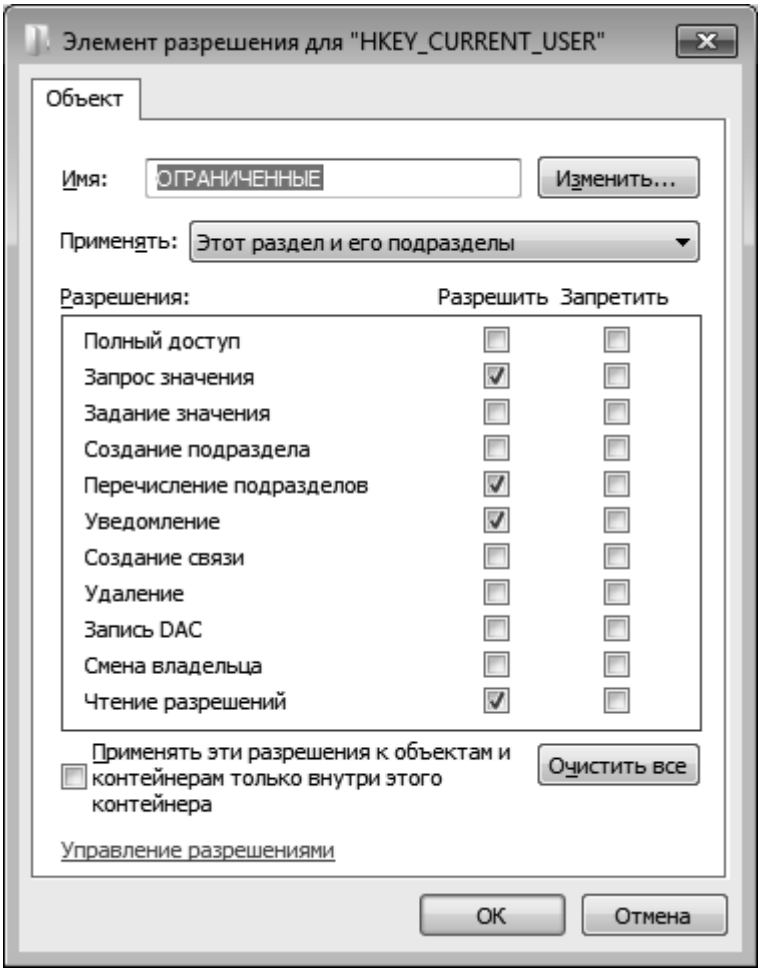

Рисунок 4.6 Разрешения на доступ к разделам реестра.

Разрешения для разделов реестра и устанавливающие их атрибуты сведены в таблице 4.2. Сокращения для названий разрешений:

- полный доступ – ПД;
- запрос значения – ЧЗ;
- задание значения – ЗЗ;
- создание подраздела – СП;
- перечисление подразделов – ПП;
- уведомление – У;
- создание связи – СС;
- удаление – УД;
- запись DAC – З;
- смена владельца – СВ;
- чтение разрешений – ЧР.

Таблица 4.2 Атрибуты элемента PermissionEx и устанавливаемые разрешения для доступа к разделам реестра.

Разрешение / Атрибут	ПД	ЧЗ	ЗЗ	СП	ПП	У	СС	УД	З	СВ	ЧР
ChangePermission									●		
CreateLink							●				
CreateSubkeys				●							
Delete								●			
EnumerateSubkeys					●						
GenericAll	●	●	●	●	●	●	●	●	●	●	●
GenericExecute		●			●	●					●
GenericRead + Read		●			●	●					●
GenericWrite			●	●							●
Notify						●					
Read		●									
ReadPermission											●
TakeOwnership										●	
Write			●								

Регистрация счетчиков производительности

Развертываемое приложение может взаимодействовать с существующими счетчиками производительности, так и регистрировать собственные. Регистрация категорий и счетчиков требует наличия административных полномочий, поэтому данную процедуру целесообразно выполнять в процессе установки программы.

```
          <Component Id="PerfCounter" Guid="????????-725C-4B49-901A-08201BD14E75">
            <CreateFolder />
            <util:PerformanceCategory Id="TestCategory" Name="Test Performance Category"
Help="Demo category" MultiInstance="yes" >
              <util:PerformanceCounter Name="Test Counter" Type="numberOfItems32" Help="Demo
performance counter" />
            </util:PerformanceCategory>
          </Component>
```

При создании категории в элементе PerformanceCategory указываются идентификатор, отображаемое название – если не указано, в качестве имени будет использован идентификатор, поясняющий текст в атрибуте Help и количество экземпляров счетчика в атрибуте MultiInstance. Последний атрибут наиболее важен – значение no устанавливает использование единственного экземпляра, а yes – возможность создания именованных экземпляров. Указание значения атрибута DefaultLanguage может привести к тому, что категория и счетчики зарегистрированы не будут.

Счетчики создаются внутри категории с помощью элемента PerformanceCounter. Кроме имени и поясняющего текста важным является указание типа счетчика. Всего доступны 28 типов, каждый из которых доступен в библиотеке MSDN в описании перечисления PerformanceCounterType.

Зарегистрированный с помощью приведенного выше фрагмента счетчик приведен на рисунке 4.7

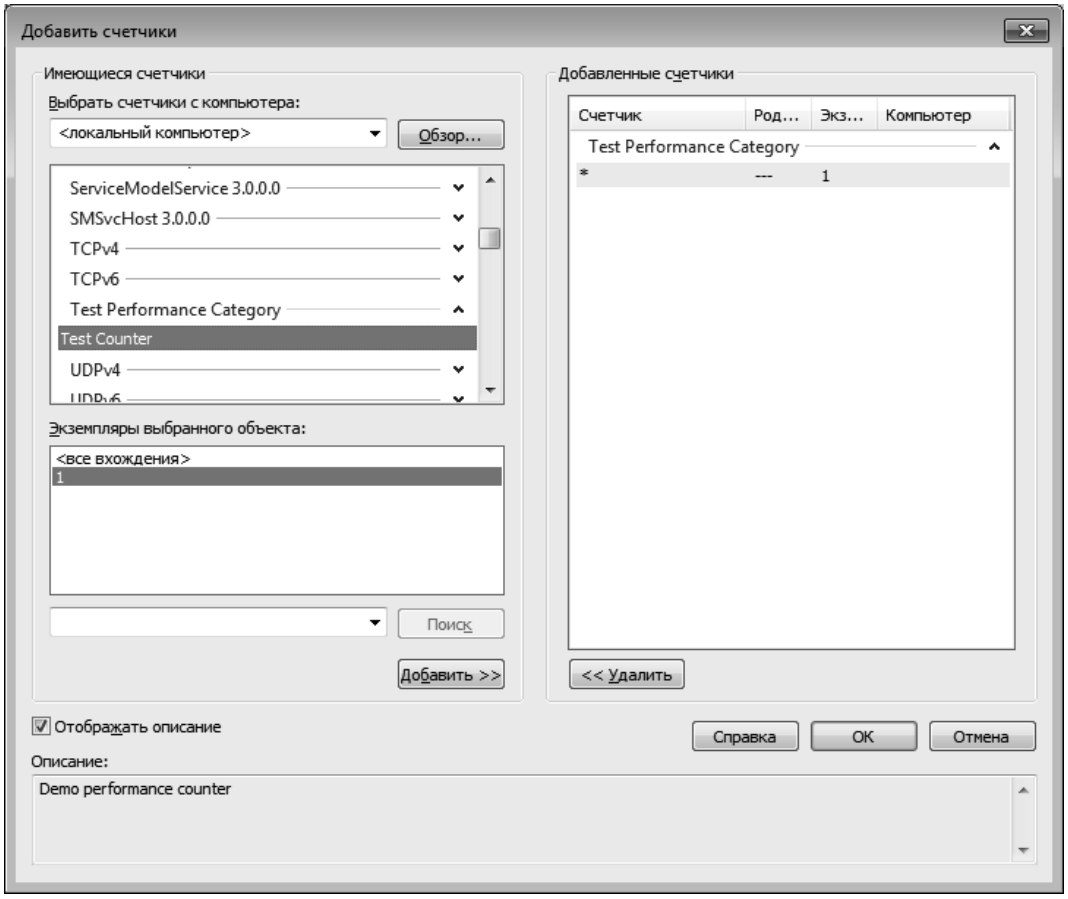

Рисунок 4.7 Зарегистрированный программой установки счетчик производительности.

Создание ссылок на веб-страницы

Иногда может потребоваться создать ярлык для веб-страницы. Сделать это позволяет элемент InternetShortcut. Кроме уникального идентификатора необходимо указать отображаемое имя и адрес гиперссылки:

```
          <Component Id="TdShortcutComponent" Guid="????????-6BF5-4EE1-BE60-4A4583172AE1">
            <CreateFolder/>
            <util:InternetShortcut Id="TechdaysShortcut" Name="TechDays.ru"
Target="http://www.techdays.ru/" />
          </Component>
```

Расширение WixNetFxExtension – работа с .NET Framework

Данное расширение предоставляет функции, востребованные в случае использования возможностей платформы .NET Framework. Оно содержит множество свойств и единственный

элемент, NativeImage, для подключения которого в заголовке документа необходимо добавить пространство имен «http://schemas.microsoft.com/wix/NetFxExtension»:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:netfx="http://schemas.microsoft.com/wix/NetFxExtension">
```

В случае, когда вам необходимы только свойства, объявлять пространство имен нет необходимости.

Генерация образа в машинном коде для .NET сборки

В некоторых случаях вы можете решить не использовать для своих сборок JIT-компиляцию, а воспользоваться утилитой ngen.exe для немедленного создания их образа в машинном коде и помещения его кэш сборок. Данный подход имеет свои плюсы и минусы и решение о его использовании должен принимать разработчик. Одной из важнейших особенностей наличия предварительно сгенерированного образа является ускорение загрузки приложений (не обязательно, необходимо тестирование каждого приложения). Возможность выполнить указанную процедуру предоставляет элемент NativeImage:

```
        <File Id="DemoApplicationExe" Name="$(var.DemoApplication.TargetFileName)"
Source="$(var.DemoApplication.TargetPath)" DiskId="1" KeyPath="yes" >
          <netfx:NativeImage Id="NativeImageDemoAppExe" Platform="32bit" Priority="0" />
        </File>
```

Platform указывает целевую платформу и может принимать значения 32bit, 64bit и All. Атрибут Priority может принимать значения 0, 1, 2 и 3, где 0 – наивысший приоритет, генерация образа выполняется немедленно в процессе установки, а 3 – низший, при котором образ будет генерироваться в момент бездействия машины. Кроме того, атрибуты Profile и Debug позволяют создавать сборку с возможностью запуска из профайлера и подключения отладчика. AppBaseDirectory и AssemblyApplication используются для DLL-сборок и предназначены для разрешения зависимостей при запуске в случае, когда для зависимых сборок не созданы образы.

Проверка наличия .NET Framework, .NET Framework SDK, Windows SDK

На сегодняшний день доступна уже четвертая версия .NET Framework. Кроме того, в различное время выходили пакеты обновления платформы, а также языковые пакеты для различных языков. Проверку наличия любого из этих пакетов предлагает данное расширение. Оно же предлагает несколько свойств, предназначенных для обнаружения .NET Framework SDK и Windows SDK.

Давайте подробнее рассмотрим предоставляемые расширением свойства. Прежде всего, свойство NETFRAMEWORKINSTALLROOTDIR возвращает базовый каталог, куда устанавливается .NET. Как правило, это каталог c:\WINDOWS\Microsoft.NET\Framework\. Все остальные свойства в целях экономии места сведены в таблице 4.3. Все свойства в таблице начинаются с префикса NETFRAMEWORK, после которого добавляется префикс, соответствующий версии .NET: 10 для версии 1.0, 35_CLIENT для .NET 3.5 CLIENT PROFILE, в таблице обозначаемый символом процента. Таким образом, свойство NETFRAMEWORK35 проверяет наличие на компьютере .NET версии 3.5, а свойство NETFRAMEWORK40FULL_RU_RU_LANGPACK – наличие языкового пакета для русского языка для полной версии .NET 4.0.

Таблица 4.3 Свойства для работы с .NET Framework.

Разрешение	1.0	1.1	2.0	3.0	3.5		4.0	
подмножество, если применимо	-	-	-	-	полный пакет	клиентский профиль	полный пакет	клиентский профиль
проверка наличия пакета и префикс для всех свойств группы (%)	10	11	20	30	35	35_CLIENT	40FULL	40CLIENT
каталог установки 32-х битной версии (%INSTALLROOTDIR)	●	●	●	●	●	●	●	●
каталог установки 64-х битной версии (%INSTALLROOTDIR64)			●	●	●		●	●
версия пакета обновления (%_SP_LEVEL)		●	●	●	●	●		
версия пакета обновления (%_SERVICING_LEVEL)							●	●
Языковые пакеты								
арабский язык (%_AR_SA_LANGPACK)							●	●
чешский язык (%_CS_CZ_LANGPACK)		●	●	●	●		●	●
датский язык (%_DA_DK_LANGPACK)		●	●	●	●		●	●
немецкий язык (%_DE_DE_LANGPACK)		●	●	●	●		●	●
греческий язык (%_EL_GR_LANGPACK)		●	●	●	●		●	●
испанский язык (%_ES_ES_LANGPACK)		●	●	●	●		●	●
финский язык (%_FI_FI_LANGPACK)		●	●	●	●		●	●
французский язык (%_FR_FR_LANGPACK)		●	●	●	●		●	●
иврит (%_HE_IL_LANGPACK)							●	●
венгерский язык (%_HU_HU_LANGPACK)		●	●	●	●		●	●
итальянский язык (%_IT_IT_LANGPACK)		●	●	●	●		●	●
японский язык (%_JA_JP_LANGPACK)		●	●	●	●		●	●
корейский язык (%_KO_KR_LANGPACK)		●	●	●	●		●	●
норвежский язык (%_NB_NO_LANGPACK)		●	●	●	●		●	●
голландский язык (%_NL_NL_LANGPACK)		●	●	●	●		●	●
польский язык (%_PL_PL_LANGPACK)		●	●	●	●		●	●
португальский (бразильский) язык (%_PT_BR_LANGPACK)		●	●	●	●		●	●

португальский язык (%_PT_PT_LANGPACK)		●	●	●	●		●	●
русский язык (%_RU_RU_LANGPACK)		●	●	●	●		●	●
шведский язык (%_SV_SE_LANGPACK)		●	●	●	●		●	●
турецкий язык (%_TR_TR_LANGPACK)		●	●	●	●		●	●
китайский язык (%_ZH_CN_LANGPACK)		●	●	●	●		●	●
китайский (тайваньский) язык (%_ZH_TW_LANGPACK)		●	●	●	●		●	●

Для использования свойства его необходимо объявить с использованием элемента PropertyRef:

```
<PropertyRef Id="NETFRAMEWORKINSTALLROOTDIR" />
<PropertyRef Id="NETFRAMEWORK10" />
<PropertyRef Id="NETFRAMEWORK11" />
<PropertyRef Id="NETFRAMEWORK20" />
<PropertyRef Id="NETFRAMEWORK20_SP_LEVEL" />
<PropertyRef Id="NETFRAMEWORK30" />
<PropertyRef Id="NETFRAMEWORK30_SP_LEVEL" />
<PropertyRef Id="NETFRAMEWORK35" />
<PropertyRef Id="NETFRAMEWORK35_SP_LEVEL" />
<PropertyRef Id="NETFRAMEWORK35_CLIENT" />
<PropertyRef Id="NETFRAMEWORK35_FR_FR_LANGPACK" />
```

После окончания установки откроем файл журнала Windows Installer, и найдем там значения переменных. Как видно, переменные для отсутствующих версий (.NET 1.0 и 1.1, .NET 3.5 CLIENT PROFILE) и языкового пакета для французского языка не были инициализированы значениями и не попали в журнал:

```
Property(C): NETFRAMEWORKINSTALLROOTDIR = c:\WINDOWS\Microsoft.NET\Framework\
Property(C): NETFRAMEWORK20 = #1
Property(C): NETFRAMEWORK20_SP_LEVEL = #2
Property(C): NETFRAMEWORK30 = #1
Property(C): NETFRAMEWORK30_SP_LEVEL = #2
Property(C): NETFRAMEWORK35 = #1
Property(C): NETFRAMEWORK35_SP_LEVEL = #1
```

Кроме того, мы имеем еще семь свойств, дающих информацию о версиях SDK:

- NETFRAMEWORK11SDKDIR и NETFRAMEWORK20SDKDIR возвращают путь к .NET Framework SDK версий 1.1 и 2.0 соответственно;
- свойства WINDOWSSDKCURRENTVERSIONDIR, WINDOWSSDK60ADIR, WINDOWSSDK61DIR и WINDOWSSDK70ADIR возвращают пути к текущей версии Windows SDK, версиям 6.0A, 6.1, 7.0A соответственно, а WINDOWSSDKCURRENTVERSION – номер текущей версии.

Расширение WixSqlExtension – управление базами данных SQL Server

При создании программ установки может возникнуть необходимость развертывания баз данных. На рынке существует множество различных СУБД и универсального решения данной задачи не существует, но Windows Installer XML предлагает нам расширение, позволяющее упростить управление базами данных для Microsoft SQL Server.

Данное расширение позволяет:

- создавать и удалять базы данных. Удаление баз при деинсталляции не рекомендуется – в случае наличия открытых соединений операция не будет выполнена;
- задавать свойства файла данных (*.mdf) и файла журнала (*.ldf) – их расположение, начальный размер и величину приращения при исчерпании доступного места;
- выполнять отдельные запросы и пакеты запросов.

Для использования описанных в расширении элементов подключим соответствующее пространство имен:

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi"
xmlns:sql="http://schemas.microsoft.com/wix/SqlExtension">
```

Расширение предоставляет пять элементов:

- SqlDatabase – обрабатывает базу данных. Если элемент помещается внутрь компонента, то используется для создания и удаления базы, а при размещении в элементе Product – только для ссылки на уже существующую, для выполнения над ней запросов;
- SqlFileSpec, SqlLogFileSpec – определяют параметры создаваемого файла данных и файла журналов соответственно;
- SqlScript, SqlString – исполнение SQL-скрипта из файла или отдельной SQL-команды соответственно.

Рассмотрим подробнее атрибуты предлагаемых элементов.

Для SqlDatabase, кроме идентификатора, важными являются:

- Database – имя базы данных, может содержать форматированную строку;
- Server – имя или IP-адрес сервера. Если используется отличный от 1433 порт, он указывается через запятую после имени сервера, например «127.0.0.1,9900»;
- Instance – используется при обращении к именованному экземпляру;
- User – идентификатор пользователя, используемого для соединения;
- атрибуты CreateOn…/DropOn… позволяют определить момент создания и/или удаления базы данных. Следует внимательно относиться к удалению базы данных – наличие открытых соединений не позволит успешно завершить обработку. Игнорировать ошибки позволяет атрибут ContinueOnError.

Свойства файлов задаются с помощью элементов SqlFileSpec и SqlLogFileSpec, имеющих одинаковый набор атрибутов:

- Filename – полное имя файла, может содержать форматированную строку;

- Name – логическое имя файла;
- Size – начальный размер файла. Размеры могут устанавливаться в килобайтах, мегабайтах, гигабайтах. Для этого после числового значения указывается суффикс «KB», «MB» или «GB»;
- GrowthSize – размер приращения объема при необходимости. Устанавливается аналогично Size, но также может выражаться в процентах. Значение по умолчанию – «10%»;
- MaxSize – максимальный объем файла. Если не указан, то не ограничен. Значения указываются аналогично Size.

Элементы SqlScript и SqlString также очень похожи. Отличие заключается в наличии у SqlScript атрибута BinaryKey, куда помещается идентификатор файла скрипта; для SqlScript определен атрибут SQL, куда помещается текст запроса. Остальные атрибуты полностью совпадают:

- Sequence – числовое значение, определяющее взаимный порядок выполнения скриптов и команд;
- ExecuteOn…/RollbackOn… – две взаимоисключающие группы атрибутов, устанавливающие выполнение при установке или откате установки;
- SqlDb – используется для указания идентификатора базы данных в случаях, когда элемент не является дочерним для SqlDatabase. Применяется для изменения существующих баз данных.

В тестовом примере мы используем все предоставляемые данным расширением элементы: создадим базу данных, установим параметры файла данных и журнала, после чего поочередно выполним скрипт и отдельный SQL-запрос. SQL-скрипт будет содержать в себе единственную команду, создающую таблицу:

```
CREATE TABLE WixTable (Id int, Value varchar(50))
```

Сохраним скрипт с именем WixDatabaseStructure.sql, добавим полученный файл в проект и подключим с помощью элемента Binary:

```
    <Binary Id="WixDatabaseStructureSql" SourceFile="WixDatabaseStructure.sql" />
```

Теперь подключимся к СУБД. Это можно сделать как от имени текущего пользователя, так и от имени учетной записи SQL Server, если такая возможность активирована. Для использования встроенной аутентификации достаточно оставить пустым атрибут User элемента SqlDatabase. Если же необходимо подключиться от имени, например, пользователя sa, его надо подключить с помощью элемента User из расширения WixUtilExtension. В этом случае нам потребуются реквизиты пользователя sa:

```
            <util:User Id="user_sa" Name="sa" CreateUser="no" FailIfExists="no"
Password="abc*$123456" />
```

Идентификатор пользователя мы поместим в атрибут User элемента SqlDatabase. Простой законченный пример создающего базу данных компонента приведен ниже. Все элементы, кроме SqlDatabase, являются необязательными.

```
         <Component Id="SqlDatabaseComponent" Guid="????????-3FBC-4ACF-B650-D1D05B643EDD">
           <CreateFolder Directory="INSTALLLOCATION" />
```

```
            <sql:SqlDatabase Id="TestSqlDatabase" Database="WixDatabase" CreateOnInstall="yes"
Server="127.0.0.1" DropOnUninstall="no" User="user_sa">
              <sql:SqlFileSpec Id="SqlDataFile" Filename="[INSTALLLOCATION]WixDatabase.mdf"
Name="WixDatabaseDatafile" />
              <sql:SqlLogFileSpec Id="SqlLogFile" Filename="[INSTALLLOCATION]WixDatabase.ldf"
Name="WixDatabaseLogfile" />
              <sql:SqlScript Id="DatabaseStructureScript" BinaryKey="WixDatabaseStructureSql"
ExecuteOnInstall="yes" Sequence="1" />
              <sql:SqlString Id="DataValues" SQL="INSERT INTO WixTable (Id, Value) VALUES (1,
'test value')" ExecuteOnInstall="yes" Sequence="2" />
            </sql:SqlDatabase>
          </Component>
```

Таким образом, расширение WixSqlExtension позволяет решать широкий круг задач, связанных с развертыванием баз данных Microsoft SQL Server. Это связано с возможностью выполнить любую задачу, в том числе настройку свойств базы данных средствами Transact-SQL – достаточно подготовить соответствующий скрипт и выполнить его в процессе работы программы установки.

Глава 5. Настройка и расширение интерфейса

Необходимость взаимодействия с пользователем в процессе установки возникает очень часто: для отображения информации, установки значений различных параметров – просмотра текста лицензионного соглашения и ввода серийного номера, выбора каталога для установки, наборов с требуемой функциональностью и многих других. Windows Installer XML позволяет создавать удобные, профессионально выглядящие интерфейсы пользователя, ограниченные только возможностями технологии Windows Installer и дизайнерскими талантами создающей продукт команды.

Встроить пользовательский интерфейс в WiX можно различными способами:

- добавлением стандартных, входящих в комплект поставки наборов диалогов;
- изменением стандартных наборов, добавляя, удаляя и изменяя диалоги;
- созданием своего собственного набора диалогов.

Первый способ наиболее прост и позволяет получить полноценный интерфейс, добавив в файл сценария всего несколько строк кода. Существующие наборы диалогов способны удовлетворить значительную часть потребностей, их может быть достаточно для большинства создаваемых программ установки.

Если ни один из стандартных наборов не предоставляет необходимой функциональности, можно модифицировать наиболее подходящий. Нам доступны исходные коды WiX и всех расширений, но такие изменения все равно потребуют понимания принципов и элементов, используемых для описания диалогов и взаимосвязей между ними. Второй раздел этой главы полностью посвящен данному вопросу.

И, наконец, однажды вам потребуется добавить в набор свой диалог. Возможно, он будет только отображать несколько строк текста, или позволит передать программе установки вводимое пользователем строковое значение. Вне зависимости от задач, способ добавления диалогов всегда одинаков, поэтому вопрос добавления диалога вынесен в отдельный раздел.

Редактирование диалогов не ограничивается добавлением на них надписей, кнопок и окон для ввода текста. Технология Windows Installer предоставляет в распоряжение разработчика множество элементов управления, в том числе специфических. В посвященном элементам управления разделе рассматриваются их основные свойства и приводятся примеры использования.

И, наконец, нельзя обойти стороной механизм событий. Даже простейшие окна используют события для перехода к другим диалогам в наборе, а реализовать сложную логику без их использования просто невозможно. Описанием механизма событий завершается данная глава.

Для добавления в программу любого, даже самого простого интерфейса пользователя, необходимо начать с подключения расширения WixUIExtension. Подключение расширений описывается в разделе «Добавление ссылок на проекты и библиотеки» главы 2. Все используемые элементы описаны в стандартном пространстве имен, прописывать отдельное пространство имен для данного расширения (в отличие от всех остальных) нет необходимости. Единственный случай, когда данное расширение не потребуется – если вы решите создать собственный набор диалогов «с нуля» - но данный подход достаточно трудоемок.

Стандартные наборы диалогов и их простая настройка

В расширении WixUIExtension содержатся описания около трех десятков различных диалогов. Среди них, в том числе, все стандартные, описанные в MSI SDK. Из этих диалогов составлены пять стандартных наборов, реализующих наиболее часто применимые сценарии установки приложений. Наличие заранее описанных наборов делает их добавление в пакет установки простейшей задачей – необходимо только выбрать тот, который в наибольшей степени удовлетворяет вашим потребностям. Чтобы выбрать было проще, в таблице 5.1 перечислены диалоги, входящие в комплект каждого из стандартных наборов. Подробное описание всех диалогов приведено в приложении. Наиболее часто используемым является набор WixUI_Mondo, предоставляющий собой привычный и знакомый по множеству программ установки интерфейс.

Таблица 5.1 Состав стандартных наборов диалогов.

Название набора диалогов	Внутреннее название диалога
WixUI_Minimal	WelcomeEulaDlg
WixUI_Mondo	BrowseDlg, CustomizeDlg, DiskCostDlg, LicenseAgreementDlg, SetupTypeDlg, WelcomeDlg
WixUI_InstallDir	BrowseDlg, DiskCostDlg, InstallDirDlg, InvalidDirDlg, LicenseAgreementDlg, WelcomeDlg
WixUI_FeatureTree	BrowseDlg, CustomizeDlg, DiskCostDlg, LicenseAgreementDlg, WelcomeDlg
WixUI_Advanced	AdvancedWelcomeEulaDlg, BrowseDlg, DiskCostDlg, FeaturesDlg, InstallDirDlg, InstallScopeDlg, InvalidDirDlg
Все наборы	CancelDlg, ErrorDlg, ExitDlg, FatalError, FilesInUse, MaintenanceTypeDlg, MaintenanceWelcomeDlg, MsiRMFilesInUse, OutOfDiskDlg, OutOfRbDiskDlg, PrepareDlg, ProgressDlg, ResumeDlg, UserExit, VerifyReadyDlg, WaitForCostingDlg

Выбрав подходящий набор, подключим его. Для этого добавим ссылку на набор WixUI_Mondo – или любой другой, воспользовавшись элементом UIRef. Поместить его можно непосредственно внутрь элемента Product, либо внутрь элемента UI. Сюда же добавим ссылку на WixUI_ErrorProgressText, предоставляющий локализованный текст сообщений об ошибках и служебных сообщений.

```
<!-- Интерфейс пользователя -->
<UI>
  <UIRef Id="WixUI_Mondo" />
  <UIRef Id="WixUI_ErrorProgressText" />
</UI>
```

И это все. В результате сборки мы получим msi-пакет, обладающий интерфейсом пользователя. Остальные стандартные наборы, за исключением WixUI_Advanced, подключаются аналогично.

Набор WixUI_Advanced

Отличие WixUI_Advanced заключается в том, что он позволяет пользователю выбрать установку программы в режиме «для всех пользователей» или «только для меня». Для поддержки данных режимов нам потребуется выполнить несколько дополнительных условий. Прежде всего, каталог для копирования файлов должен иметь идентификатор APPLICATIONFOLDER. Кроме того, необходимо определить свойство ApplicationFolderName, содержащее имя каталога для установки:

```
<Property Id="ApplicationFolderName" Value="My Application Folder" />
```

Вне зависимости от описанной структуры каталогов, приложение будет установлено:

- «для всех пользователей» - в каталог [ProgramFilesFolder][ApplicationFolderName];
- «только для меня» - в каталог [LocalAppDataFolder]Apps\[ApplicationFolderName].

Также необходимо указать значение свойства WixAppFolder. Принимая значения WixPerMachineFolder или WixPerUserFolder, оно устанавливает состояние переключателя по умолчанию, «для всех пользователей» или «только для меня» соответственно.

```
<Property Id="WixAppFolder" Value="WixPerMachineFolder" />
```

Если пользователь не должен производить указанный выбор, есть возможность установить значение принудительно, установив в ноль значение одного из двух свойств: WixUISupportPerMachine или WixUISupportPerUser.

```
<WixVariable Id="WixUISupportPerUser" Value="0" />
```

Если приравнять нулю свойство WixUISupportPerUser, программа будет установлена для всех пользователей компьютера, если же WixUISupportPerMachine – только для текущего. Следует учесть, что при этом вам потребуется дополнительно указать значение глобального свойства ALLUSERS.

Корректными значениями для свойства ALLUSERS являются 1, 2 и пустая строка. Пустая строка конфигурирует установку для пользователя, значение 1 – для всех пользователей. Значение 2 интерпретируется по-разному, в зависимости от версии операционной системы и ее текущей настройки.

Простая настройка внешнего вида стандартных диалогов

Заменив всего несколько элементов внешнего вида диалогов на наши собственные, мы можем придать стандартным наборам вполне приемлемый вид. Для этого предназначены свойства, перечисленные в таблице 5.2.

Таблица 5.2 Стандартные свойства для настройки внешнего вида диалогов.

Имя переменной	Описание
WixUIBannerBmp	Изображение, размещаемое в верхней части всех диалогов, кроме первого и последнего (при наличии), размер изображения 493 × 58 пикселей
WixUIDialogBmp	Фоновое изображение на первом и последнем диалогах, размер изображения 493 × 312 пикселей
WixUIExclamationIco	Восклицательный знак - пиктограмма для диалога WaitForCostingDlg, размер 32 × 32 пикселя
WixUIInfoIco	Информация – пиктограмма для диалогов отмены и ошибки, размер 32 × 32 пикселя
WixUINewIco	Пиктограмма для диалога BrowseDlg для кнопки создания диалога, размер 16 × 16 пикселей
WixUIUpIco	Пиктограмма для диалога BrowseDlg для кнопки перехода в родительский каталог, размер 16 × 16 пикселей
WixUILicenseRtf	Имя файла с текстом лицензии в формате RTF

Использовать эти свойства просто. Сначала следует подготовить файлы, например файлы изображений и лицензионного соглашения, скопировать их в каталог с проектом, а затем добавить в проект, воспользовавшись пунктом меню Add -> Existing Item. Останется только указать имена добавленных файлов в качестве значений интересующих нас свойств.

```
<!-- Дополнительные свойства для персонализации интерфейса -->
```

```
    <WixVariable Id="WixUIBannerBmp" Value="BannerImageSizes.bmp" />
    <WixVariable Id="WixUIDialogBmp" Value="DialogImageSizes.bmp" />
    <WixVariable Id="WixUILicenseRtf" Value="License.rtf" />
```

Конечно, это только наиболее часто употребляемые свойства. Как мы увидим далее в этой же главе, имея представление об описании интерфейса, можно как угодно изменить внешний вид диалогов.

Замечание: при подготовке файлов изображения следует учитывать, что размеры диалогов и элементов управления указываются не в пикселях, а в так называемых Installer Units. Чтобы изображение не выглядело искаженным, следует выполнять пересчет размеров. Так, для 96 DPI отношение будет 4/3, а для 120 DPI – 5/3. Принимая за образец 96 DPI, получаем: для элемента управления Bitmap шириной (Width) 330 единиц и высотой (Height) 110 единиц размеры изображения в пикселях рассчитываются как Width = 330 * 1,33 = 439, Height = 110 * 1,33 = 146.

Наборы диалогов – взгляд внутрь

Мы умеем подключать стандартные наборы. В этом разделе мы воспользуемся преимуществами, которые дает нам свободный доступ к исходным текстам Windows Installer XML, и взглянем немного глубже. Это потребуется для того, чтобы в случае необходимости суметь получить возможность управлять внешним видом и поведением наборов. Существующие тексты разметки стандартных диалогов и описания наборов – прекрасный материал для экспериментов, который часто становится основой для практических решений. Сначала возьмем исходные тексты WiX, пройдя по адресу http://wix.sourceforge.net/releases/, где можно найти любую интересующую нас версию. Скачав и распаковав архив, внутри мы найдем решения в формате Visual Studio 2008. Из него можно извлечь массу полезных сведений, но нас в данном случае интересует только содержимое каталога \src\ext\UIExtension\wixlib. Еще раз вернемся к содержимому таблицы 5.1 – каждый из присутствующих в ней элементов описан в одноименном файле с расширением wxs, как наборы, так и отдельные диалоги.

Рассмотрим описание на примере набора WixUI_FeatureTree, включающем в себя меньшее, чем другие - всего пять – число диалогов. Откроем файл WixUI_FeatureTree.wxs в любом XML редакторе. Сначала идет обязательный заголовок с объявлением пространства имен по умолчанию, после него – элемент Fragment, описывающий данный документ не как самостоятельный файл сценария, но как подключаемый фрагмент. За заголовком следует элемент UI, атрибуту Id которого в нашем случае присвоено значение WixUI_FeatureTree. Это идентификатор, позволяющий ссылаться на данный набор из других файлов сценариев. В документе обязательно присутствует ссылка на набор WixUI_Common, который содержит описания общих текстовых переменных, а также подключает необходимые библиотеки. Все остальные элементы, относящиеся к настройке интерфейса, должны быть размещены внутри элемента UI.

```
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Fragment>
        <UIRef Id="WixUI_Common" />
        <UI Id="WixUI_FeatureTree">
            <!-- Здесь находятся элементы, описывающие интерфейс -->
        </UI>
```

```
    </Fragment>
</Wix>
```

В первую очередь в теге UI определяются три стиля текста, устанавливающие шрифт. Имя и размер указываются обязательно, также можно использовать дополнительные атрибуты стиля (жирный - Bold, курсив - Italic, подчеркнутый - Underline, перечеркнутый - Strike) и цвет в виде совокупности компонентов Red, Green и Blue.

```
      <TextStyle Id="WixUI_Font_Normal" FaceName="Tahoma" Size="8" />
      <TextStyle Id="WixUI_Font_Bigger" FaceName="Tahoma" Size="12" />
      <TextStyle Id="WixUI_Font_Title" FaceName="Tahoma" Size="9" Bold="yes" />
```

Стиль WixUI_Font_Bigger используется для отображения надписей на начальной и конечной страницах, а WixUI_Font_Title – для отображения наименования текущего диалога в заголовке промежуточных диалогов. Далее один из стилей назначается для использования в качестве основного, для чего его идентификатор присваивается стандартному свойству Windows Installer – DefaultUIFont.

```
      <Property Id="DefaultUIFont" Value="WixUI_Font_Normal" />
```

Если не присваивать значение свойству, то при сборке мы получим ошибку валидации ICE31. Подавив, в качестве эксперимента, проверку ICE31 на закладке Tool Settings в свойствах проекта, мы получим проект с достаточно неудачным шрифтом текста, поэтому данное присваивание достаточно важно. Далее в тексте может определяться свойство WixUI_Mode. Оно является устаревшим и не должно использоваться в наборах. В WiX версии 3.5 ни один из диалогов не использует его, поэтому данное объявление можно безболезненно удалить.

Теперь подключаются служебные диалоги. Нет необходимости включать их в последовательность переходов, так как они вызываются стандартными диалогами по мере необходимости.

```
            <DialogRef Id="ErrorDlg" />
            <DialogRef Id="FatalError" />
            <DialogRef Id="FilesInUse" />
            <DialogRef Id="MsiRMFilesInUse" />
            <DialogRef Id="PrepareDlg" />
            <DialogRef Id="ProgressDlg" />
            <DialogRef Id="ResumeDlg" />
            <DialogRef Id="UserExit" />
```

Далее, путем назначения обработчиков событиям описывается последовательность переходов между диалогами. Для назначения обработчика используется элемент Publish, для которого в данном случае необходимо указать значения атрибутов Event и Value. В атрибут Value помещается идентификатор вызываемого диалога, Event должен содержать название обрабатываемого события. В примере элементы Publish располагаются непосредственно в теге UI, поэтому для каждого публикуемого события необходимо также указать диалог и генерирующий его элемент управления – для этого используются атрибуты Dialog и Control.

```
            <Publish Dialog="ExitDialog" Control="Finish" Event="EndDialog" Value="Return"
Order="999">1</Publish>
```

```
            <Publish Dialog="WelcomeDlg" Control="Next" Event="NewDialog"
Value="LicenseAgreementDlg">1</Publish>

            <Publish Dialog="LicenseAgreementDlg" Control="Back" Event="NewDialog"
Value="WelcomeDlg">1</Publish>
            <Publish Dialog="LicenseAgreementDlg" Control="Next" Event="NewDialog"
Value="CustomizeDlg">LicenseAccepted = "1"</Publish>

            <Publish Dialog="CustomizeDlg" Control="Back" Event="NewDialog"
Value="MaintenanceTypeDlg" Order="1">Installed</Publish>
            <Publish Dialog="CustomizeDlg" Control="Back" Event="NewDialog"
Value="LicenseAgreementDlg" Order="2">NOT Installed</Publish>
            <Publish Dialog="CustomizeDlg" Control="Next" Event="NewDialog"
Value="VerifyReadyDlg">1</Publish>
            <!-- Остальные переходы описываются аналогично -->
```

В демонстрирующемся выше фрагменте кода обработчики назначаются не для всех кнопок; при отсутствии явного назначения используется то, которое описано в исходном файле для каждого диалога. Это позволяет не повторять дублирующиеся элементы, а наследовать их. Также следует обратить внимание на возможность генерации событий в определенной последовательности и в зависимости от выполнения условия. Так, например, в приведенном ниже фрагменте будет сгенерировано только одно из событий: если продукт установлен (инициализировано свойство Installed), будет отображен диалог MaintenanceTypeDlg, в противном случае – диалог LicencseAgreementDlg. Чтобы событие генерировалось всегда, внутрь элемента достаточно поместить единицу.

```
            <Publish Dialog="CustomizeDlg" Control="Back" Event="NewDialog"
Value="MaintenanceTypeDlg" Order="1">Installed</Publish>
            <Publish Dialog="CustomizeDlg" Control="Back" Event="NewDialog"
Value="LicenseAgreementDlg" Order="2">NOT Installed</Publish>
```

В приведенном для примера наборе WixUI_FeatureTree используются только два типа событий, одно из которых возникает при нажатии на кнопку – NewDialog, а также EndDialog, вызываемое при завершении установки. На самом деле общее количество доступных событий – более тридцати, их использование и сам механизм событий описываются в соответствующем разделе в конце данной главы.

Добавление простого диалога

Рано или поздно, нам потребуется добавить собственный диалог. Для простоты примера он будет содержать только кнопки «Назад», «Далее», некоторый текст, и иметь идентификатор EmptyDlg. Удобнее не создавать диалог с нуля, а взять за основу один из существующих. Описание диалога будет выглядеть так:

```
            <Dialog Id="EmptyDlg" Width="370" Height="270" Title="Пустой диалог">
              <Control Id="Title" Type="Text" X="15" Y="6" Width="200" Height="15"
Transparent="yes" NoPrefix="yes" Text="{\WixUI_Font_Title}Пустой диалог" />
```

```
            <Control Id="SomeText" Type="Text" X="15" Y="123" Width="340" Height="15"
Transparent="yes" NoPrefix="yes" Text="Некоторый текст на диалоге EmptyDlg" />
            <Control Id="Next" Type="PushButton" X="236" Y="243" Width="56" Height="17"
Default="yes" Text="!(loc.WixUINext)">
                <Publish Event="NewDialog" Value="CustomizeDlg" Order="10" />
              </Control>
              <Control Id="Back" Type="PushButton" X="180" Y="243" Width="56" Height="17"
Text="!(loc.WixUIBack)">
                <Publish Event="NewDialog" Value="LicenseAgreementDlg" Order="10" />
              </Control>
          </Dialog>
```

Как видно, описание помещается внутри элемента Dialog, который, в свою очередь, размещается в теге UI. Описание сделано на основе содержимого файла LicenseAgreementDlg.wxs, в который внесены минимальные изменения. В диалоге использованы две кнопки (Back и Next), для каждой из которых с помощью элемента Publish публикуется событие NewDialog. Данные события формируют последовательность переходов и делают диалог пригодным для работы. Кроме того, здесь же присутствуют два элемента типа Text, формирующие статический текст. Интерес здесь представляет элемент Title – помещенное перед текстом значение {\WixUI_Font_Title} позволяет использовать для написания указанный стиль текста. Объявление стилей с использованием элемента TextStyle описано в предыдущем разделе.

Полученный диалог мы добавим в последовательность WixUI_Mondo, поместив его между диалогами LicenseAgreementDlg и CustomizeDlg. Здесь необходимо учесть один важный момент: поскольку взят готовый набор диалогов, в нем уже описан порядок публикации событий, предусматривающий переход от диалога LicenseAgreementDlg к CustomizeDlg и обратно. Удалить публикуемые события нельзя, но можно добавить собственные таким образом, чтобы они выполнялись после стандартных. Если элемент управления последовательно генерирует несколько событий NewDialog, переход будет выполнен к последнему. Изменим последовательность переходов для диалогов LicenseAgreementDlg и CustomizeDlg следующим образом, вставив между ними EmptyDlg:

```
      <Publish Dialog="LicenseAgreementDlg" Control="Next" Event="NewDialog" Value="EmptyDlg"
Order="10" >LicenseAccepted = "1"</Publish>
      <Publish Dialog="CustomizeDlg" Control="Back" Event="NewDialog" Value="EmptyDlg"
Order="10">NOT Installed</Publish>
```

Каждому из публикуемых событий явно присвоен порядок вызова, для чего использован атрибут Order. Чтобы не искать перечень публикуемых событий в исходных кодах или с помощью утилиты Orca, можно установить порядок очередного события равным достаточно большому числу – значение 10 подойдет.

Соберем проект и запустим его. Мы увидим изображение, аналогичное приведенному на рисунке 5.1.

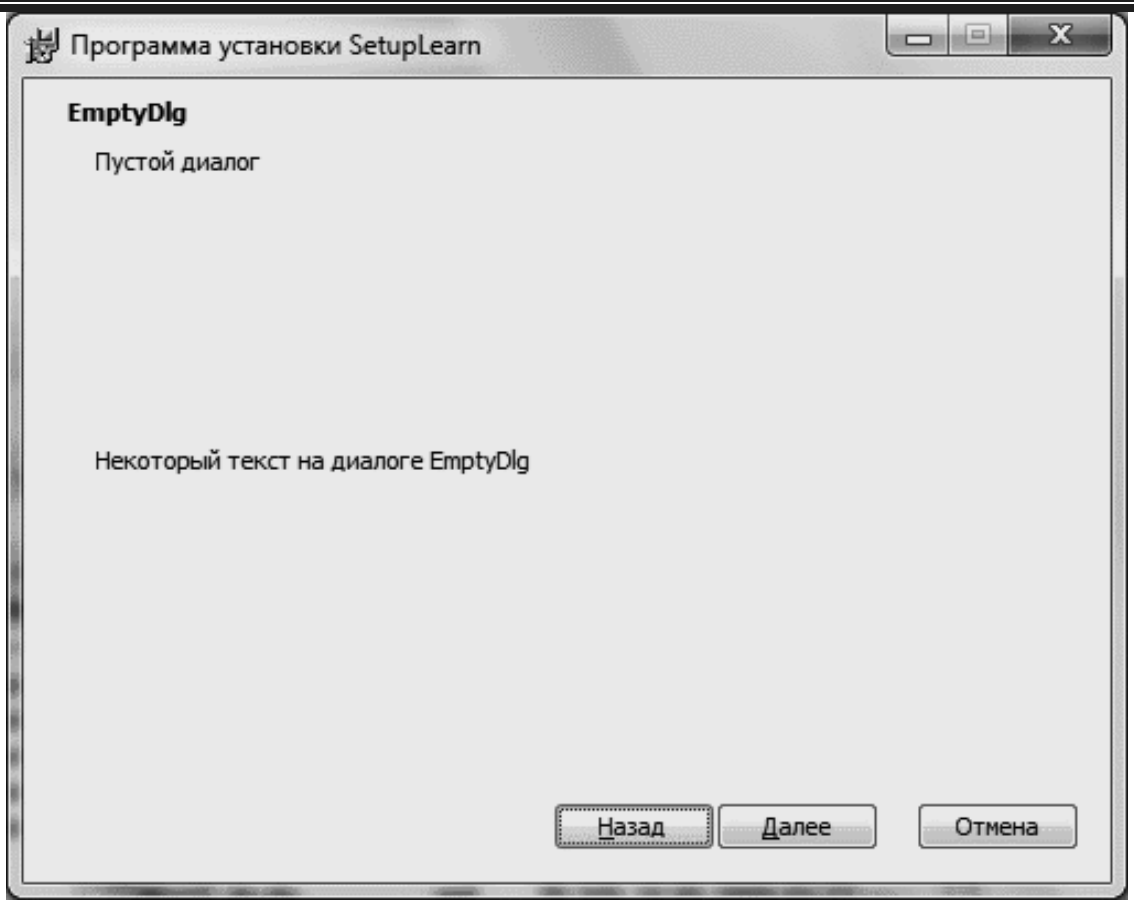

Рисунок 5.1 Пустой диалог.

Элементы управления

При разработке интерфейсов мы подключаем элементы управления, которые предоставляет Windows Installer. Некоторые из них – кнопки, текст, переключатели – настолько распространены, что их использование не вызывает никаких сложностей, в то время как другие специфичны для программ установки. Их подключение не столь очевидно, поэтому в данном разделе я рассмотрю все доступные элементы управления и приведу примеры их использования.

Элементы управления можно условно разделить на пассивные и активные. Активные элементы могут изменять свое состояние, для чего они ассоциируются со свойствами. Состояние пассивных элементов не может изменяться, но некоторые из них могут порождать события. К способным генерировать события элементам относятся кнопка (PushButton), переключатель (CheckBox) и SelectionTree.

Для описания всех элементов управления применяется элемент Control. Он имеет множество атрибутов, но каждый элемент управления использует только некоторое их подмножество. Так, для каждого элемента управления обязательно должны быть указаны значения следующих атрибутов:

- Id - уникальный идентификатор в пределах диалога;
- Type - наименование типа элемента управления, значение из таблицы 5.3;
- Height - высота;
- Width - ширина;

- X - координата X;
- Y - координата Y.

Ряд других атрибутов также применим ко всем элементам управления, но они являются необязательными:

- Default – устанавливает срабатывание элемента при нажатии клавиши «Enter». Работает только для элементов управления, способных получить фокус ввода, элементом по умолчанию устанавливается первый активный в списке);
- Disabled – управляет доступностью элемента. «no» (по умолчанию) - доступен, «yes» - недоступен;
- Hidden – управляет видимостью элемента, аналогично Disabled;
- Sunken – придает внешнему виду элемента «утопленный» вид, добавляя рамку;
- TabSkip – при значении «yes» пропускает элемент при обходе с использованием клавиши табуляции. Работает только для элементов управления, способных получить фокус ввода;
- ToolTip – текст всплывающей подсказки, работает для элементов, способных получить фокус ввода.

Таблица 5.3 Перечень доступных элементов управления.

Элемент управления	Связанное свойство	Краткое описание
Billboard		Панель для отображения изменяемого содержимого в процессе установки
Bitmap		Изображение в формате BMP или JPG
CheckBox	●	Переключатель с двумя доступными состояниями
ComboBox	●	Раскрывающийся список с возможностью редактирования значения
DirectoryCombo	●	Раскрывающийся список с деревом каталогов
DirectoryList	●	Список дочерних по отношению к текущему каталогов
Edit	●	Редактор строковых или целочисленных значений
GroupBox	●	Рамка, позволяющая визуально группировать объекты
Hyperlink		Гиперссылка, открываемая в браузере. Поддерживается, начиная с Windows Installer 5.0
Icon		Пиктограмма
Line		Горизонтальная линия
ListBox	●	Список без возможности редактирования значения
ListView	●	Список доступных для выбора значений
MaskedEdit	●	Поле для редактирования с поддержкой маски ввода
PathEdit	●	Текстовое поле, отображает каталог или полный путь, проверяет введенное значение на допустимость
ProgressBar		Индикатор хода установки
PushButton		Кнопка
RadioButtonGroup	●	Группа взаимоисключающих переключателей
ScrollableText		Отображение форматированного текста с поддержкой прокрутки

SelectionTree	●	Элемент управления, отображающий информацию о доступных наборах и позволяющий пользователю производить выбор
Text		Текст
VolumeCostList		Перечень доступных носителей с расчетом необходимого для выполнения установки дискового пространства
VolumeSelectCombo	●	Раскрывающийся список доступных дисков

Давайте перейдем к рассмотрению доступных элементов управления.

Элементы оформления (Bitmap, Icon, Line, GroupBox, Hyperlink, Text, ScrollableText)

К элементам оформления я отнес те, которые используются для декорации. Не неся в себе функциональной нагрузки, они востребованы при разработке любого интерфейса. Здесь же описан элемент Hyperlink, позволяющий выполнять переход к указанной веб-странице.

Элементы Bitmap и Icon предназначены для вывода изображений и пиктограмм. Bitmap поддерживает форматы jpg и bmp, Icon – ico. Отличие между ними заключается в том, что файл пиктограммы, как будет показано ниже, может одновременно хранить несколько изображений различного размера.

Элемент Icon поддерживает отображение пиктограмм различных размеров с масштабированием или без. Чтобы использовать, их необходимо предварительно добавить в проект в виде ресурсов, воспользовавшись пунктом меню Add -> Existing Item…, а затем подключить с использованием элемента Binary – одинаково для изображений и пиктограмм. Затем изображения можно отобразить на диалоге. Для элемента Bitmap:

```
    <Binary Id="BmpImage" SourceFile="Images\best_robust_big.bmp" />
    <Binary Id="JpgImage" SourceFile="Images\best_secure_big.jpg" />
...
        <Control Id="SimpleImageBmp" Type="Bitmap" Text="BmpImage" X="6" Y="75" Width="220"
Height="50" FixedSize="yes" />
        <Control Id="SunkenImageJpb" Type="Bitmap" Text="JpgImage" X="230" Y="50" Width="131"
Height="87" Sunken="yes" />
```

Идентификатор отображаемого ресурса помещается в атрибут Text. В приведенном выше примере выше описаны два изображения, отличающиеся типом загружаемого ресурса. Кроме того, для второго атрибут Sunken установлен в yes, что делает его визуально несколько «утопленным». Атрибут FixedSize позволяет сохранить оригинальный размер изображения; при этом оно может поместиться частично или, наоборот, занять только часть отведенного размера, но сохранит пропорции. Результат приведен на рисунке 5.2.

Для элемента Icon применимо несколько большее число параметров. Идентификатор ресурса также помещается в атрибут Text. Если файл пиктограммы содержит несколько изображений различного размера, необходимо также указать используемый размер в атрибуте IconSize, принимающем значения 16, 32 и 48, в противном случае изображение показано не будет. Если же файл содержит единственное изображение, значение IconSize можно не задавать.

```
    <Binary Id="HelpIcon" SourceFile="Images\Symbol-Help.ico" />
...
```

```
        <Control Id="Fixed16Icon" Type="Icon" Text="HelpIcon" X="6" Y="145" Width="48"
Height="48" IconSize="16" FixedSize="yes" />
        <Control Id="Fixed32Icon" Type="Icon" Text="HelpIcon" X="76" Y="145" Width="48"
Height="48" IconSize="32" FixedSize="yes" />
        <Control Id="Fixed48Icon" Type="Icon" Text="HelpIcon" X="146" Y="145" Width="48"
Height="48" IconSize="48" FixedSize="yes" />
        <Control Id="Scaled16Icon" Type="Icon" Text="HelpIcon" X="216" Y="145" Width="48"
Height="48" IconSize="16" />
        <Control Id="Scaled32Icon" Type="Icon" Text="HelpIcon" X="286" Y="145" Width="48"
Height="48" IconSize="32" />
        <Control Id="Scaled48Icon" Type="Icon" Text="HelpIcon" X="356" Y="145" Width="48"
Height="48" IconSize="48" />
```

Важен атрибут FixedSize. Посмотрим на пример и на рисунок 5.2. Загруженный файл пиктограммы содержит в себе изображения различного размера. Первые три записи последовательно используют изображения размером 16x16, 32x32 и 48x48 пикселей, при этом FixedSize установлен в yes. При этом размер контейнера игнорируется и пиктограммы выводятся в реальном размере. Три последние записи отличаются тем, что для них FixedSize не указан, принимая значение по умолчанию no, что заставляет их масштабироваться по размеру контейнера (48x48 пикселей). Как видно из примера, при установке FixedSize в yes изображение выглядит приятнее. Все остальные атрибуты, кроме описанных выше, для элементов Bitmap и Icon значения не имеют.

Line и GroupBox предназначены для рисования горизонтальной линии и прямоугольной рамки соответственно. Для них задаются координаты и размеры, причем значение высоты (Height) для линии игнорируется, хотя и должно быть указано. Другие линии, кроме горизонтальных, отсутствуют.

```
        <Control Id="SimpleLine" Type="Line" X="6" Y="205" Width="450" Height="0" />
```

Для элемента GroupBox следует корректно задать высоту и ширину. Значение атрибута Text, если его указать, будет использовано в качестве заголовка.

```
        <Control Id="SimpleGroupBox" Type="GroupBox" X="6" Y="235" Width="70" Height="80" />
        <Control Id="HeaderedGroupBox" Type="GroupBox" X="82" Y="235" Width="70" Height="80"
Text="Заголовок" />
```

Пример использования этих двух элементов управления также на рисунке 5.2.

Гиперссылка – Hyperlink – позволяет по щелчку на ней открыть в браузере страницу с указанным адресом. Адрес помещается в дочерний элемент Text, как показано на примере.

```
        <Control Id="HyperlinkCtrl" Height="20" Width="100" Type="Hyperlink" X="20" Y="332" >
          <Text><![CDATA[<a href="http://www.techdays.ru">www.TechDays.ru</a>]]></Text>
        </Control>
```

Поддержка Hyperlink появилась в Windows Installer 5, поэтому элемент управления работает только в Windows 7 и Windows Server 2008 R2. В ранних версиях Windows вызов диалога, содержащего этот элемент управления, выдаст ошибку (ошибка 2885). Если ваша программа устанавливается только для этих операционных систем, то можно использовать данный элемент управления без каких-либо ограничений. Иначе придется создать два диалога, отличающихся

только наличием гиперссылки на одном из них и выполнять переход в зависимости от значения переменной VersionMsi, как показано ниже.

```
        <Control Id="MsiVersionTransitionButton" Type="PushButton" X="330" Y="180" Width="56"
Height="17" Text="Открыть диалог" TabSkip="no" Sunken="no">
          <Publish Event="NewDialog" Value="VersionMsiGT5Dlg">VersionMsi >= "5.00"</Publish>
          <Publish Event="NewDialog" Value="VersionMsiLT5Dlg">VersionMsi &lt; "5.00"</Publish>
        </Control>
```

Элемент Text является, пожалуй, наиболее востребованным из всех, он встречается практически везде. Следует отметить, что Text – единственный элемент с поддержкой прозрачного фона, это позволяет размещать его поверх изображений, устанавливая в yes значение атрибута Transparent.

```
        <Control Id="SimpleText" Type="Text" Text="Текст" X="180" Y="220" Height="15"
Width="300" />
```

Кроме того, для данного элемента имеют смысл атрибуты NoPrefix и NoWrap. Атрибут NoPrefix позволяет отображать символ & в надписи, в противном случае следующий за амперсандом символ будет отображаться подчеркнутым. Установленный в yes атрибут NoWrap позволяет отключить перенос текста в случаях, когда он не умещается на одной строке.

И, наконец, ScrollableText. Он выводить отображать форматированный текст в формате RTF, при необходимости отображая вертикальную полосу прокрутки. Данную полосу можно расположить слева, установив в yes атрибут LeftScroll. Файл содержимого подключается к проекту с использованием элемента Binary, а затем выводится на компонент с помощью дочернего элемента Text. Есть возможность не использовать отдельный RTF-файл, а сохранить его в виде строки текста, однако в этом случае теряется удобство изменения его содержимого.

```
    <Binary Id="SampleDocumentRtf" SourceFile="SampleDocument.rtf" />
...
        <Control Id="SimpleScrollableText" Type="ScrollableText" X="180" Y="235" Height="80"
Width="130" Sunken="yes">
          <Text SourceFile="SampleDocument.rtf" />
        </Control>
        <Control Id="LeftScrollableText" Type="ScrollableText" X="320" Y="235" Height="80"
Width="130" Sunken="no" LeftScroll="yes">
          <Text SourceFile="SampleDocument.rtf" />
        </Control>
```

Все описанные выше элементы управления изображены на рисунке 5.2.

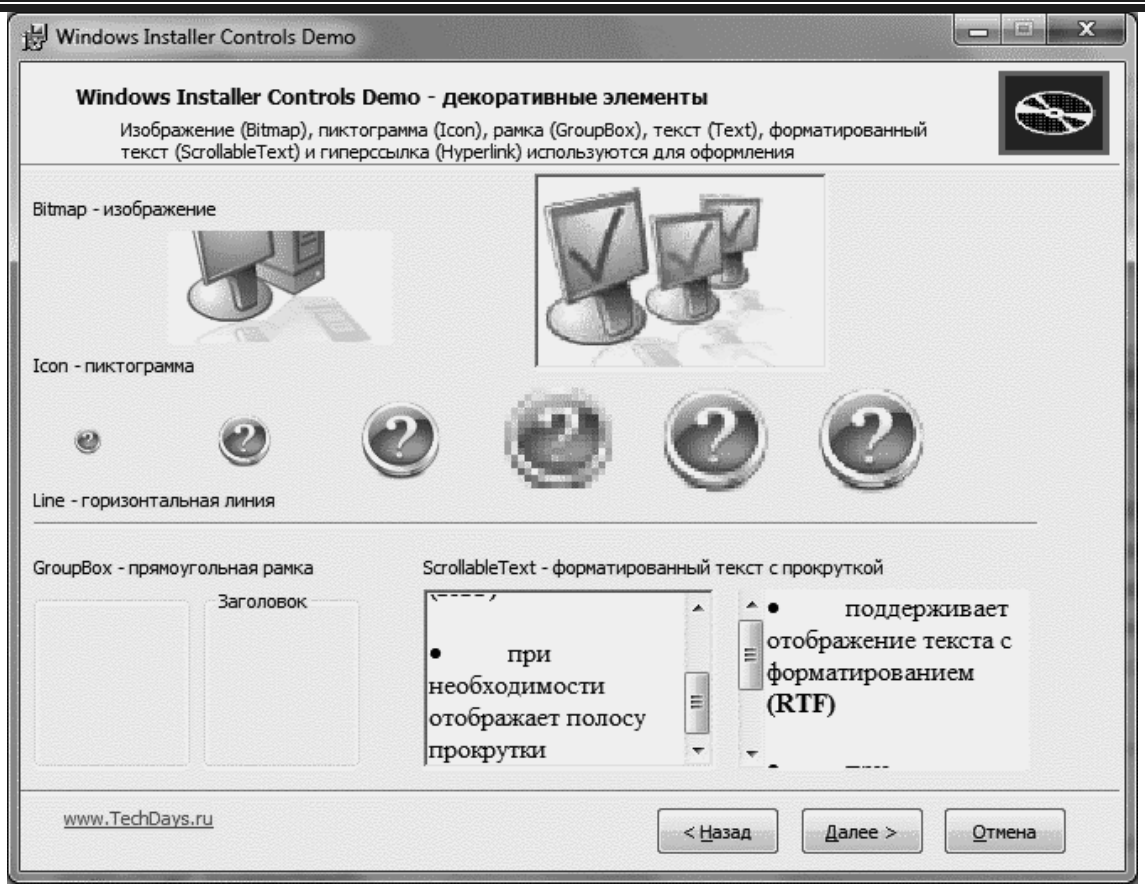

Рисунок 5.2 Декоративные элементы оформления.

Кнопки и переключатели (CheckBox, PushButton, RadioButtonGroup)

Различные виды кнопок являются основным способом взаимодействия с пользователем. Элемент CheckBox принимает одно из двух доступных состояний, группа переключателей позволяет выбрать один вариант из списка, кнопки генерируют события.

CheckBox. Переключатель, способный принимать два состояния. Являясь активным элементом управления, он должен быть связан со свойством, идентификатор которого помещается в атрибут Property. Если свойство на момент отображения диалога инициализировано любым значением, он отобразится включенным. Также необходимо указать значение атрибута CheckBoxValue – именно этим значением будет инициализироваться свойство при включенном переключателе. Если значение для CheckBoxValue не установлено, щелчок мышью на элементе управления не будет вызывать смены его состояния. При снятии флажка связанное свойство удаляется.

Переключатель может зависеть от состояния другого переключателя. В примере ниже это элемент LinkedCheckBox. Для него достаточно установить атрибут CheckBoxPropertyRef в идентификатор принадлежащего другому переключателю, атрибуты Property и CheckBoxValue при этом не используются. Существование «опорного» переключателя является обязательным и контролируется компилятором, но ничто не мешает этим элементам находиться на разных диалогах.

И, наконец, переключатель может отображаться в виде кнопки, для чего в yes устанавливается атрибут PushLike. Установленное свойство отображает кнопку нажатой, отсутствующее – отжатой. В примере это элемент PushLikeCheckBox.

```
    <Property Id="LicenseAccepted" Value="1" />

        <Control Id="LicenseAcceptedCheckBox" Type="CheckBox" X="6" Y="80" Width="100"
Height="12" Property="LicenseAccepted" Text="Я согласен" TabSkip="no" CheckBoxValue="1" />
        <Control Id="LinkedCheckBox" Type="CheckBox" X="130" Y="80" Width="100" Height="12"
Text="Связанный CheckBox" TabSkip="no" CheckBoxPropertyRef="LicenseAccepted" />
        <Control Id="PushLikeCheckBox" Type="CheckBox" X="260" Y="80" Width="100" Height="17"
Text="CheckBox в виде кнопки" TabSkip="no" Property="CheckBoxExtProperty" PushLike="yes"
CheckBoxValue="Checked" />
```

Приведенные выше примеры иллюстрируются рисунком 5.3.

Следует отметить, что фон текста переключателя является непрозрачным. Располагаясь поверх фонового изображения, он приобретает не слишком привлекательный внешний вид. Выход из этой ситуации достаточно предсказуем: для отображения подписи достаточно использовать элемент Text с прозрачным фоном. Размер переключателя при этом следует задавать равным 10x10 пикселей.

PushButton. Пожалуй, кнопка является одним из наиболее часто применяемых элементов управления. Пример ее использования найти не сложно, но не все возможности являются очевидными. Первый пример – элемент SimpleButton – обычная кнопка с содержимым в виде текста. Каждая кнопка при нажатии должна публиковать хотя бы одно событие или устанавливать значение переменной. Генерация событий описывается ниже в этой главе, в этом же примере нажатие на каждую из кнопок устанавливает в 1, 2 или 3 значение переменной ButtonProperty.

Вторая кнопка, ShiedButton, имеет специальное поведение, связанное с настройками системы. Включаемое с помощью атрибута ElevationShield, оно связано с User Account Control (UAC) и имеет смысл только в операционных системах Windows Vista, Windows Server 2008 и старше. Если программа установки запущена без прав администратора – а это поведение по умолчанию – на данной кнопке будет отображена пиктограмма в виде щита, свидетельствующая о запуске операции, требующей повышения полномочий. Если же программа уже запущена с правами администратора, например, с использованием инициирующего загрузчика (Bootstrapper), то пиктограмма на кнопке отображена не будет.

И, наконец, кнопка может вместо текста отображать изображение или пиктограмму. Для изображения следует установить в yes атрибут Bitmap, а в атрибут Text поместить идентификатор изображения, подключенного с использованием элемента Binary. Подключение изображения выполняется так же, как и для описанного выше элемента Bitmap. По умолчанию изображение будет растягиваться на весь элемент управления. Чтобы сохранить оригинальный размер, следует установить в yes атрибут FixedSize.

Для отображения на кнопке пиктограммы следует установить в yes атрибут Icon, в Text поместить идентификатор двоичного ресурса с пиктограммой, кроме того, необходимо аналогично элементу Icon установить значения атрибутов IconSize и FixedSize.

```
        <Control Id="SimpleButton" Type="PushButton" Text="OK" X="6" Y="270" Height="20"
Width="60" >
          <Publish Property="ButtonProperty" Value="1" />
        </Control>
```

```
        <Control Id="ShieldButton" Type="PushButton" Text="Shield" X="72" Y="270" Height="20"
Width="60" ElevationShield="yes">
          <Publish Property="ButtonProperty" Value="2" />
        </Control>
        <Control Id="ImageButton" Type="PushButton" Text="HelpIcon" X="138" Y="270"
Height="20" Width="60" Icon="yes" IconSize="16" FixedSize="yes">
          <Publish Property="ButtonProperty" Value="3" />
        </Control>
```

Изображения описанных в примере выше кнопок приведены на рисунке 5.3.

Компонент RadioButtonGroup представляет группу переключателей, причем в каждый момент времени может быть выбран только один. Свойство, указанное в атрибуте Property элемента Control, используется только для установки первичного состояния переключателей. Действия пользователя отображаются на значении свойства, указанного в атрибуте Property дочернего для Control элемента RadioButtonGroup. Это может быть как одно свойство, так и различные. Переключатели описываются дочерними для RadionButtonGroup элементами RadioButton, причем для каждого из них должен быть установлен атрибута Value – это значение, присваиваемое связанному свойству при выборе. Элементы внутри группы позиционируются вручную, поэтому можно разместить их произвольно. Следует обратить внимание на то, чтобы все элементы RadioButton имели уникальное в пределах списка значение атрибута Value, а связанное с элементом Control свойство – равное одному из перечисленных в этом списке начальное значение. Выполнение этих условий контролируется при сборке.

```
    <Property Id="IAgree" Value="Yes" />
...
        <Control Id="SimpleRadioButtonGroup" Type="RadioButtonGroup" X="6" Y="130" Width="120"
Height="73" Property="IAgree" >
          <RadioButtonGroup Property="IAgree">
            <RadioButton Height="15" Text="Я &amp;amp;принимаю условия" Value="Yes" Width="120"
X="0" Y="0" />
            <RadioButton Height="15" Text="Я н&amp;amp;е принимаю условия" Value="No" Width="120"
X="0" Y="18" />
          </RadioButtonGroup>
        </Control>
```

Группа переключателей может иметь прямоугольную рамку, для чего достаточно установить в yes атрибут HasBorder. Кроме того, при наличии рамки можно задать текст ее заголовка, просто указав его в атрибуте Text. Элементы списка могут выглядеть не как переключатели, а как кнопки, для чего достаточно установить в yes атрибут PushLike элемента Control.

```
    <Property Id="Choice2" Value="1" />
...
        <Control Id="HeaderedRadioButtonGroup" Type="RadioButtonGroup" X="260" Y="130"
Width="120" Height="73" Property="Choice2" HasBorder="yes" Text="Заголовок" PushLike="yes">
          <RadioButtonGroup Property="Choice2">
            <RadioButton Height="15" Text="Выбор 1" Value="1" Width="100" X="4" Y="10" />
            <RadioButton Height="15" Text="Выбор 2" Value="2" Width="100" X="4" Y="25" />
```

```
            <RadioButton Height="15" Text="Выбор 3" Value="3" Width="100" X="4" Y="40" />
          </RadioButtonGroup>
        </Control>
```

И, наконец, вместо текста элементы списка могут использовать изображения. Для этого устанавливаются значения атрибутов Bitmap – для изображения, или Icon, IconSize и FixedSize – для пиктограммы. RadioButtonGroup ведет себя при этом идентично PushButton, а в качестве примера можно использовать элементы управления Bitmap и Icon. Идентификаторы ресурсов устанавливаются в поле Text соответствующих элементов RadioButton, при этом каждый из них может иметь свое собственное изображение.

```
    <Property Id="Choice3" Value="2" />
...
        <Control Id="ImageRadioButtonGroup" Type="RadioButtonGroup" X="130" Y="130" Width="69"
Height="73" Property="Choice3" PushLike="yes" HasBorder="yes" Icon="yes" IconSize="32"
FixedSize="yes" >
          <RadioButtonGroup Property="Choice3">
            <RadioButton Height="32" Text="HelpIcon" Value="1" Width="32" X="3" Y="7" />
            <RadioButton Height="32" Text="HelpIcon " Value="2" Width="32" X="35" Y="7" />
            <RadioButton Height="32" Text="HelpIcon " Value="3" Width="32" X="3" Y="39" />
            <RadioButton Height="32" Text="HelpIcon " Value="4" Width="32" X="35" Y="39" />
          </RadioButtonGroup>
        </Control>
```

Как и для остальных элементов управления, описанных в этом разделе, приведенные примеры демонстрируются на рисунке 5.3.

Рисунок 5.3 Кнопки и переключатели.

Редакторы (Edit, MaskedEdit, PathEdit)

Редакторы используются для ввода пользователями различных значений. В Windows Installer они представлены элементами управления Edit, MaskedEdit и PathEdit.

Начнем с рассмотрения Edit, как наиболее часто используемого. Примеры для разных вариантов этого компонента я размещу вместе, так как отличаются они несущественно. Первый случай – простое поле ввода, не выделяется ничем особенным, достаточно указать тип компонента и идентификатор связанного свойства. При этом если связанное свойство описано, его значение будет использовано в качестве начального, в противном случае оно будет создано при вводе значения в поле. Мы можем, установив в yes атрибут Integer, рассматривать передаваемое в элемент управления значение как целочисленное, но в случае несоответствия типов пользователь увидит некрасивое сообщение об ошибке, поэтому для этих целей лучше использовать MaskedEdit, о котором несколько ниже.

```
    <Property Id="SimpleEditProperty" Value="Значение" />
    <Property Id="PasswordEditProperty" Value="pwd" Hidden="yes" />
    <Property Id="MultilineEditProperty" Value="Первая строка
Вторая строка" />
…
        <Control Id="SimpleEdit" Type="Edit" X="6" Y="70" Width="200" Height="16"
Property="SimpleEditProperty" Sunken="no" />
```

```
        <Control Id="PasswordEdit" Type="Edit" X="6" Y="90" Width="200" Height="16"
Property="PasswordEditProperty" Password="yes" Sunken="no" />
        <Control Id="MultilineEdit" Type="Edit" X="6" Y="110" Width="200" Height="32"
Property="MultilineEditProperty" Multiline="yes" LeftScroll="yes" />
```

Второй вариант добавляет установленный в yes атрибут Password, отображающий вводимый текст в виде звездочек. Собственно, может использоваться для ввода паролей. При этом значение связанного свойства не помещается в отладочный вывод, но я все равно устанавливаю в yes атрибут Hidden для данного свойства.

Установив в yes атрибут Multiline, мы получим многострочное поле ввода. Переход на новую строчку осуществляется только нажатием сочетания клавиш Ctrl + Enter, что может смутить пользователя. При необходимости элемент управления отображает вертикальную полосу прокрутки, которую можно передвинуть влево, просто установив в yes атрибут LeftScroll. Все варианты компонента Edit приведены на рисунке 5.4.

Элемент управления MaskedEdit позволяет выполнить ввод значения в соответствии с определенной маской. Маска ввода задается в свойстве, которое передается в атрибут Text в квадратных скобках. В справочной системе написано достаточно много по этому вопросу, но для решения основной части задач достаточно следующего:

- значение должно заключаться в угловые скобки;
- цифры могут представляться символами «#», «%» или «@»;
- цифра, буква или знак представляется любым из символов «&», «^», или «?»;
- разделять поля можно пробелом, дефисом или знаком подчеркивания.

Наличие различных символов в каждой категории позволяет различать их, когда производится сложная обработка значения, например, из пользовательской операции.

```
    <Property Id="MaskedEditTemplate" Value="&lt;###-^^^-??&gt;" />
    <Property Id="MaskedEditTemplate2"><![CDATA[<###-^^^-??>]]></Property>
...
    <Property Id="MaskedEditTemplate" Value="&lt;###-^^^-??&gt;" />
        <Control Id="SimpleMaskedEdit" Type="MaskedEdit" X="6" Y="185" Width="100" Height="16"
Property="MaskedEditProperty" Text="[MaskedEditTemplate]" />
```

После завершения ввода связанному свойству присваивается значение, включающее символы-разделители.

Последний элемент – PathEdit – позволяет контролировать вводимые значения на наличие недопустимых для пути символов. Начальное значение связанной переменной должно быть присвоено обязательно. Если оно отсутствует или некорректно задано, при переходе к содержащему элемент управлению диалогу будет отображено сообщение об ошибке с завершением работы пакета.

```
    <Property Id="PathEditProperty" Value="c:\temp\shared\" />
...
        <Control Id="SimplePathEdit" Type="PathEdit" X="6" Y="235" Width="350" Height="20"
Property="PathEditProperty" />
```

Введенное значение проверяется на допустимость при потере элементом фокуса ввода. Если оно является некорректным, выводится диалог с уведомлением об ошибке и значение сбрасывается на предыдущее.

Все перечисленные выше элементы управления, используемые для ввода значений, приведены на рисунке 5.4.

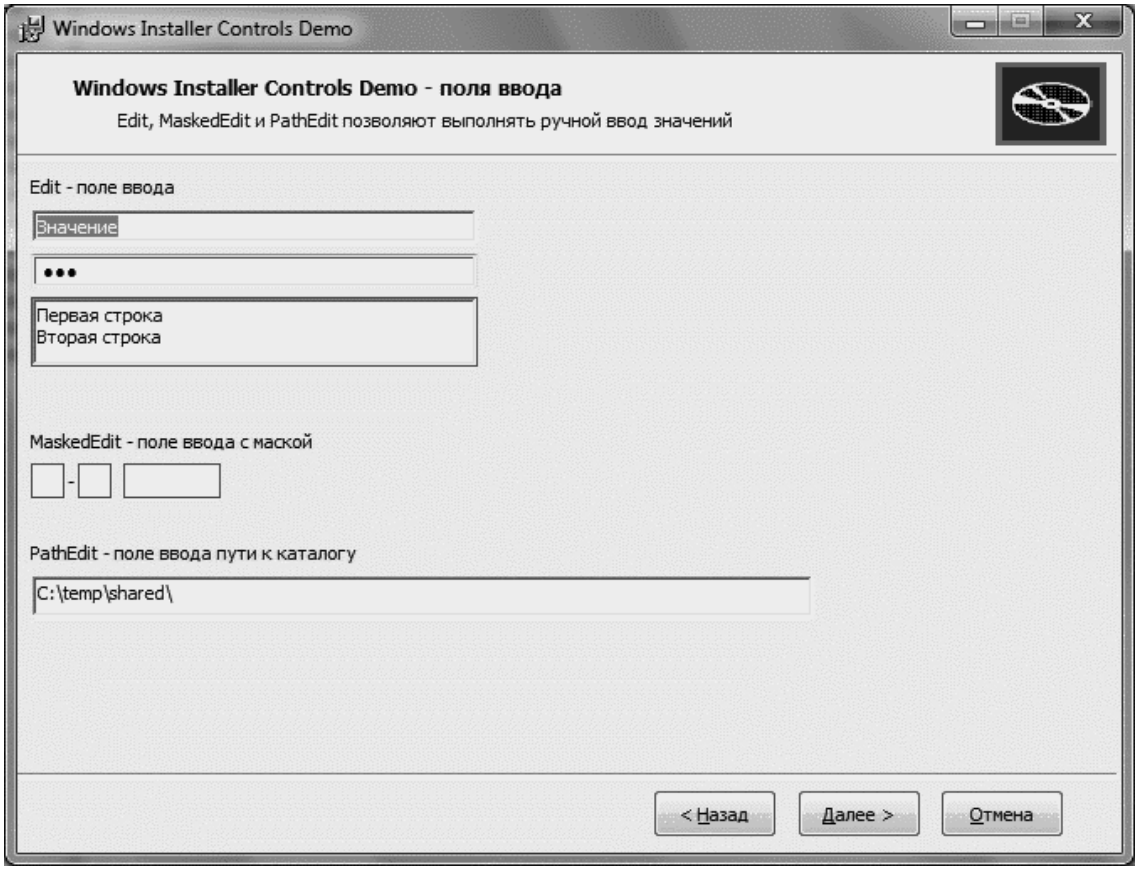

Рисунок 5.4 Редакторы.

Списки (ComboBox, ListBox, ListView)

Элементы для работы со списками представлены в Windows Installer тремя компонентами. Это раскрывающийся список – ComboBox и два обычных – ListBox и ListView, последний из которых отличается только возможностью отображения пиктограмм. Все три компонента поддерживают атрибут Sorted, определяющий наличие сортировки. Он работает несколько специфически: если значение атрибута не указано или установлено в no, элементы будут автоматически расположены в алфавитном порядке. Если же установить Sorted в yes, элементы будут выведены в том порядке, в котором описаны. Следует аккуратно использовать сортировку для элемента ListView, так как при этом в нем иногда не отображаются пиктограммы.

Начнем с рассмотрения элемента управления ComboBox. Пример, представленный ниже, описывает список, содержащий три элемента. Кроме того, по умолчанию поле значения доступно для редактирования, что позволяет инициализировать список любым значением. Начальное значение элемента управления присваивается равным значению переменной, идентификатор которой указан в атрибуте Property элемента Control. Если переменная не инициализирована, при

открытии диалога поле выбора будет пустым. При выборе одного из вариантов в поле выбора отображается значение из атрибута Value соответствующего элемента ListItem, оно же присваивается свойству с идентификатором, указанным в атрибуте Property элемента ComboBox.

```
    <Property Id="ComboBoxProperty" Value="ComboBox - начальное значение" />
...
        <Control Id="SimpleComboBox" Type="ComboBox" X="6" Y="70" Width="330" Height="100"
Property="ComboBoxProperty" TabSkip="yes" Sunken="yes" >
          <ComboBox Property="ComboBoxProperty">
            <ListItem Text="Вариант 2" Value="2" />
            <ListItem Text="Вариант 1" Value="1" />
            <ListItem Text="Вариант 3" Value="3" />
          </ComboBox>
        </Control>
```

Возможность ввода значения наряду с выбором из списка достаточно интересна, но иногда является излишней. Если такая необходимость возникла, следует обратить внимание на атрибут ComboList. Будучи установлен в yes, он не только отключает возможность ввода значения вручную, но и несколько изменяет поведение компонента. Так, начальное значение устанавливается только в том случае, если оно совпадает со значением атрибута Value одно из вложенных элементов ListItem. Кроме того, при выборе значения из списка в поле выбора связанному свойству присваивается значение атрибута Value, как в предыдущем случае, а вот в поле выбора отображается значение атрибута Text соответствующего элемента ListItem. В последнем случае я установил в yes атрибут Sorted, поэтому элементы в списке отобразятся в том порядке, в котором я привел их здесь.

```
    <Property Id="ComboBoxProperty2" Value="2" />
...
        <Control Id="NoEditComboBox" Type="ComboBox" X="6" Y="90" Width="330" Height="100"
Property="ComboBoxProperty2" TabSkip="yes" Sunken="yes" ComboList="yes" Sorted="yes">
          <ComboBox Property="ComboBoxProperty2">
            <ListItem Text="Вариант 2" Value="2" />
            <ListItem Text="Вариант 1" Value="1" />
            <ListItem Text="Вариант 3" Value="3" />
          </ComboBox>
        </Control>
```

Примеры использования списков, в том числе и раскрывающихся, приведены на рисунке 5.5.

Элементы ListBox и ListView очень похожи. Как уже описывалось ранее, отличие заключается только в том, что ListView обладает способностью отображать элементы списка вместе с пиктограммами. Сначала добавим в наш диалог простой ListBox. Как видно, он практически идентичен элементу ComboBox, даже несколько проще. Атрибут Property элемента Control определяет свойство с начальным значением; атрибут Property элемента ListBox – свойство, куда помещается результат выбора – значение атрибута Value выбранного элемента ListItem.

```
        <Control Id="SimpleListBox" Type="ListBox" X="6" Y="125" Width="330" Height="70"
Property="ListBoxProperty" >
          <ListBox Property="ListBoxProperty">
```

```
            <ListItem Text="Вариант 1" Value="1" />
            <ListItem Text="Вариант 2" Value="2" />
            <ListItem Text="Вариант 3" Value="3" />
          </ListBox>
        </Control>
```

И, наконец, компонент ListView с поддержкой пиктограмм. В примере на рисунке 5.5 я в качестве примера использовал пути к стандартным каталогам, но в тексте, для краткости, оставил только несколько значений. Здесь же видно, как можно использовать форматированные строки: в квадратных скобках указаны идентификаторы стандартных свойств, в процессе выполнения заменяемые их значениями. Идентификаторы пиктограмм помещаются в атрибут Icon каждого из элементов ListItem. Несмотря на то, что компонент ListBox также использует вложенные элементы ListItem, указание значения атрибута Icon для них вызовет ошибку компиляции.

```
        <Control Id="SimpleListView" Type="ListView" X="6" Y="210" Width="440" Height="90"
Property="PathId" Sorted="no">
          <ListView Property="Path">
            <ListItem Text="WindowsFolder: [WindowsFolder]" Value="23" Icon="HelpIcon" />
            <ListItem Text="AdminToolsFolder: [AdminToolsFolder]" Value="1" Icon="HelpIcon" />
            <ListItem Text="WindowsVolume: [WindowsVolume]" Value="24" Icon="HelpIcon" />
          </ListView>
        </Control>
```

Элемент ListView требует внимательного к себе отношения. Так, задавая пиктограммы, нет возможности пропустить один из элементов – в этом случае пиктограммы всех элементов просто будут сдвинуты вверх, а пустые значения останутся внизу списка. Для создания строк без пиктограмм вам потребуются «пустые» значки. Также следует внимательно отнестись к установке атрибута Sorted. В некоторых случаях значение yes может привести к тому, что пиктограммы вообще не будут отображаться.

В любом случае, компоненты для работы со списками очень важны для создания развитого интерфейса пользователя. Описанные выше примеры приведены на рисунке 5.5.

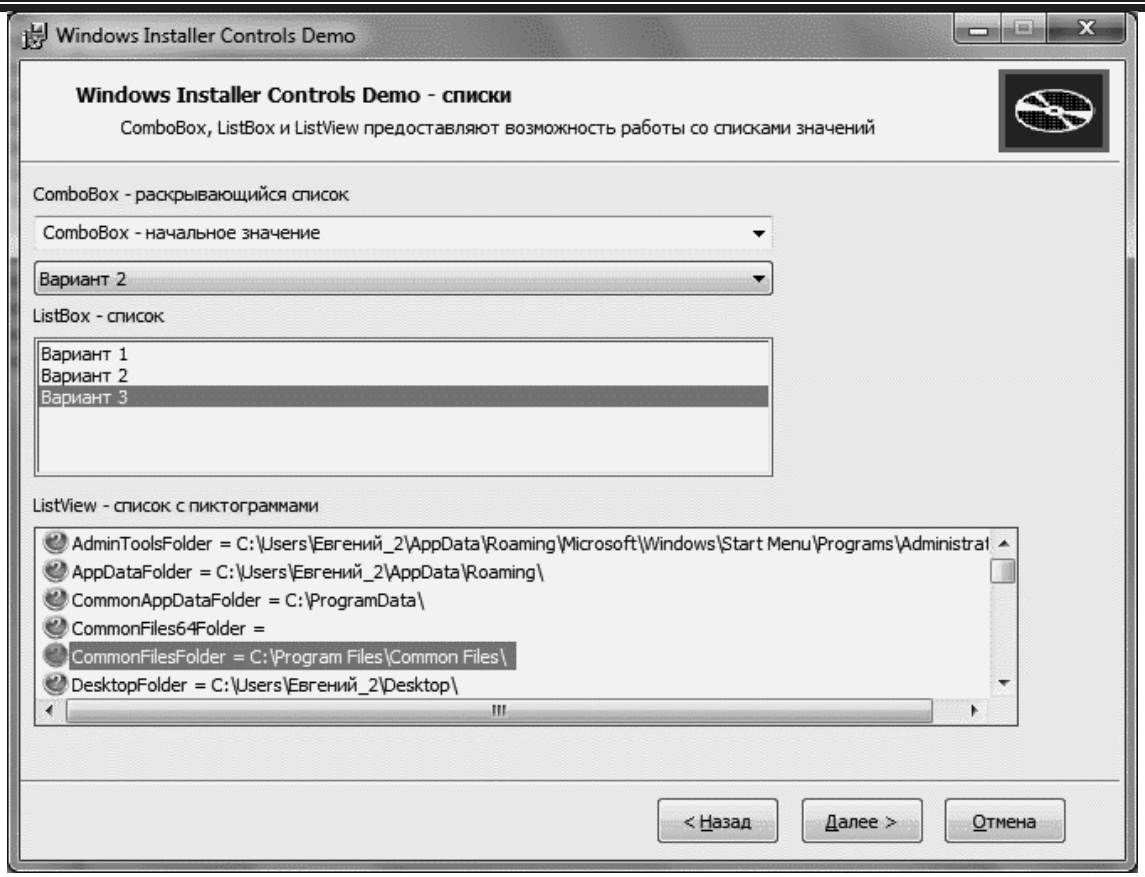

Рисунок 5.5 Компоненты для работы со списками.

Работа с каталогами (VolumeSelectCombo, DirectoryCombo, DirectoryList)

Для просмотра перечня дисков и каталогов и выбора используются три элемента управления: VolumeSelectCombo представляет собой раскрывающийся список доступных логических дисков, DirectoryCombo в раскрывающемся списке представляет путь к текущему каталогу с возможностью выбора логического диска; DirectoryList отображает все дочерние каталоги в текущем. Все эти компоненты следует привязывать к единственному свойству, так как они дополняют друг друга.

VolumeSelectCombo и DirectoryCombo очень похожи и используют один набор атрибутов. Я привязываю их к свойству INSTALLLOCATION, определяющему каталог для установки приложения. В вашем приложении идентификатор каталога для установки может отличаться. Атрибуты Fixed, CDROM, RAMDisk, Remote, Removable, Floppy определяют перечень доступных для выбора дисков по типам. Они включают (yes) или отключают (no) отображение локальных жестких дисков, CD/DVD-приводов, RAM-, сетевых, съемных дисков и дискет.

```
        <Control Id="VolumeSelectCombo" Type="VolumeSelectCombo" Property="INSTALLLOCATION"
X="6" Y="70" Width="200" Height="20" TabSkip="yes" Fixed="yes" CDROM="yes" RAMDisk="yes"
Remote="yes" Removable="yes" Floppy="yes" />

        <Control Id="DirectoryComboCommentText" Type="Text" Text="DirectoryCombo -
раскрывающийся список, отображающий текущий путь в виде дерева" X="6" Y="110" Height="15"
Width="400" />
```

```
        <Control Id="DirectoryCombo" Type="DirectoryCombo" X="6" Y="125" Width="200"
Height="20" Property="INSTALLLOCATION" TabSkip="no" Fixed="yes" CDROM="yes" RAMDisk="yes"
Remote="yes" Removable="yes" Floppy="yes" >
          <Subscribe Event="IgnoreChange" Attribute="IgnoreChange" />
        </Control>
```

Целесообразно на одном диалоге не использовать одновременно VolumeSelectCombo и DirectoryCombo – это связано с тем, что VolumeSelectCombo не изменяет выбранный диск при его смене в DirectoryCombo. Хотя обратная привязка работает без проблем, это приводит к несколько несогласованному поведению. Подписка элемента DirectoryCombo на событие IgnoreChange, публикуемое элементов DirectoryList, позволяет получить их согласованную работу. Подробнее данный момент рассмотрен ниже в разделе «Механизм событий».

Элемент управления DirectoryList привязывается к тому же свойству INSTALLLOCATION. Он отображает перечень дочерних каталогов в текущем, позволяя переходить к ним по двойному щелчку мышью.

```
        <Control Id="DirectoryList" Type="DirectoryList" X="6" Y="175" Width="400"
Height="100" Property="INSTALLLOCATION" Text="List of directories" TabSkip="no" />
```

Данный элемент неявно подписывается на события DirectoryListUp, DirectoryListOpen и DirectoryListNew – достаточно разместить на этом же диалоге кнопки, публикующие эти события, чтобы оживить интерфейс. Первое событие приводит к переходу в родительский каталог, второе открывает выбранную в текущий момент папку, последнее – создает новый каталог с возможностью задания для него имени. Описанные элементы управления приведены на рисунке 5.6.

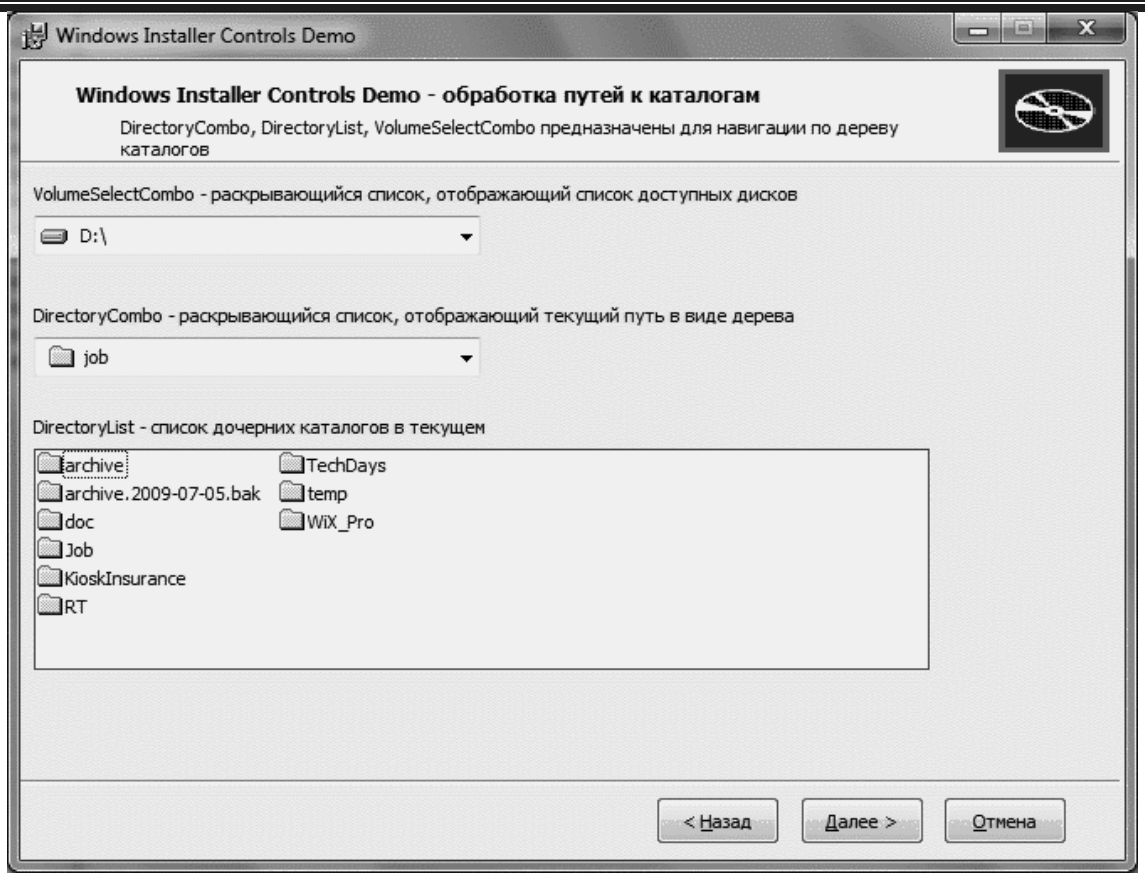

Рисунок 5.6 Компоненты для работы со списком каталогов.

Наборы компонентов и связанные задачи (SelectionTree, VolumeCostList)

Компонент SelectionTree является неотъемлемой частью технологии Windows Installer и не применим где-либо еще. Он отображает наборы компонентов в виде древовидного списка, позволяя указать различные варианты их установки. Компонент VolumeCostList предназначен для отображения доступных и необходимых объемов места на носителях и, как правило, отображается поблизости от выбора наборов. Следует обратить внимание на то, что объемы пересчитываются в момент инициализации диалога, поэтому проще всего разнести данные компоненты по разным окнам.

```
        <Control Id="SelectionTree" Type="SelectionTree" X="6" Y="70" Width="240" Height="100"
Text="Компоненты приложения" Property="NotAProperty" />
```

Для SelectionTree значимыми являются только свойства, определяющие размеры. Значение атрибута Property установить необходимо, но привязка к свойству в случае данного компонента не используется – в нем достаточно указать любое строковое значение. Значение атрибута Text может быть использовано средствами, упрощающими работу людям со слабым зрением.

Элемент SelectionTree обрабатывает события AddLocal, AddSource и Remove, что позволяет дублировать часть функционала нажатием кнопок. Первое событие позволяет выбрать набор для установки локально, второе – настраивает его для запуска с установочного носителя. Последнее отменяет установку указанного набора. В примере ниже кнопка с идентификатором AddLocalFeatureSet1 настраивает для установки на диск набор FeatureSet1, AddSourceFeatureSet2 – набор FeatureSet2 для установки с носителя, а кнопка RemoveAll отменяет установку всех наборов.

```
        <Control Id="AddLocalFeatureSet1" Type="PushButton" Text="Набор 1 - локально" X="6"
Y="175" Width="112" Height="22" TabSkip="no">
          <Publish Event="AddLocal" Value="FeatureSet1" />
        </Control>
        <Control Id="AddSourceFeatureSet2" Type="PushButton" Text="Набор 2 - с источника"
X="120" Y="175" Width="112" Height="22" TabSkip="no">
          <Publish Event="AddSource" Value="FeatureSet2" />
        </Control>
        <Control Id="RemoveAll" Type="PushButton" Text="Отменить все" X="250" Y="175"
Height="22" Width="112">
          <Publish Event="Remove" Value="ALL" />
        </Control>
```

Обращу внимание на то, что запуск с источника потребует некоторых дополнительных действий – в противном случае данная опция будет недоступна. Подробно этот процесс описан в главе 3. Кроме того, нажатие на любую из данных кнопок не приведет к немедленной перерисовке элементов управления SelectionTree и VolumeCostList – придется произвести переход между диалоговыми окнами.

Компонент SelectionTree также неявно публикует события SelectionAction, SelectionDescription, SelectionNoItems, SelectionPath, SelectionPathOn и SelectionSize. Они предоставляют дополнительную информацию о выбранном в настоящее время наборе. В приведенном ниже примере два элемента типа Text подписываются на события SelectionDescription и SelectionSize. Соответственно, первый будет отображать значение атрибута Description выбранного в настоящий момент набора, второй – показывать необходимый для установки выбранного набора объем.

```
        <Control Id="ItemDescription" Type="Text" X="250" Y="70" Width="180" Height="50"
Text="Описание выбранного элемента" TabSkip="yes" Sunken="yes">
          <Subscribe Event="SelectionDescription" Attribute="Text" />
        </Control>
        <Control Id="ItemSize" Type="Text" X="250" Y="120" Width="180" Height="50"
Text="Объем, необходимый для установки выбранного элемента" TabSkip="yes" Sunken="yes">
          <Subscribe Event="SelectionSize" Attribute="Text" />
        </Control>
```

Текст, получаемый от события SelectionSize, различен для конкретной ситуации. Образующие его строки задаются элементами UIText со стандартными идентификаторами, начинающимися с подстроки «Sel…», например:

```
      <UIText Id="SelChildCostPos">Для данного компонента требуется [1] на жестком
диске.</UIText>
      <UIText Id="SelParentCostNegPos">Данный компонент освобождает [1] на жестком диске. Для
него выбраны [2] из [3] подкомпонентов. Для подкомпонентов требуется [4] на жестком
диске.</UIText>
```

Полный перечень идентификаторов достаточно длинный и найти его можно в разделе MSDN, посвященном данному элементу управления.

Компонент VolumeCostList используется для отображения места, необходимого для установки выбранных элементов. Интерес здесь представляют атрибуты, управляющие списком видимых дисков: Fixed – локальные жесткие, CDROM – CD/DVD-приводы, RAMDisk – RAM-диски, Remote – сетевые, Removable – съемные. Каждый принимает значения yes или no. Установленный в yes атрибут ShowRollbackCost добавляет в расчет объем, необходимый для выполнения отката установки.

```
        <Control Id="VolumeCostList" Type="VolumeCostList" X="6" Y="220" Width="460"
Height="80" Text="{120}{70}{70}{70}{70}" TabSkip="yes" Sunken="yes" Fixed="yes" CDROM="yes"
RAMDisk="yes" Remote="yes" Removable="yes" ShowRollbackCost="yes" Indirect="yes" />
```

Отдельного внимания заслуживает атрибут Text – здесь он содержит строку формата, определяющую ширину используемых колонок – всего их пять (том, размер диска, доступно, требуется, разница). Чтобы скрыть одну из колонок, установить для нее размер в нулевое значение. Как и для SelectionTree, пересчет объемов производится при создании диалога.

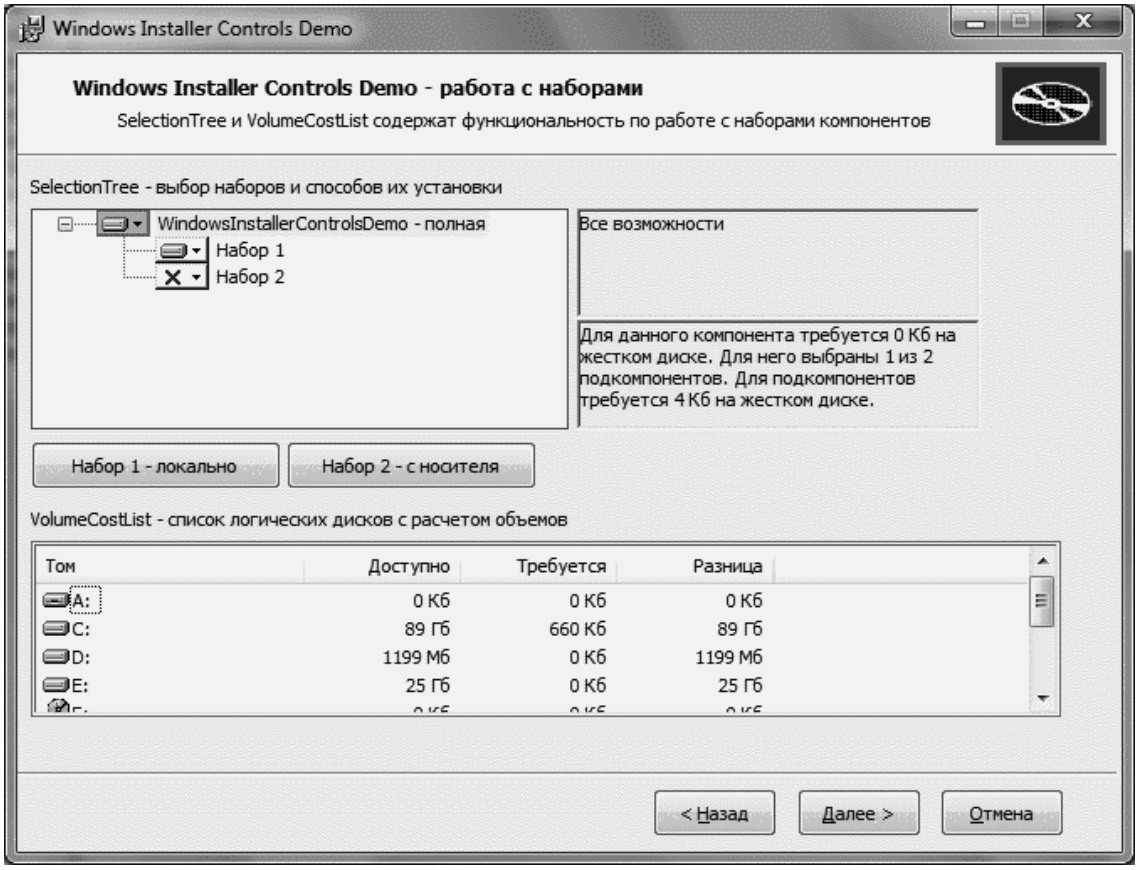

Рисунок 5.7 Компоненты SelectionTree и VolumeCostList.

Элементы процесса установки (Billboard, ProgressBar)

Billboard. Windows Installer позволяет в процессе установки отображать сменяющиеся статические фрагменты интерфейса, для чего предназначен компонент Billboard. Его функционал может быть полезен при подготовке установочных пакетов для крупных продуктов, установка которых занимает значительное время. Следует отметить некоторые ограничения, присущие использованию указанного компонента:

- элемент Billboard связывается с событиями процесса установки и не может быть использован на других диалогах;
- фрагменты выводятся в области внутри диалогового окна. Нет возможности заполнить, например, фон рабочего стола.

Изложенное выше показывает, что Billboard нельзя использовать для смены изображений при переходе между диалогами, а только для «оживления» непосредственно процесса развертывания, как наиболее длительного.

Для использования потребуется наличие элемента управления с типом Billboard, а также непосредственно описание отображаемых фрагментов и их привязки к событиям установки и, необязательно, к установке отдельных наборов.

Перейдем к реализации. Лучше всего добавить в проект несколько отличающихся изображений – так их смена будет выглядеть нагляднее. Подключим изображения.

```
    <Binary Id="Bitmap1" SourceFile="Bitmap1.bmp" />
    <Binary Id="Bitmap2" SourceFile="Bitmap2.bmp" />
```

Теперь внутри тега UI добавим описание фрагментов и привязок. Для каждого элемента BillboardAction в атрибуте Id задается привязка к названию стандартной операции. В нашем случае мы связываем их с операциями InstallValidate и InstallFiles. Подробнее использование операций и их последовательностей описано в главе 6. Внутри каждого из BillboardAction размещается один или несколько элементов Billboard, в свою очередь привязываемые к отдельным наборам, идентификаторы которых указываются в атрибуте Feature. Внутри Billboard может размещаться статическое содержимое: текст, изображения и линии.

```
      <BillboardAction Id="InstallValidate">
        <Billboard Id="Billboard1" Feature="ProductFeature">
          <Control Id="BillboardControl1" Type="Text" X="90" Y="14" Width="264" Height="20"
Text="{\WixUI_Font_Title}Подготовка к установке продукта" />
          <Control Id="BillboardControl2" Type="Line" X="15" Y="34" Width="300" Height="0"
TabSkip="yes" Disabled="yes" />
        </Billboard>
      </BillboardAction>
      <BillboardAction Id="InstallFiles">
        <Billboard Id="Billboard2" Feature="FeatureSet1">
          <Control Id="Billboard2Image" Type="Bitmap" Text="Bitmap1" X="0" Y="0" Width="330"
Height="110" FixedSize="yes" />
          <Control Id="Billboard2Text" Type="Text" X="90" Y="14" Width="240" Height="20"
Text="{\WixUI_Font_Title}Установка RequiredComponents" Transparent="yes" />
          <Control Id="Billboard2Line" Type="Line" X="15" Y="34" Width="300" Height="0"
TabSkip="yes" Disabled="yes" />
        </Billboard>
        <Billboard Id="Billboard3" Feature="FeatureSet2">
          <Control Id="Billboard3Image" Type="Bitmap" Text="Bitmap2" X="0" Y="0" Width="330"
Height="110" FixedSize="yes" />
```

```
          <Control Id="Billboard3Text" Type="Text" X="90" Y="14" Width="240" Height="20"
Text="{\WixUI_Font_Title}Установка HelpFiles" Transparent="yes" />
          <Control Id="Billboard3Line" Type="Line" X="15" Y="34" Width="300" Height="0"
TabSkip="yes" Disabled="yes" />
        </Billboard>
      </BillboardAction>
```

Приведенный выше пример описывает три варианта содержимого: при подготовке к установке (операция InstallValidate) будет отображаться содержимое элемента Billboard1 (надпись и горизонтальная линия), а в процессе копирования файлов (операция InstallFiles) последовательно отобразятся элементы Billboard2 и Billboard3, на каждом из которых размещены изображение, надпись и линия. При этом Billboard1 привязан к корневому набору с идентификатором ProductFeature, а последние два – к наборам FeatureSet1 и FeatureSet2 соответственно. Такое деление позволяет достаточно гибко управлять отображаемым содержимым. Установка программы выполняется достаточно быстро, поэтому, чтобы понаблюдать за процессом смены содержимого, потребуется добавить в тестовый проект несколько достаточно объемных файлов.

И, наконец, на диалог установки добавим сам элемент Billboard, который и будет отображать все содержимое. Следует правильно подбирать размеры элементов управления, выходящие за границы элементы могут не быть отображены.

```
        <Control Id="Billboard1" Type="Billboard" X="20" Y="120" Width="330" Height="110"
TabSkip="yes" Disabled="yes">
          <Subscribe Event="SetProgress" Attribute="Progress" />
        </Control>
```

Еще раз отмечу способ пересчета размеров управления из Installer Units в пиксели: для элемента управления Bitmap размером 330x110 необходимо создать файлы изображений размером 439x146 (коэффициент 4/3).

Диалог установки несколько отличается от других стандартных диалоговых окон, поэтому завершенный пример, в котором используется элемент Billboard, приведен ниже в разделе «Отображение прогресса установки».

Отображение модального диалога

Нередко возникает необходимость создания модального диалогового окна поверх существующего. В Windows Installer такое поведение реализуется с помощью событий SpawnDialog и SpawnWaitDialog. Их отличие от события NewDialog заключается в том, что текущий диалог не уничтожается, а становится неактивным до возврата ему управления.

Первое событие позволяет отобразить ожидающее реакции пользователя окно; это может быть запрос подтверждения важной операции, например, прерывания процесса установки.

```
        <!-- Кнопка на вызывающем диалоге - отобразить модальный диалог -->
        <Control Id="buttonCancelRequest" Type="PushButton" X="236" Y="243" Width="56"
Height="17" Default="yes" Text="Нажми">
          <Publish Event="SpawnDialog" Value="CancelInstallationDlg" Order="2">1</Publish>
        </Control>
```

```
      <!-- Диалог подтверждения выхода -->
      <Dialog Id="CancelInstallationDlg" Width="200" Height="80" Title="Внимание">
        <Control Type="Text" Id="textQuestion" Width="128" Height="17" X="38" Y="15"
Text="Прервать установку программы?" />
        <Control Type="PushButton" Id="buttonOk" Width="69" Height="17" X="22" Y="52"
Text="OK">
          <Publish Event="EndDialog" Value="Exit" />
        </Control>
        <Control Type="PushButton" Id="buttonCancel" Width="70" Height="17" X="109" Y="52"
Text="Отмена">
          <Publish Event="EndDialog" Value="Return" />
        </Control>
      </Dialog>
```

Кнопка buttonCancel публикует событие SpawnDialog, передавая ему значение CancelInstallationDlg. При этом Windows Installer создает диалог с указанным идентификатором и отображает поверх текущего. Вызывающее окно в примере выше не обрабатывает результат, вместо этого кнопки на диалоге CancelInstallationDlg публикуют событие EndDialog с атрибутом return или exit. Его вызов с атрибутом return закрывает текущий диалог; значение exit прерывает процесс установки программы.

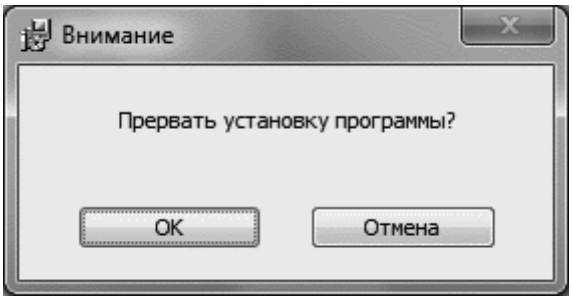

Рисунок 5.8 Диалог подтверждения выхода.

Другое событие – SpawnWaitDialog – позволяет отобразить модальный диалог до изменения значения указанного свойства. К сожалению, нет возможности вызвать данный диалог из интерфейса пользователя одновременно с асинхронным запуском операции – в этом случае вызывающий диалог не получит уведомления об изменении состояния. Это существенно ограничивает область применения данного события.

Механизм событий

Windows Installer предоставляет основанную на событиях модель взаимодействия элементов управления. Одна из сторон публикует событие, а другая (или другие), выступая в качестве подписчика, обрабатывает его. Вместе с событием может передаваться один аргумент, тип которого зависит от типа события.

Как правило, для конкретного события на разработчика возлагается или только его публикация, или только обработка. Другая часть действия при этом неявно выполняется службой (установщиком) или элементом управления. Из существующих элементов управления три –

PushButton, CheckBox и SelectionTree способны при взаимодействии с ними пользователя порождать события. При этом элемент может последовательно сгенерировать несколько событий, за некоторыми исключениями.

Для публикации события используется элемент Publish, который может находиться внутри элемента UI или внутри элемента управления. В первом случае необходимо дополнительно установить значения атрибутов Dialog и Control – идентификаторы диалога и располагающегося на нем элемента управления соответственно:

```
    <UI>
      <Publish Dialog="CustomDialogID" Control="ButtonID" Event="NewDialog"
Value="NextDialogID">1</Publish>
    </UI>
```

В примере выше для находящегося на диалоге CustomDialogID элемента с идентификатором ButtonID назначается событие NewDialog. При вызове события в качестве аргумента ему будет передано значение NextDialogID. Текст внутри элемента может содержать логическое условие; если условие на момент вызова вычисляется как ложное, соответствующее событие не будет сгенерировано. Единица всегда вычисляется как истинное условие.

Замечание: задание условия не является обязательным и не контролируется в процессе сборки – при его отсутствии единственное публикуемое событие будет вызвано всегда. Но в случаях, когда в элементе описано несколько событий, для них следует явно задавать условия – в противном случае может быть вызвано только первое событие из списка.

При использовании элемента Publish внутри элемента управления нет необходимости задавать значения атрибутов Dialog и Control – они будут унаследованы от родительских элементов.

Еще один атрибут – Order – позволяет установить порядок публикации событий. Он незаменим в тех случаях, когда необходимо переназначить события для элементов, описанных в другом месте.

```
    <Control Id="SelectApplication" Type="PushButton" X="230" Y="100" Width="100" Height="17"
Text="Выбрать">
      <!-- SelectApplication – собственная расширенная операция -->
      <Publish Event="DoAction" Value="SelectApplication" Order="1">1</Publish>
      <Publish Property="APPLICATIONTOOPENLOG" Value="[APPLICATIONTOOPENLOG]"
Order="2">1</Publish>
    </Control>
```

Механизм событий также позволяет устанавливать значения свойств, для чего следует использовать приведенный ниже синтаксис. В этом случае идентификатор свойства помещается в атрибут Property, а в атрибут Value – присваиваемое значение.

```
        <Control Id="SetPropertyButton" Type="PushButton" Text="Установить" X="315" Y="80"
Height="22" Width="60">
          <Publish Value="Новое значение" Property="ComboBoxProperty" />
        </Control>
```

Наличие значения в атрибуте Value является обязательным, поэтому, чтобы сбросить значение свойства, оно должно быть инициализировано следующим образом:

```
          <Publish Value="{}" Property="ComboBoxProperty" />
```

Для подписки на события используется элемент Publish, всегда помещаемый внутрь элемента управления. Для него задаются значения двух атрибутов – в Event помещается название события, а в Attribute – атрибут, в который будет помещено полученное от события значение.

```
        <Control Id="ItemPath" Type="Text" X="250" Y="120" Width="180" Height="50" Text="Путь
установки выбранного набора" TabSkip="yes" Sunken="yes">
          <Subscribe Event="SelectionPath" Attribute="Text" />
        </Control>
```

Какую часть работы придется выполнить вручную – зависит от типа события. Все автоматически публикуемые события передают подписчику один дополнительный параметр. Чаще он является строковым, иногда числовым, поэтому в роли универсального подписчика может выступать элемент управления Text.

Все события можно условно разделить на пять категорий. К первой можно отнести информационные события, публикуемые службой Windows Installer. Их обработка позволяет повысить наглядность процесса установки. К ним относятся:

- ActionData - описание выполняемого в данный момент действия, например: «Файл File1.bin, каталог: c:\Program Files\Demo Application\, размер: 24027648»;
- ActionText - название выполняемого в данный момент действия, например: «Идет копирование новых файлов»;
- ScriptInProgress – возвращает единицу в момент компиляции исполняемого скрипта, после окончания – нулевое значение;
- TimeRemaining – расчетное время до окончания установки – при установке будет заполнено значением вида: «Осталось: 10 сек.»;
- SetProgress – текущий прогресс установки.

События ActionData и ActionText передают форматированный текст, поэтому в качестве подписчика может выступать только элемент Text. Событие SetProgress передает числовое значение, корректно обрабатываемое элементом ProgressBar.

Для обработки события TimeRemaining на него необходимо подписаться несколько иначе: в атрибут Attribute элемента Publish соответствующего элемента Text помещается значение TimeRemaining:

```
        <Control Type="Text" Id="textTimeRemaining" Width="116" Height="17" X="22" Y="56" >
            <Subscribe Event="TimeRemaining" Attribute="TimeRemaining" />
        </Control>
```

Перечисленные выше события генерируются только в процессе установки, а потому имеют смысл только на соответствующем диалоге. Пример использования указанных событий приведен ниже в этой главе, в разделе "Отображение прогресса установки".

Далее идут события, неявно публикуемые элементом SelectionTree, позволяющие предоставить пользователю дополнительную информацию о выбранном наборе, его размере и местоположении. К ним относятся:

- SelectionAction - описание действия, предусмотренного для выбранного набора;

- SelectionDescription - описание выбранного набора. Отображает значение атрибута Description элемента Feature;
- SelectionNoItems – публикуется при отсутствии выделенных узлов для очистки текстовых полей и перевода элементов управления в недоступное состояние;
- SelectionPath - путь к выбранному набору, при наличии;
- SelectionPathOn – возвращает единицу, если для выбранного набора предусмотрен выбор пути установки, иначе нулевое значение;
- SelectionSize - размер устанавливаемого набора и, при наличии, всех вложенных.

Примеры использования данных событий, а также событий AddLocal, AddSource и Remove приведены выше в данной главе в разделе «Наборы компонентов и связанные задачи (SelectionTree, VolumeCostList)».

Следующая группа событий позволяет инициировать переходы между диалогами, добавлять и удалять наборы, устанавливать значения некоторых стандартных свойств. Данные события генерируются, преимущественно, при нажатии кнопки, но небольшая их часть может также порождаться действиями переключателей. Начнем рассмотрение с общих для кнопки и переключателя событий:

- AddLocal - настраивает набор, идентификатор которого указан в аргументе, для локальной установки. Допустимо значение All - для выбора всех наборов;
- AddSource - настраивает набор, идентификатор которого указан в аргументе, для установки с источника. Допустимо значение All - для выбора всех наборов;
- Remove - удаляет набор, идентификатор которого указан в аргументе. Допустимо значение All.

Отдельного внимания заслуживает событие DoAction – оно позволяет выполнить стандартную или расширенную операцию. Фактически, данное событие позволяет немедленно запустить допустимое в данный момент действие в ответ на действия пользователя над элементами интерфейса. Его применение тесно связано с созданием пользовательских операций, поэтому оно рассматривается в главе 6, в разделе «Расширение функционала с помощью элемента CustomAction».

И, наконец, события, порождать которые способна только кнопка:

- EnableRollback – позволяет включить или отключить поддержку отката транзакции;
- EndDialog - уведомляет службу о необходимости закрыть текущий диалог. Аргумент принимает значения: Return - нормальное завершение, Exit – прерывание процесса установки, Retry – закрытие диалога с возвратом значения Suspend, Ignore – закрытие диалога с возвратом значения Finished;
- NewDialog - уведомляет службу о необходимости удалить текущий диалог и открыть новый, идентификатор которого указан в аргументе;
- Reinstall - запрашивает немедленную переустановку набора с указанным идентификатором. Допустимо значение ALL;
- ReinstallMode - устанавливает значение свойства REINSTALLMODE;

- Reset - восстанавливает состояние всех элементов управления на момент создания диалога;
- SetInstallLevel - устанавливает значение свойства INSTALLLEVEL;
- SetTargetPath – используется для назначения пути установки. Вызывается внутри диалога, отображаемого с помощью события SelectionBrowse.
- SpawnDialog – создает и отображает новый диалог поверх существующего;
- SpawnWaitDialog - создает диалог поверх существующего в случае, если условие вычисляется как FALSE. Диалог удаляется, как только условие вычисляется как TRUE.

Событие SelectionBrowse отображает модальный диалог, позволяющий выбрать каталог для устанавливаемого локально набора. Отличие от события SpawnDialog заключается в том, что контролируется изменение пути установки для выделенного набора. Внутри вызванного диалога для присваивания нового пути используется событие SetTargetPath.

Последние события позволяют управлять поведением элемента управления DirectoryList, для чего их следует назначить располагающимся на этом же диалоге кнопкам:

- DirectoryListUp - возвращает из текущего каталога на один уровень вверх;
- DirectoryListOpen - переходит внутрь выбранного каталога;
- DirectoryListNew - создает новый каталог с возможностью изменения его имени.

Событие IgnoreChange, также публикуемое элементом DirectoryList, позволяет изменить взаимное поведение элементов DirectoryList и DirectoryCombo. Если на одном диалоге располагаются оба этих элемента, как в примере на рисунке 5.6, то при выделении каталога в DirectoryList он сразу же отображается и в DirectoryCombo. Если же подписать DirectoryCombo на событие IgnoreChange, как показано ниже, то он будет отображать только название родительского каталога, что является более привычным поведением.

```
    <Control Id="DirectoryCombo" Type="DirectoryCombo" X="70" Y="55" Width="220" Height="80"
Property="INSTALLLOCATION" Fixed="yes" Remote="yes">
      <Subscribe Event="IgnoreChange" Attribute="IgnoreChange" />
    </Control>
```

Отображение прогресса установки

После завершения всех этапов взаимодействия с пользователем начинается непосредственно установка ресурсов на целевую машину. В зависимости от размера файлов и перечня выполняемых операций, этот процесс может быть достаточно длительным, поэтому во всех стандартных наборах используется диалог ProgressDlg. Данный диалог отображает индикатор состояния, описание выполняемой в настоящий момент операции и подходит для подавляющего большинства случаев. Однако может возникнуть потребность в замене этого диалога собственным.

Нам потребуется собственный диалог прогресса установки, которому мы присвоим идентификатор ProgressModDlg. Разместим на нем индикатор состояния и текстовые поля, которые будут подписаны на события, информирующие о ходе установки: ActionText и ActionData. Кроме того, «оживим» диалог с помощью элемента управления Billboard, подробно описанного ранее в этой главе – позаимствуем разметку из этого раздела.

```
      <!-- Модифицированный диалог прогресса -->
      <Dialog Id="ProgressModDlg" Width="370" Height="270" Title="!(loc.ProgressDlg_Title)"
Modeless="yes">
        <Control Id="Cancel" Type="PushButton" X="304" Y="243" Width="56" Height="17"
Default="yes" Cancel="yes" Text="!(loc.WixUICancel)">
          <Publish Event="SpawnDialog" Value="CancelDlg">1</Publish>
        </Control>
        <Control Id="BannerBitmap" Type="Bitmap" X="0" Y="0" Width="370" Height="44"
TabSkip="no" Text="!(loc.ProgressDlgBannerBitmap)" />
        <Control Id="Back" Type="PushButton" X="180" Y="243" Width="56" Height="17"
Disabled="yes" Text="!(loc.WixUIBack)" />
        <Control Id="Next" Type="PushButton" X="236" Y="243" Width="56" Height="17"
Disabled="yes" Text="!(loc.WixUINext)" />
        <Control Id="Title" Type="Text" X="15" Y="15" Width="200" Height="15"
Transparent="yes" NoPrefix="yes">
          <Text>{\WixUI_Font_Title}Выполняется установка приложения</Text>
        </Control>
        <Control Id="BannerLine" Type="Line" X="0" Y="44" Width="370" Height="0" />
        <Control Id="BottomLine" Type="Line" X="0" Y="234" Width="370" Height="0" />
        <Control Id="ProgressBar" Type="ProgressBar" X="20" Y="100" Width="330" Height="10"
ProgressBlocks="yes" Text="!(loc.ProgressDlgProgressBar)">
          <Subscribe Event="SetProgress" Attribute="Progress" />
        </Control>
        <Control Id="textActionText" Type="Text" X="22" Y="86" Width="285" Height="10"
Text="ActionText">
          <Subscribe Event="ActionText" Attribute="Text" />
        </Control>
        <Control Type="Text" Id="textActionData" Width="323" Height="17" X="22" Y="71"
Text="ActionData">
          <Subscribe Attribute="Text" Event="ActionData" />
        </Control>
        <Control Id="Billboard1" Type="Billboard" X="20" Y="115" Width="330" Height="110"
TabSkip="yes" Disabled="yes">
          <Subscribe Event="SetProgress" Attribute="Progress" />
        </Control>
        <Control Type="Text" Id="textTimeRemaining" Width="116" Height="17" X="22" Y="56">
          <Subscribe Event="TimeRemaining" Attribute="TimeRemaining" />
        </Control>
      </Dialog>
```

Кроме прочего, для поддержки описываемого элементом управления Billboard содержимого нам потребуются изображения с идентификаторами Bitmap1 и Bitmap2. Также необходимо наличие наборов компонентов: родительского ProductFeature и вложенных в него FeatureSet1 и

FeatureSet2. Идентификаторы наборов используются для определения отображаемого в настоящий момент содержимого на элементе Billboard.

Завершив описание диалога, заменим им стандартный, поместив его непосредственно перед операцией ExecuteAction:

```
<InstallUISequence>
    <Show Dialog="ProgressModDlg" Before="ExecuteAction" />
</InstallUISequence>
```

В процессе установки будет отображен наш новый диалог, приведенный на рисунке 5.9.

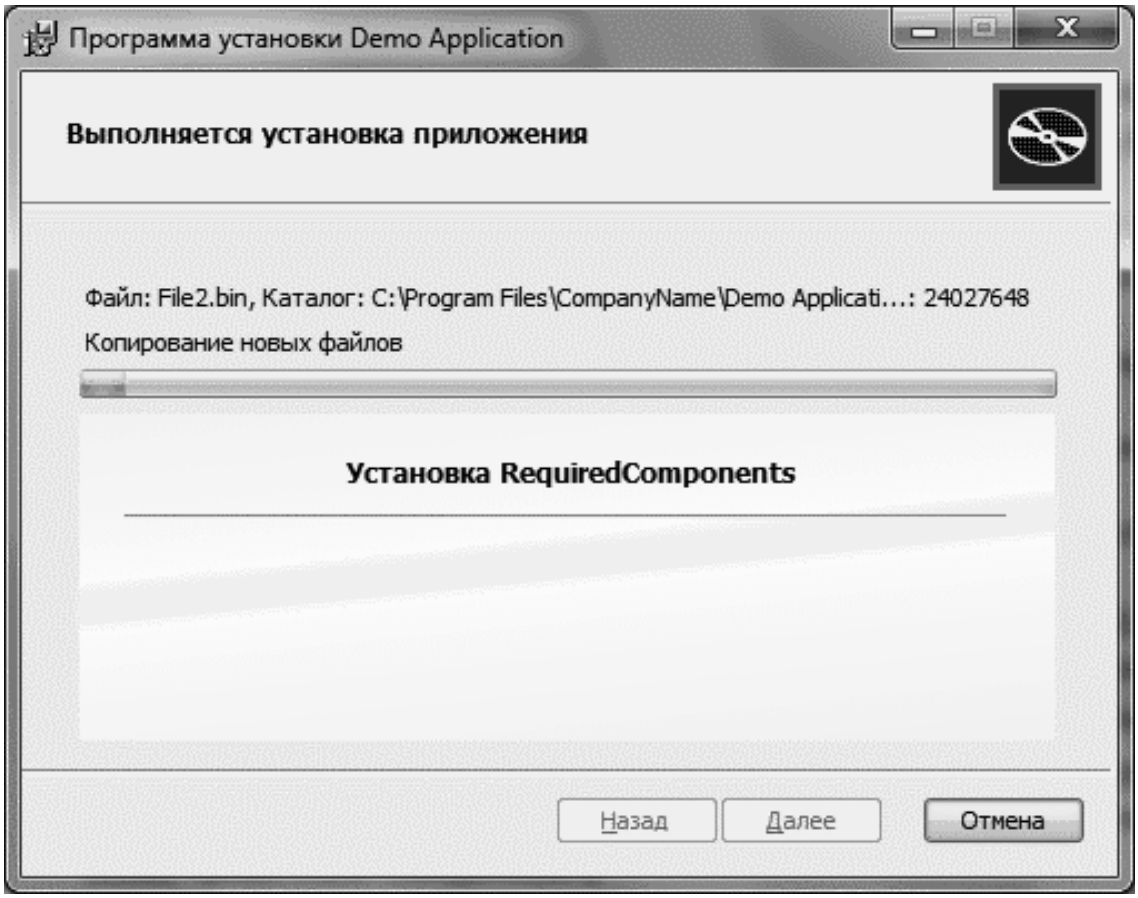

Рисунок 5.9 Диалог прогресса установки с элементом Billboard.

Локализация ресурсов

Во всех примерах, приведенных в данной книге, мы размещали строковое содержимое – надписи на кнопках, поясняющий текст – непосредственно в коде разметки. Это удобно ровно до тех пор, пока не возникнет необходимость локализации приложения – и программы установки – на другой язык. Данная задача является достаточно общей, поэтому Windows Installer XML предлагает ее решение: все подлежащие переводу строки представляются в виде переменных, для каждой из которых могут существовать многочисленные варианты перевода. Для каждой языковой культуры создается файл с расширением *.wxl, при этом содержимое необходимого используется в процессе сборки.

Чтобы добавить файл локализации в проект, необходимо вызвать меню Add -> New Item, выбрать пункт WiX Localization File, указать его имя нажать кнопку OK. Имя файла формируется следующим образом: «произвольное_имя».«языковая культура».wxl, например ExitDialog.ru-ru.wxl. Так будет выглядеть файл локализации для русского языка с единственным строковом значением ExitConfirmationText:

```
<?xml version="1.0" encoding="utf-8"?>
<WixLocalization Culture="ru-ru" Codepage="1251"
xmlns="http://schemas.microsoft.com/wix/2006/localization">
   <String Id="ExitConfirmationText" Overridable="yes">Вы действительно хотите выйти?</String>
</WixLocalization>
```

В атрибутах Culture и Codepage корневого элемента WixLocalization указываются наименование культуры и номер кодовой страницы. В элементе String устанавливается идентификатор значения, а присвоение значения yes атрибуту Overridable позволяет переопределять его в последующем.

Для использования значения в коде разметки применяется синтаксис !(loc.PropertyName), например:

```
        <Control Type="Text" Id="textQuestion" Width="128" Height="17" X="38" Y="15"
Text="!(loc.ExitConfirmationText)" />
```

Стандартные диалоги используют этот же подход; в настоящее время в Windows Installer XML официально поддерживаются 12 языковых культур.

Визуальное проектирование диалоговых окон

Проектирование диалоговых окон в Visual Studio является в настоящее время длительным и трудоемким процессом ввиду отсутствия механизма отображения диалогов. Вероятно, в будущем эта возможность будет реализована. Сейчас же можно порекомендовать использовать для этих целей свободно распространяемую утилиту WixEdit, которую можно найти и скачать по адресу http://wixedit.sourceforge.net/. Этот инструмент позволяет выполнять основные операции без необходимости редактирования XML-содержимого, но к наиболее полезным, несомненно, следует отнести наличие WYSIWYG-редактора диалоговых окон, приведенного на рисунке 5.10.

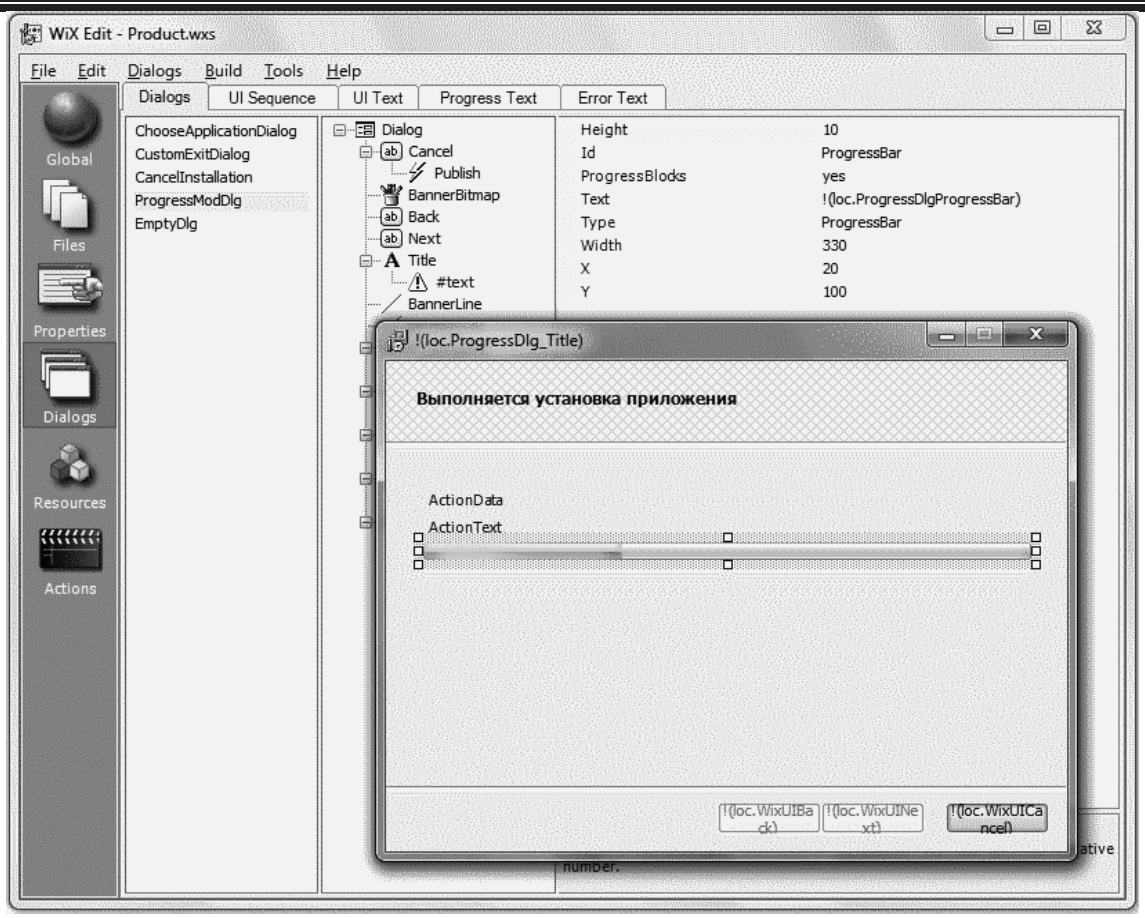

Рисунок 5.10 WixEdit – редактор диалогового окна.

Данный редактор также обладает некоторыми ограничениями, например, не может отобразить диалоги, внутри которых присутствуют XML-комментарии. Тем не менее, он может существенно упростить задачу создания и редактирования интерфейсов в программах установки.

Глава 6. Последовательности, стандартные и расширенные операции

Установка, производимая службой Windows Installer, заключается в выполнении последовательности из множества простых действий. Только перечисление перечня доступных операций может занять значительное время. Поэтому не может не радовать тот факт, что таблицы последовательностей в большинстве случаев будут заполняться даже без нашего участия – как я уже писал ранее, создание значительной части программ установки может быть выполнено даже без представления о последовательностях и стандартных операциях. Тем не менее, сложные установочные пакеты могут потребовать в значительной степени изменить порядок и набор выполняемых действий. Windows Installer XML в данном вопросе дублирует функциональность Windows Installer, размещая описания таблиц последовательности в простых узлах. В этой главе рассматриваются общие вопросы использования операций и отличия их применения в WiX – подробная информация по данному вопросу может быть найдена в библиотеке MSDN.

Доступные режимы установки и уровни интерфейса

До начала рассмотрения таблиц последовательностей следует рассказать о режимах установки и различных уровнях отображения пользовательского интерфейса, так как эти вопросы тесно связаны. Отличия обусловлены тем, что технология Windows Installer разрабатывалась с учетом поддержки разнообразных сценариев установки: вручную пользователем при взаимодействии с интерфейсом либо в автоматизированном режиме администратором сети. Кроме того, доступны более специфические варианты установки, а также их комбинации.

Работа программы установки в том или ином режиме зависит от способа запуска: перечисленная функциональность является частью технологии и для ее поддержки не требуется каких-либо дополнительных действий со стороны разработчика пакета.

Обычная установка, административная и по требованию

Windows Installer поддерживает три вида установки: обычную, административную и по требованию. Выполняемые в каждом случае наборы операций могут существенно отличаться, что требует отдельного описания последовательностей для каждого из них. Кроме того, поскольку установочный пакет может запускаться в различных режимах отображения интерфейса – полном, упрощенном, базовом и без интерфейса – это также влияет на способ отображения элементов пользовательского интерфейса.

Рассмотрим режимы установки подробнее.

Обычная установка представляет собой наиболее привычный метод установки программ, когда все действия по копированию ресурсов и настройке выполняются единовременно после ее запуска. Может выполняться как с отображением интерфейса пользователя, так и без него.

Административная установка позволяет администратору создать на сетевом диске образ, который в последующем может быть использован для установки программы на клиентские машины. Если настроить запуск файлов с источника, это позволит избежать копирования файлов

на каждую клиентскую машину. Аналогично обычной установке, выполняется с отображением интерфейса или без него.

Замечание: для запуска административной установки используется ключ /a:

msiexec.exe /a [путь к msi-пакету]

Установка «по требованию» всегда производится неявно для пользователя и не предполагает его взаимодействия с элементами интерфейса. Она заключается в том, что при начальной установке на целевой машине создаются только интерфейсы (управляемые ярлыки) без непосредственного копирования файлов. При первом обращении к файлу (например, к файлу справки) производится его фактическое копирование на целевую машину – это позволяет экономить место, так как многие элементы могут никогда не потребоваться пользователю.

Уровни отображения интерфейса

Как было сказано выше, обычная и административная установки позволяют установить уровень отображения интерфейса.

Полный (Full) режим применяется при ручной установке, при этом отображаются модальные и немодальные диалоги и сообщения об ошибках. Такая установка предполагает обязательное взаимодействие с пользователем.

В упрощенном (Reduced) режиме не отображаются модальные диалоговые окна, за исключением сообщений об ошибках и некоторых других диалогов.

В базовом (Basic) режиме отображается только немодальный диалог, отображающий прогресс процесса установки. Также будут показаны диалоги с сообщениями об ошибках.

Установка без отображения интерфейса (no UI), как следует из названия, не отображает каких-либо видимых окон, выполняя «тихую» установку. Использование последних двух режимов позволяет автоматизировать процесс инсталляции на основе предварительно подготовленного файла сценария.

Замечание: для указания уровня отображения интерфейса используются разновидности ключа /q:

/q, /qn – без интерфейса;

/qb – базовый режим (/qb! – без кнопки «Отмена»). Дополнительное указание знака «+» позволяет отобразить модальный диалог после успешного завершения установки (/qb+). Использование знака «-» отключает отображение модальных окон (/qb-);

/qr – упрощенный режим.

Например, для запуска пакета без отображения интерфейса следует указать:

msiexec.exe /qn [путь к msi-пакету]

Реализация таблиц последовательностей в WiX

Операции выполняются по очереди, при этом циклы и ветвления в них не предусмотрены – это позволяет использовать для их описания простые таблицы. Существует по две таблицы для

обычной и административной установки и одна таблица для установки по требованию, описываемые следующими элементами схемы:

- InstallUISequence и InstallExecuteSequence – описание последовательностей, выполняемых при обычной установке;
- AdminUISequence и AdminExecuteSequence – содержимое последовательностей, выполняемых в процессе административной установки;
- AdvertiseExecuteSequence – используется при выполнении установки по требованию.

Содержимое таблиц InstallUISequence и AdminUISequence используется в тех случаях, когда используется полный или упрощенный уровень отображения интерфейса. В них кроме перечня операций помещаются идентификаторы диалоговых окон – для определения взаимной последовательности их вызовов.

Структура всех таблиц последовательностей Windows Installer идентична и включает три столбца – идентификатор операции, порядковый номер, определяющий порядок запуска и условие, позволяющее пропустить выполнение операции. Разработчики WiX создали для каждой из стандартных операций отдельный элемент. С одной стороны, это снижает количество ошибок, связанных с опечатками; с другой – перегружает схему и справочную систему множеством – а их более семидесяти – элементов, потребность в которых возникает достаточно редко. Тем не менее, это позволяет контролировать на основании схемы правильность вложенности элементов и заполнения значений их атрибутов.

Существенное отличие таблиц, используемых в режиме отображения пользовательского интерфейса, заключается в том, что в общую последовательность включаются диалоги, для чего используется элемент Show. Таким образом, на выходе мы получаем комбинацию вызовов операций и диалогов, как показано в примере ниже. Элементы Show описывают диалоговые окна, а LaunchConditions и все последующие – операции. Внутри элементов могут дополнительно указываться условия, при этом вызов соответствующего диалога или операции производится только в том случае, когда условие вычисляется как истинное:

```
    <InstallUISequence>
      <Show Dialog="Fatal_Error" OnExit="error" />
      <Show Dialog="User_Exit" OnExit="cancel" />
      <Show Dialog="Exit_Dialog" OnExit="success" />
      <Show Dialog="Setup_Dialog" Sequence="140" />
      <Show Dialog="Welcome_Dialog" Sequence="170">NOT Installed</Show>
      <Show Dialog="Maintenance_Welcome_Dialog" Sequence="1250">Installed AND NOT RESUME AND
NOT Preselected AND NOT PATCH</Show>
      <Show Dialog="Progress_Dialog" Sequence="1280" />
      <LaunchConditions Sequence="100" />
      <FindRelatedProducts Sequence="146" />
      <AppSearch Sequence="150" />
      <CostInitialize Sequence="165" />
      <FileCost Sequence="166" />
      <IsolateComponents Sequence="167" />
      <CostFinalize Sequence="168" />
      <MigrateFeatureStates Sequence="169" />
```

```
    <ExecuteAction Sequence="1300" />
  </InstallUISequence>
```

Все таблицы последовательностей размещаются внутри тега Product, но таблицы InstallUISequence и AdminUISequence также можно поместить внутрь элемента UI.

Каждый элемент последовательности может иметь до пяти атрибутов: After, Before, Overridable, Sequence и Suppress:

- After – строка, содержащая имя предыдущего элемента последовательности;
- Before – строка, указывает на следующий элемент последовательности;
- Overridable – yes/no – может ли поведение быть переопределено в другом месте;
- Sequence – числовое значение, определяющее порядок элемента в последовательности. Не может использоваться совместно с After и Before;
- Suppress – yes/no – выполнять или нет операцию.

Как и для операций, для элемента Show также определены атрибуты After, Before, Overridable и Sequence, но присутствуют два дополнительных атрибута: Dialog и OnExit. Dialog используется для указания идентификатора отображаемого диалога. Атрибут OnExit устанавливает необходимость отобразить указанный диалог в одном из четырех особых случаев: success – в конце успешно завершенной установки; cancel – в случае отмены установки пользователем; error – при ошибке в ходе установки и suspend – при приостановке процесса установки для его последующего возобновления.

Расширение функционала с помощью элемента CustomAction

Для реализации сложной логики незаменимым является элемент CustomAction. Представляя собой расширенную операцию, он может вызываться при взаимодействии пользователя с элементами управления или добавлен в последовательность операций. Описанная в элементе CustomAction операция может быть синхронной или асинхронной, с проверкой возвращаемого значения и без него. Данное поведение определяется атрибутом Return, принимающим одно из четырех значений:

- check – ожидание завершения операции с проверкой результата;
- ignore – ожидание завершения операции без проверки результата;
- asyncWait и asyncNoWait – асинхронный вызов операции с проверкой результата или без нее соответственно.

Элемент CustomAction позволяет решать множество задач, в том числе:

- присваивать значения свойствам;
- прерывать установку с сообщением об ошибке;
- запускать исполняемые файлы с возможностью передачи им аргументов;
- вызывать функции, описанные во внешних библиотеках.

Для включения в последовательность используется элемент Custom:

```
  <InstallUISequence>
    <Custom Action="SelectApplication" Before="InstallFinalize" />
```

```
    </InstallUISequence>
```

Для вызова операции при нажатии на кнопку или переключатель предназначено событие DoAction:

```
      <Control Id="SelectApplication" Type="PushButton" X="230" Y="100" Width="100"
Height="17" Text="Выбрать">
        <Publish Event="DoAction" Value="SelectApplication" Order="1">1</Publish>
      </Control>
```

В приведенных выше примерах используется наша собственная расширенная операция SelectApplication, создание которой подробно описано ниже в этой главе.

Присваивание значения свойству

Возможность присваивания значения свойству позволяет обрабатывать завершение выполнения той или иной операции. Это может быть востребовано тех в случаях, когда часть значения вычисляется в процессе работы другой операции или вводится пользователем. Непосредственно присваивание значения выглядит так:

```
    <CustomAction Id="SetStarted" Property="IsStarted" Value="Запущено" />
```

Атрибуту Id присваивается уникальный идентификатор операции, Property – идентификатор свойства-получателя, Value – присваиваемое значение. Присваиваться может как константа, так и составленное из других свойств значение, например:

```
    <CustomAction Id="SetFileFullName" Property="FileFullName"
Value="[PathProperty][FileNameProperty].exe" />
```

Если какое-либо из образующих значение свойств не инициализировано, вместо него будет подставлена пустая строка.

Для решения этой же задачи может быть использован специально предназначенный элемент SetProperty, формирующий аналогичные записи в базе данных Windows Installer:

```
    <SetProperty Id="IsStarted" After="SelectApplication" Sequence="both" Value="1" />
```

Здесь в атрибут Id помещается идентификатор получающего значение свойства. Взаимоисключающие атрибуты Before и After позволяют установить порядок выполнения операции, а в Value формируется присваиваемое значение. Для последнего атрибута – Sequence – допустимыми являются значения execute, ui и both. Установка в execute заносит запись на выполнение в таблицу InstallExecuteSequence, ui – в таблицу InstallUISequence, both – в обе перечисленные таблицы.

Элемент CustomAction также позволяет переназначить путь к каталогу. Для этого идентификатор каталога помещается в атрибут Directory:

```
    <CustomAction Id="SetFolderPath" Directory="InstallFolderPath" Value="C:\Program
Files\Demo Application" />
```

Прерывание установки с сообщением об ошибке

Элемент Condition, помещенный внутрь Product, позволяет проверить выполнение обязательных условий при запуске программы установки, а при их невыполнении – прервать установку. Чтобы

прервать уже работающий процесс, следует воспользоваться другой возможностью элемента CustomAction:

```
    <CustomAction Id="AbortInstallation" Error="Установка прервана" />
```

Такое поведение может потребоваться в случаях, когда специфическая проверка выполняется в собственной операции, то есть уже в процессе работы программы установки.

Запуск исполняемого файла

Необходимость запуска исполняемого файла возникает достаточно часто. Это может быть запуск самой программы после ее установки или утилиты настройки. В некоторых случаях может потребоваться открытие для просмотра документов или переход к веб-странице. Данная задача также решается с помощью элемента CustomAction. Для запуска скопированного в процессе работы программы установки файла операция выполняется асинхронно без проверки возвращаемого значения. Чтобы запустить файл, следует поместить его идентификатор в атрибут FileKey:

```
    <CustomAction Id="RunApplication" FileKey="ApplicationMain" Return="asyncNoWait" />
```

В атрибут ExeCommand дополнительно могут быть помещены аргументы, передаваемые исполняемому файлу при запуске. Чтобы передать в качестве аргумента путь к файлу, перед его идентификатором следует поместить символ «#»:

```
    <CustomAction Id="OpenDocumentWithEditor" FileKey="EditorExe" ExeCommand="[#ReadmeTxt]"
Return="asyncNoWait" />
```

Возможен вариант, когда запускаемый файл не копируется на целевую машину, а включен в установочный пакет в виде двоичного ресурса:

```
    <Binary Id="ConfigExe" SourceFile="$(var.ConfigurationUtility.TargetPath)" />
```

В этом случае вместо атрибута FileKey используется BinaryKey, куда помещается идентификатор двоичного ресурса.

```
    <CustomAction Id="RunConfigurator" BinaryKey="ConfigExe" Return="asyncNoWait" />
```

И, наконец, вариант, когда необходимо запустить внешний исполняемый файл. Прежде всего, необходимо определить переменную, значением которой является полный путь к данному файлу или только его имя, если путь содержится в переменной среды PATH. Наиболее популярным является пример открытия копируемого текстового документа с помощью «блокнота».

```
    <Property Id="NotepadExe" Value="Notepad.exe" />
```

Теперь для запуска приложения необходимо указать идентификатор свойства в атрибуте Property. Как и в других случаях, можно дополнительно указать передаваемые при вызове аргументы:

```
    <CustomAction Id="OpenDocumentWithNotepad" Property="NotepadExe" ExeCommand="[#ReadmeTxt]"
Return="asyncNoWait" />
```

Вызов функций, определенных во внешних библиотеках

Пожалуй, наиболее важной возможностью элемента CustomAction является вызов с его помощью функций, описанных во внешних библиотеках. Такие функции могут быть описаны в библиотеках расширения, как показано в разделе «Расширение WixDirectXExtension» главы 4, или созданы

самостоятельно. Предположим, что уже существует библиотека ThirdParty.dll, внутри которой с соблюдением всех соглашений описана функция DoSomething. Вопросы создания собственной операции на языке C# и ее подключения, а также краткий обзор доступной .NET разработчику объектной модели описаны ниже в этой главе.

Добавим файл библиотеки в проект, воспользовавшись меню Add -> Existing Item. С помощью элемента Binary поместим его в пакет:

```
    <Binary Id="ThirdPartyDll" Source="ThirdParty.dll" />
```

Теперь опишем операцию, вызывающую функцию из данной библиотеки:

```
    <CA Id="DoSomethingFunction" BinaryKey="ThirdPartyDll" DllEntry="DoSomething" />
```

Единственным атрибутом, который мы не рассматривали ранее, является DllEntry. В нем необходимо указать имя вызываемой функции.

Отложенное выполнение операции

Описанная выше операция DoSomethingFunction инициирует немедленное выполнение метода. Такое поведение обуславливается атрибутом Execute, установленным по умолчанию в immediate. Немедленное выполнение позволяет получить доступ к контексту, в том числе читать и записывать значения свойств. Однако в ряде случаев может потребоваться отложенное выполнение операции, после завершения установки программы, для чего следует присвоить атрибуту Execute значение deferred. Отложенная операция имеет доступ к очень ограниченному набору параметров, но есть возможность передать ей одно или более значений, помещаемых внутрь свойства CustomActionData. Инициализация данного свойства производится с помощью дополнительной операции:

```
    <!-- Отложенная операция -->
    <CustomAction Id="DeferredAction" BinaryKey="CustomActionCADll"
DllEntry="DeferredOperation" Execute="deferred" />
    <CustomAction Id="SetDeferredActionData" Property="DeferredAction"
Value="Parameter1=Значение 1;LogApplication=[APPLICATIONTOOPENLOG]" />
```

Значения передаются в формате [Имя параметра]=[Значение], различные параметры отделяются точкой с запятой и могут содержать форматированный текст.

Идентификаторы операций назначаются произвольно, но в атрибуте Property устанавливающей значение операции должен быть указан идентификатор отложенной операции. Кроме того, необходимо запланировать выполнение устанавливающей операции до запуска отложенной, например, как показано ниже:

```
    <InstallExecuteSequence>
      <Custom Action="SetDeferredActionData" Before="InstallFiles" />
      <Custom Action="DeferredAction" Before="InstallFinalize" />
    </InstallExecuteSequence>
```

Если посмотреть на объект CustomActionData в режиме отладки, то видно, что внутри он содержит коллекцию, ключи и значения которой представлены строковыми значениями.

```
session.CustomActionData
```

```
{Parameter1=Тестовое значение 1;LogApplication=notepad.exe}
   Count: 2
   IsReadOnly: false
   Keys: Count = 2
   Values: Count = 2
session.CustomActionData["Parameter1"]
"Тестовое значение 1"
session.CustomActionData["LogApplication"]
"notepad.exe"
```

Заполнение и обработка содержимого данного объекта полностью возлагается на разработчика и зависит от характера решаемой задачи. Обращение к отдельным параметрам выполняется по имени с использованием индексатора.

Создание операции и добавление в последовательность

При разработке программ установки однажды наступит тот момент, когда стандартный функционал перестанет удовлетворять нашим потребностям. И в этом случае нам потребуется вызывать методы, описанные во внешних библиотеках. Собственно, библиотеки расширения целесообразно писать на неуправляемом коде, чтобы не добавлять лишних зависимостей в проект установки. Но, предпочитая удобство при разработке и тестировании, я воспользуюсь возможностью написать библиотеку с использованием управляемого кода.

Добавим в решение проект типа C# Custom Action Project (или VB/C++ - в зависимости от предпочитаемого языка разработки). Ссылки на все необходимые библиотеки добавляются автоматически. Откроем сгенерированный файл CustomAction.cs и посмотрим на заглушку созданного метода CustomAction1:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Deployment.WindowsInstaller;

namespace CustomAction
{
   public class CustomActions
   {
      [CustomAction]
```

```
    public static ActionResult CustomAction1(Session session)
    {
      session.Log("Begin CustomAction1");
      return ActionResult.Success;
    }
  }
}
```

В открытый (public) класс добавлен открытый статический метод, помеченный атрибутом CustomAction. Данный метод принимает единственный параметр типа Session и возвращает элемент перечисления типа ActionResult. Заготовка метода записывает информацию в журнал событий и сообщает вызывающей стороне об успешном завершении выполнения операции.

Открытие файла с использованием расширенной операции

Теперь, когда создана заготовка метода, нам осталось реализовать непосредственно логику операции и подключить ее к проекту программы установки. Наш пример будет отображать диалог открытия файла. После закрытия диалога путь к выбранному файлу будет помещен в свойство APPLICATIONTOOPENLOG.

```
    [CustomAction]
    public static ActionResult SelectLogFileApplication(Session session)
    {
      session.Log("Begin SelectLogFileApplication");
      Microsoft.Win32.OpenFileDialog openFileDialog = new Microsoft.Win32.OpenFileDialog();
      openFileDialog.DefaultExt = "*.exe";
      openFileDialog.CheckFileExists = true;
      openFileDialog.CheckPathExists = true;
      openFileDialog.Filter = "*.exe|*.exe";
      openFileDialog.Multiselect = false;
      openFileDialog.Title = "Укажите приложение для открытия журнала";
      Boolean? showResult = openFileDialog.ShowDialog();
      if ((showResult.HasValue) && (showResult.Value))
      {
        session["APPLICATIONTOOPENLOG"] = openFileDialog.FileName;
      }
      return ActionResult.Success;
```

```
    }
```

Как видно из приведенного выше кода, взаимодействие с вызывающей стороной осуществляется через объект session типа Session. Данный тип является ключевым при программном взаимодействии с Windows Installer и содержит множество свойств и методов, поэтому к его рассмотрению мы вернемся несколько позже. Сейчас достаточно обратить внимание на доступ к открытым свойствам по имени с использованием индексатора. В начале метода также производится вызов метода Log для записи отладочной информации в журнал событий.

Для отображения диалога воспользуемся типом Microsoft.Win32.OpenFileDialog, описанным в сборке PresentationFramework.dll, что введет зависимость от .NET Framework версии 3.0. В данном примере это не имеет существенного значения, но в реальных системах к этому вопросу следует подходить осмысленно.

Теперь вернемся в проект установки. Добавим ссылку на проект расширенной операции (в нашем случае - CustomAction), для чего воспользуемся меню Add Reference… проекта установки. С помощью элемента Binary добавим ссылку на файл библиотеки, после чего опишем содержащуюся в ней операцию тегом CustomAction. В элементе CustomAction, кроме идентификатора, необходимо указать значения двух атрибутов – в BinaryKey помещается идентификатор описанного выше двоичного ресурса, а в DllEntry – имя вызываемой из библиотеки процедуры.

```
    <Binary Id="CustomActionCADll"
SourceFile="$(var.CustomAction.TargetDir)CustomAction.CA.dll" />
    <CustomAction Id="SelectApplication" BinaryKey="CustomActionCADll"
DllEntry="SelectLogFileApplication" />
```

Обратите внимание на элемент Binary: в атрибуте SourceFile указывается имя не библиотеки CustomAction.dll, которая также находится в выходном каталоге проекта, а библиотеки CustomAction.CA.dll. Это правило необходимо соблюдать всегда, в противном случае произвести вызов внешней процедуры не удастся.

Замечание: библиотека с именем <Имя_проекта>.CA.dll на самом деле представляет собой архив, в котором размещаются три файла:

<Имя_проекта>.dll;

<Имя_проекта>.config;

Microsoft.Deployment.WindowsInstaller.dll.

Результат работы операции будет помещен в свойство APPLICATIONTOOPENLOG, которому целесообразно присвоить значение по умолчанию:

```
    <Property Id="APPLICATIONTOOPENLOG" Value="notepad.exe" />
```

Для вызова нашей операции нам потребуется соответствующий диалог. Подробно добавление диалоговых окон описано в посвященной интерфейсу пользователя главе. Разметка простейшего диалога с текстовым полем и единственной кнопкой, запускающей данную операцию, приведена ниже.

```
      <Dialog Id="ChooseApplicationDialog" Width="370" Height="270" Title="Программа установки
[ProductName]" NoMinimize="yes" >
```

```
        <Control Id="ApplicationTextBox" Type="Edit" X="25" Y="100" Width="200" Height="17"
Property="APPLICATIONTOOPENLOG" TabSkip="no" ToolTip="Приложение для открытия файла журнала"
Disabled="yes" />
        <Control Id="SelectApplicationButton" Type="PushButton" X="230" Y="100" Width="100"
Height="17" Text="Выбрать" >
          <Publish Event="DoAction" Value="SelectApplication" Order="1">1</Publish>
          <Publish Property="APPLICATIONTOOPENLOG" Value="[APPLICATIONTOOPENLOG]"
Order="2">1</Publish>
        </Control>
      </Dialog>
```

Из описания диалога для краткости удалены строки, в которых создаются стандартные элементы управления – изображения, надписи и кнопки переходов; оставлена только необходимая для текущего примера кнопка. Описанный выше диалог со всеми декоративными элементами приведен на рисунке 6.1.

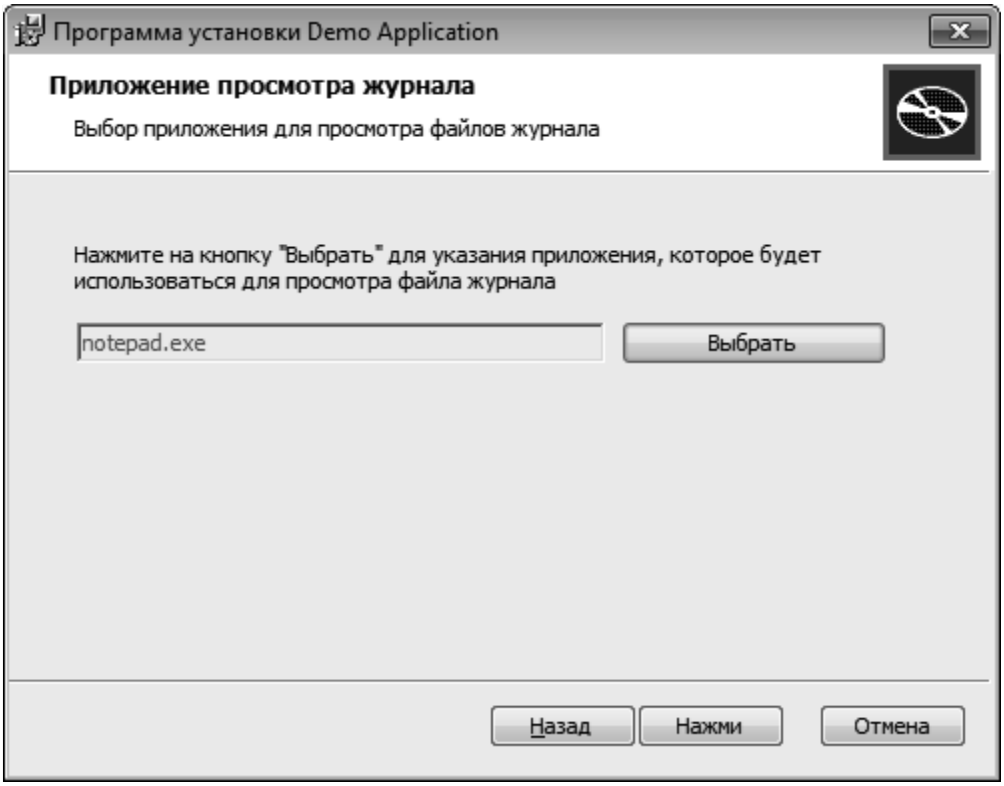

Рисунок 6.1 Диалог вызова расширенной операции.

Самая важная часть в описании диалога ChooseApplicationDialog – обработчик нажатия на кнопку SelectApplication.

```
        <Control Id="SelectApplicationButton" Type="PushButton" X="230" Y="100" Width="100"
Height="17" Text="Выбрать" >
          <Publish Event="DoAction" Value="SelectApplication" Order="1">1</Publish>
          <Publish Property="APPLICATIONTOOPENLOG" Value="[APPLICATIONTOOPENLOG]"
Order="2">1</Publish>
        </Control>
```

Сначала генерируется событие DoAction, которому передается имя стандартной или расширенной операции. В нашем случае вызывается описанная ранее операция SelectApplication. После выполнения свойству APPLICATIONTOOPENLOG будет присвоен путь к исполняемому файлу, предназначенному для открытия файлов журнала.

Вторая строка, в которой свойству APPLICATIONTOOPENLOG присваивается ее собственное значение, используется для немедленного обновления текстового поля. Если ее убрать, то после выбора исполняемого файла в поле останется предыдущее значение, которое будет обновлено только после перехода к следующему диалогу, а затем возврата к данному.

В заключение следует сказать о том, какие операции следует реализовывать на основании расширенных операций. Прежде всего, это первичное конфигурирование устанавливаемых приложений. Предполагая, что инсталляцию сложных приложений чаще выполняют квалифицированные сотрудники, целесообразно вынести в программу установки конфигурирование подключений к базам данных, указание адресов внешних служб с возможностью проверки их корректности. Например, конфигурирование строки подключения к БД SQL Server может выглядеть так, как показано на рисунке 6.2.

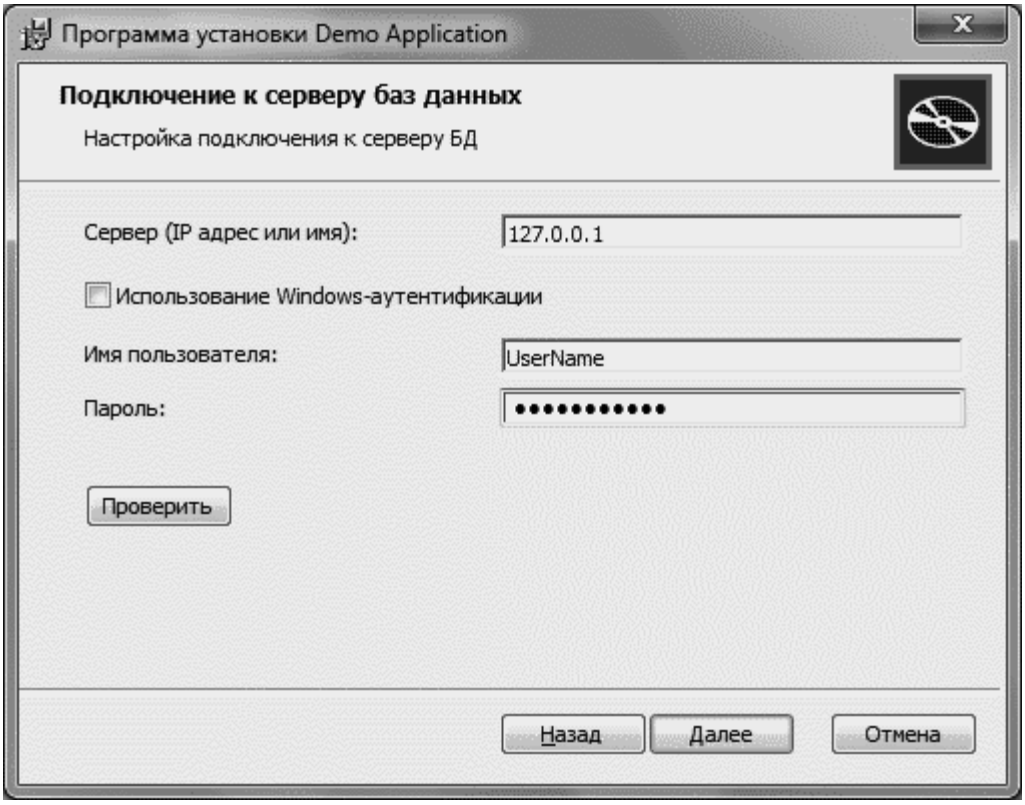

Рисунок 6.2 Диалог настройки подключения к базе данных.

Объект Session – основа взаимодействия с Windows Installer

Если внимательно посмотреть на текст любой расширенной операции, реализованной на управляемом коде, то видно, что взаимодействие с контекстом службы Windows Installer осуществляется через объект типа Session. Объектная модель данного типа позволяет контролировать ход процесса установки, предоставляя управляемые оболочки для важнейших

типов Windows Installer. Если не указано иное, используемые ниже типы объявлены в пространстве имен Microsoft.Deployment.WindowsInstaller.

Представление об основных свойствах и методах типа Session поможет разработчику сосредоточиться на необходимой приложению функциональности.

Индексатор позволяет прочитать или записать значение свойства, которое хранится в виде строки. Если при чтении свойство не существует, будет возвращена пустая строка.

```
String propertyName = session["PropertyName"];
session["APPLICATIONTOOPENLOG"] = openFileDialog.FileName;
```

В случае отсутствия свойства оно будет создано при присваивании ему значения.

Свойства Components и Features позволяют обратиться по имени к перечню компонентов и наборов соответственно. Они доступны только для чтения, возвращая их текущее состояние – CurrentState и состояние после завершения установки – RequestState. Реализуя интерфейсы IEnumerable и IEnumerable<T>, они поддерживают перебор элементов коллекции с помощью конструкции foreach. Кроме того, наличие индексатора позволяет обратиться к компоненту или набору по имени:

```
// Проход по элементам коллекции с помощью итератора.
ComponentInfoCollection components = session.Components;
foreach (var component in components)
{
    Console.WriteLine("Компонент: {0}. Текущее состояние: {1}. Состояние после установки: {2}.", component.Name, component.CurrentState, component.RequestState);
}
// Обращение к набору по имени.
FeatureInfo featureInfo = session.Features["RequiredComponents"];
```

Свойство Language возвращает числовой идентификатор используемого языка. Для русского языка возвращается значение 1049.

Свойство Database позволяет обращаться напрямую к установочной базе данных, в том числе выполнять запросы:

```
session.Database.ExecuteScalar("SELECT `Value` FROM `Property` WHERE `Property` = 'APPLICATIONTOOPENLOG'");
```

В тексте запроса названия таблиц и полей должны заключаться внутрь апострофов, а значения – внутрь одинарных кавычек.

Следующее свойство, CustomActionData, является единственным доступным для отложенных операций. Оно инициализируется с помощью другой расширенной операции и подробно описано ранее в этой главе, в разделе «Отложенное выполнение операции».

Тип Session также предоставляет ряд методов, в том числе:

- **DoAction**, как и одноименный элемент, используется для вызова диалогового окна или запуска расширенной операции

```
session.DoAction("WelcomeDlg");
```

- **EvaluateCondition** позволяет вычислить значение логического условия, представленного в виде строки с учетом используемых свойств.

```
session.EvaluateCondition("APPLICATIONTOOPENLOG=\"notepad.exe\"");
```

- **Format** предназначен для вычисления строковых значений с использованием значений свойств и учетом специальных символов.

```
session.Format("[APPLICATIONTOOPENLOG] [TARGETDIR]Docs\\Readme.txt");
```

- **Log** – данная функция позволяет выполнять запись текстовой информации в журнал, создаваемый при работе установщика.

```
session.Log("Begin TestOperation");
```

Таким образом, в Windows Installer XML версии 3.0 и старше поддерживается разработка расширенных операций на управляемом коде, при этом разработчику доступна богатая объектная модель. Если вводимая зависимость от наличия .NET Framework той или иной версии допустима, такой выбор является оправданным и дает ряд преимуществ в части удобства и скорости разработки.

Глава 7. Продвинутые возможности

В этой главе рассматриваются функции, позволяющие создавать развитые программы установки. Некоторые из описанных вопросов применимы только к Windows Installer XML, в то время как другие могут быть использованы с любыми msi-пакетами. Здесь же описывается процесс установки управляемой службы Windows с помощью WiX.

Анализ содержимого msi-сборок и интерпретация формируемых в процессе работы установщика журналов относится к технологии Windows Installer в общем, но декомпиляция рассматривается применительно к Windows Installer XML. Мы увидим, как можно извлечь функциональность существующего пакета и повторно применить ее в ваших собственных проектах. Особенно востребовано это может быть при создании пользовательских интерфейсов.

Здесь же мы рассмотрим создание загрузчика (в оригинале – Bootstrapper, устоявшийся вариант перевода отсутствует) – программы, позволяющей разрешать зависимости и последовательно устанавливать множество msi-пакетов, при необходимости предварительно выкачивая их дистрибутивы из сети. В соответствующем разделе рассматривается использование для этого бесплатного инструмента – пакета dotNetInstaller. Начиная с версии 3.6, в комплект поставки WiX входит утилита Burn, предназначенная для решения этой же задачи. Однако на сегодняшний день она еще не является достаточно зрелым инструментом и здесь не рассматривается.

Автоматическое обновление не является частью технологии Windows Installer. Для решения этой задачи могут быть использованы различные типовые решения, некоторые из которых кратко описаны ниже.

Установка служб Windows

Службы Windows выполняются в фоновом режиме, в том числе без необходимости входа пользователя в систему. Поддержка со стороны операционной системы, ведение журналов и автоматический перезапуск при необходимости позволяет применять их для решения широкого круга задач. Следует помнить, что для регистрации служб пакет следует запускать с правами локального администратора – соответствующую проверку желательно добавить при запуске.

Специфичные для регистрации службы элементы размещаются в том же компоненте, что и исполняемый файл, при этом указанный файл должен быть помечен как ключевой для компонента. В примере ниже в качестве ключевого установлен файл Server.NtService.exe.

Наиболее важным элементом, необходимым для регистрации службы, является ServiceInstall, в атрибутах которого устанавливаются основные свойства регистрируемой службы:

- Name – обязательный атрибут, содержит имя службы в системе;
- Start – обязательный атрибут, устанавливает тип запуска. Принимает значения auto – запуск при загрузке, demand – службу необходимо запускать явно, disabled – служба не будет запускаться;
- Type – обязательный атрибут. Поддерживаются значения ownProcess и shareProcess – поддержка единственной службы в исполняемом файле или нескольких;
- Account – учетная запись, от имени которой будет запускаться служба. Принимает значения LocalSystem, LocalService, NetworkService или имя учетной записи.

Используется в случае, когда атрибут ServiceType установлен в ownProcess. Если для учетной записи необходимо указать пароль, используется атрибут Password;

- Description – отображаемое в консоли управления описание службы;
- DisplayName – отображаемое в консоли управления наименование службы;
- ErrorControl – устанавливает способ реакции на ошибки с обязательной записью в журнал. Доступные значения: ignore – игнорирует ошибку, normal – отображает диалог и игнорирует ошибку, critical – инициирует перезагрузку системы с последней удачной конфигурацией;
- Interactive – принимая значения yes или no, устанавливает возможность взаимодействия службы с интерфейсом пользователя.

Следующий важный элемент – ServiceControl – определяет моменты запуска, останова и удаления службы. Связанная служба определяется с помощью обязательного атрибута Name, в который помещается имя службы в системе. Каждый из атрибутов Start, Stop и Remove может принимать одно из значений: install, uninstall, both. Соответствующее действие будет выполнено при установке пакета, при его удалении или в обоих случаях. Установка в yes атрибута Wait позволяет дождаться корректного завершения службы.

Кроме того, описанный в расширении WixUtilExtension элемент ServiceConfig позволяет установить поведение службы при отказах. Атрибуты FirstFailureActionType, SecondFailureActionType и ThirdFailureActionType могут принимать значения none, reboot, restart или runCommand и устанавливают поведение после первого, второго и всех последующих сбоев. Кроме того, устанавливаются:

- RestartServiceDelayInSeconds – количество секунд до перезапуска, если хотя бы одно из поведений установлено в restart;
- RebootMessage – отправляемое серверу сообщение перед перезапуском, если хотя бы одно из поведений установлено в reboot;
- ProgramCommandLine – выполняемая команда, если хотя бы одно из поведений установлено в runCommand.

Также для ServiceConfig можно задать значение свойства ResetPeriodInDays, определяющее количество дней до сброса счетчика ошибок.

```
      <Component Id="ServerNtServiceExe" Guid="????????-C357-4180-BADC-40435BD8E73C">
        <File Id="Server.NtService.exe" Name="Server.NtService.exe" KeyPath="yes"
Source="$(var.Server.NtService.TargetDir)" />
        <ServiceInstall Id="ServiceInstaller" Type="ownProcess" Vital="yes"
Name="DemoApplication.Server" DisplayName="Сервер доступа" Description="Обеспечивает обработку
запросов клиентов, взаимодействуя с БД." Start="auto" Account="LocalSystem"
ErrorControl="ignore" Interactive="no">
          <util:ServiceConfig FirstFailureActionType="restart"
SecondFailureActionType="restart" ThirdFailureActionType="restart"
RestartServiceDelayInSeconds="300" />
        </ServiceInstall>
        <ServiceControl Id="StartService" Start="install" Stop="both" Remove="uninstall"
Name="VTS.Server" Wait="yes" />
```

```
    </Component>
```

Поведение службы в случае отказов также может быть установлено с помощью описанных в стандартном пространстве имен элементов ServiceConfig, ServiceConfigFailureActions и Failure, но они требуют наличия MSI версии 5.0 или старше.

Осталось только упомянуть об элементе ServiceDependency, который позволяет задать имена служб, от которых зависит работоспособность текущей. Необходимое количество этих элементов помещается внутрь ServiceInstall, при этом в атрибут Id каждого помещается имя в системе необходимой службы.

Выпуск обновления

Создавая сложный продукт, который будет развиваться в течение длительного времени, необходимо задуматься о выпуске обновлений для него. Новые клиенты получат последние версии установочных пакетов, но нельзя забывать о тех, кто уже установил предыдущие версии. Для этого предназначены обновления и патчи. Отличие их заключается в том, что обновление является полноценной программой установки, а патч только обновляет часть уже установленной программы. Первый вариант более универсален, но увеличивает размер образа, что может быть неудобно в случае загрузки пакета по сети. Второй, напротив, создает файлы небольшого размера, так как включает только обновляемые элементы, но может быть установлен лишь поверх существующего продукта. Кроме того, при запуске как патча, так и неосновного (minor) обновления, необходимо дополнительно указать аргументы командной строки, что неудобно выполнять вручную. Вопрос запуска пакета с параметрами рассматривается в разделе, посвященном загрузчику (bootstrapper).

На сегодняшний день интеграция WiX в Visual Studio не поддерживает выпуск патчей, для их создания придется непосредственно воспользоваться утилитами командной строки. Более того, целесообразность их выпуска достаточно спорна – в данной книге рассматривается только вопрос подготовки обновлений.

Замечание: обновление возможно только при наличии атрибута UpgradeCode у элемента Product. Именно поэтому так важно всегда указывать его, даже если изначально обновление продукта не предусматривалось.

При создании любого обновления потребуется указать новое значение для атрибута Id элемента Package, поскольку его использует Windows Installer.

Возможен выпуск обновления, не изменяющего версию продукта, но такой подход практически идентичен выпуску неосновного обновления и может привести к путанице и ошибкам. Поэтому рекомендуется создавать обновления, увеличивающие номер версии продукта.

Неосновное обновление состоит в изменении номера версии – атрибута Version элемента Product. Оно обновляет существующий продукт и удобно во всех отношениях, за исключением одного: при необходимости произвести не установку, а обновление, пакету потребуется дополнительно передать параметры командной строки. Для обычного пользователя такой подход крайне неудобен и это одна из задач, решаемых с помощью рассматриваемого в соответствующем разделе загрузчика. Во всем остальном использование неосновного обновления является предпочтительным.

Замечание: для выполнения установки неосновного обновления необходимо запустить пакет из командной строки с аргументами:

msiexec.exe /fvomus [путь к msi-пакету]

Основное (major) обновление следует использовать в случаях, когда изменениям подвергается дерево наборов, из пакета удаляется компонент или необходимо предусмотреть наличие на компьютере одновременно старой и новой версии продукта. В этом случае необходимо установить новые значения для атрибутов Id и Version элемента Product. Для наглядности доступные варианты обновления сравниваются в таблице 7.1.

Таблица 7.1 Возможности неосновного (minor) и основного (major) обновлений.

	Неосновное	**Основное**
Установка продукта при его отсутствии	●	●
Обновление элементов (файлов и других ресурсов)	●	●
Возможность установки одновременно нескольких версий продукта		●
Отсутствие необходимости указания аргументов командной строки		●
Внесение изменений в дерево наборов		●
Изменение названия пакета		●
Удаление существующего компонента		●

Таким образом, основное обновление фактически представляет собой выпуск нового продукта с возможностью замены существующего, а неосновное – только механизм обновления и «чистой» установки.

В простейшем случае достаточно добавить в файл сценария элемент Upgrade, в атрибут Id которого помещается значение атрибута UpgradeCode элемента Product:

```
    <Upgrade Id="????????-36C9-46AE-BCDF-ABF03A5D140D" />
```

В таком виде обновление будет работать, но будет обновлять любые версии продукта, включая более новые и даже само себя. Поэтому доработаем пример так, чтобы он обновлял существующие версии, большие или равные 1.0.0, но меньшие 1.1.6:

```
    <Upgrade Id="????????-36C9-46AE-BCDF-ABF03A5D140D" >
      <UpgradeVersion OnlyDetect="no" Minimum="1.0.0" IncludeMinimum="yes" Maximum="1.1.6"
IncludeMaximum="no" Property="PREVIOUSFOUND" />
    </Upgrade>
```

В атрибутах Minimum и Maximum мы указываем наименьшую и наибольшую версии, при этом младшая версия включается в рассмотрение (IncludeMinimum=”yes”), а старшая – нет (IncludeMaximum=”no”). Атрибут OnlyDetect в нашем случае указывает на необходимость произвести обновление, если существующая версия продукта находится внутри указанного диапазона. Обязательный атрибут Property инициализирует указанное в нем свойство в случае, когда условие выполняется. Оно может быть использовано для определения конкретной версии продукта и связи с этим какой-либо дополнительной логики.

При выполнении обновления также могут учитываться языковые настройки пакетов, для чего следует использовать атрибут Language, в котором через запятую указываются идентификаторы языков. Если атрибут ExcludeLanguages установлен в yes, то рассматриваются все языки, кроме указанных в Language. Если же ExcludeLanguages установлен в no, то рассматриваются только языки, перечисленные в Language.

Установленный в yes атрибут IgnoreRemoveFailure позволяет игнорировать любые ошибки при удалении существующего продукта, а MigrateFeatures – переносить состояния установленных наборов из существующего продукта в обновляемый.

Также можно использовать элемент MajorUpgrade, упрощающий выполнение некоторых функций по обновлению. В самом простом виде он может выглядеть, как показано ниже, выдавая сообщение при попытке установить более раннюю версию поверх существующей:

```
    <MajorUpgrade DowngradeErrorMessage="Уже установлена более новая версия продукта"
Schedule="afterInstallInitialize" />
```

Данный элемент позволяет разрешить установку более старой версии – AllowDowngrades, установить версию с равным номером версии – AllowSameVersionUpgrades, запретить установку новой версии при наличии старой – Disallow, а также задать момент удаления старой версии с помощью атрибута Schedule.

Автоматическое обновление

Windows Installer предоставляет разработчикам развитые возможности по развертыванию приложений. Однако, в отличие от технологии ClickOnce, он не предоставляет функций, позволяющих распространять обновления. Задача распространения обновлений решается вне технологии Windows Installer.

Чтобы иметь возможность установить обновление, решение следует на два исполняемых файла: первый проверяет наличие обновлений и устанавливает при необходимости, затем запускает собственно программу. Существует несколько относительно универсальных реализаций, к которым можно отнести Updater Application Block от Pattern and Practices, AppLife Update, .NET Application Updater Component и другие, большая часть с открытыми исходными кодами. Конечно, адаптация такого решения к вашему проекту потребует усилий, но результат оправдает себя очень быстро.

В последнее время в собственных проектах я использую доработанный Simple Updater, который выложил Jarret Vance на сайте http://jvance.com. Простая система проверки наличия обновлений, позволяющая использовать ZIP-архивы, успешно работает уже не в одном решении.

Bootstrapper – загрузчик

При установке программы ее работоспособность, как правило, зависит от наличия в системе разнообразных компонентов. Прежде всего, для начала установки потребуется актуальная версия Windows Installer. Если приложение написано на управляемом коде, то для работы необходимо наличие той или иной версии .NET Framework. Этот список можно продолжать достаточно долго. Кроме того, некоторые пакеты в силу их размера целесообразно не включать в комплект поставки, а скачать при необходимости. Все эти действия могут быть достаточно многочисленны и необходимость их выполнения не вызовет энтузиазма у конечного пользователя. Для

автоматизации этого процесса и предназначен загрузчик – bootstrapper. Данное приложение не является частью Windows Installer, а служит посредником, позволяющим обойти некоторые ограничения технологии и упростить установку продукта для конечного пользователя. Кроме того, сам загрузчик должен зависеть от минимального количества внешних библиотек и обладать достаточно небольшим размером.

Замечание: сразу оговорюсь – термин «загрузчик» используется в этой книге ввиду отсутствия устоявшегося русскоязычного аналога для термина «Bootstrapper».

Таким образом, загрузчик - это приложение:

- способное определять наличие требуемых компонентов на целевой машине и, при необходимости, устанавливать их в правильной последовательности;
- способное скачивать отсутствующие пакеты при необходимости;
- достаточно компактное и с минимумом внешних зависимостей;
- обладающее возможностью настройки интерфейса для соответствия общему стилю.

Многие разработчики в течение длительного времени ожидали выхода утилиты Burn, в которой создатели WiX обещали реализовать многочисленные и весьма востребованные возможности. Данный инструмент доступен в Windows Installer XML, начиная с версии 3.6; но на момент завершения написания книги это еще недостаточно зрелое решение, чтобы иметь практическую ценность.

Кроме того, коммерческие пакеты для создания программ установки предлагают свои собственные загрузчики, некоторые из которых предоставляют достаточно интересную функциональность.

В комплект поставки Visual Studio входит Visual Studio Bootstrapper. Если вы работаете только в IDE, то неявно пользуетесь этим инструментом при создании проекта типа Setup Project в диалоговом окне Prerequisites, вызываемым из свойств проекта. Однако использовать Visual Studio Bootstrapper можно и из командной строки. Если вы используете именно его, рекомендую вам обратиться к статье, указанной в приложении.

Использование загрузчика dotNetInstaller

На сегодняшний день достаточно зрелым решением является свободно распространяемый пакет dotNetInstaller, доступный для скачивания по адресу http://dotnetinstaller.codeplex.com/. Он сочетает в себе достаточно развитый функционал и крайне привлекательную цену – продукт совершенно бесплатен. К минусам инструмента следует отнести отсутствие возможности работы из Visual Studio и поддержки Team Foundation Server.

Здесь мы рассмотрим несколько основных сценариев использования, которые позволят начать работу с данным пакетом. Важнейшим элементом пакета является приложение dotNetInstaller.exe, конфигурируемое с помощью XML-файла, и выполняющее все действия в процессе установки. Данное приложение написано на C++, не имеет внешних зависимостей, но добавит к получаемой программе установки около 1,3 Мб. Для подготовки файла конфигурации удобно использовать поставляющийся вместе с пакетом редактор InstallerEditor.exe, с которым мы и будем работать.

Мы последовательно рассмотрим следующие возможности:

- задание общих свойств и создание простой программы установки;
- внедрение msi-пакета внутрь сборки;
- проверка зависимостей от установленных компонентов;
- загрузка отсутствующих пакетов из сети;
- настройка интерфейса программы установки.

Общие свойства и создание простой программы установки

Начнем с простой программы установки, которая будет развертывать созданный в главе 3 установочный пакет для приложения Demo Application. На этом этапе выигрыша от использования загрузчика мы не извлечем, так как он будет выполнять те же действия, что и при непосредственном запуске msi-пакета, но описанные действия потребуются нам при подготовке более сложных сборок.

Создадим новый каталог и положим в него заранее подготовленный msi-пакет с именем SimpleSetupProject.msi. Там же будем размещать конфигурационный файл и все дополнительные ресурсы. Этот каталог назовем базовым путем приложения.

В случае запуска сборки из редактора базовый путь для внедряемых ресурсов будет совпадать с путем к каталогу, в котором находится утилита InstallerEditor.exe, а не файл конфигурации. Чтобы избежать этого, необходимо запускать компоновщик из командной строки и передавать ему базовый путь с помощью ключа /AppPath, либо в свойствах каждого внедряемого ресурса прописывать полный путь. Для запуска компоновщика проще использовать файл сценария, что позволяет передавать дополнительные параметры и экономить время. Минимальный набор включает в себя пути к необходимым файлам; здесь же целесообразно включить расширенный вывод с помощью ключа /Verbose+:

```
[Путь к утилитам]InstallerLinker.exe /AppPath:[Путь к каталогу с ресурсами]
/Output:Output\SimpleSetupProject.exe /Template:[Путь к утилитам]dotNetInstaller.exe
/Configuration:[Путь к каталогу с ресурсами]Configuration.xml /Verbose+
```

Перейдем к созданию программы установки. Запустим InstallerEditor.exe и создадим новый файл, воспользовавшись пунктом меню File -> New. Затем нажмем правой кнопкой мыши на корневом элементе Config File и выберем пункты Add -> Configurations -> Setup Configuration. Это предусматривает создание традиционной программы установки, распространяемой на съемных носителях или скачиваемой через сеть; данный вариант является более распространенным и на его примере может быть продемонстрирована вся доступная функциональность загрузчика.

Замечание: второй доступный вариант называется Web Configuration. Он позволяет создавать программу установки, распространяемую через сеть, где файл конфигурации не внедряется внутрь загрузчика, а скачивается с известного ресурса. После скачивания выполняются действия, описываемые в полученном файле. Такие программы проще сопровождать и обновлять, но для установки пользователю потребуется наличие доступа к сети. Данный вариант использует подмножество функций, доступных при создании традиционной программы установки.

К корневому элементу добавился узел с названием install. В нем мы будем указывать свойства, относящиеся ко всей программе установки. В текущей версии встроенная поддержка русского

языка отсутствует, поэтому нам придется выполнить локализацию вручную. В таблице 7.2 приведены свойства, значения которых были изменены с целью поддержки русского языка.

Таблица 7.2 Локализация свойств программы установки.

Название свойства	Новое значение	Примечание
Installing_component_wait	%s. Идет установка...	Обязательно наличие строки «%s» - будет заменено на название компонента
status_installed	(установлено)	
uninstalling_component_wait	%s. Идет удаление...	Обязательно наличие строки «%s» - будет заменено на название компонента
cancel_caption	Отмена	
dialog_caption		Название вашего продукта
dialog_message	В процессе установки будут настроены следующие компоненты:	
install_caption	Установить	
skip_caption	Пропустить	
uninstall_caption	Удалить	
failed_exec_command_continue	%s. Установка не была завершена	Обязательно наличие строки «%s» - будет заменено на название компонента.
Installation_completed	Программа успешно установлена	
installation_none	Программа уже установлена	
reboot_required	Для продолжения установки необходимо перезагрузить компьютер. Перезагрузить сейчас?	
Uninstallation_completed	Программа успешно удалена	
uninstallation_none	Установка не была произведена	

В таблице 7.3 приведено описание свойств, влияющих на поведение программы установки и их краткое описание.

Таблица 7.3 Свойства, управляющие поведением программы установки.

Название свойства	Значение по умолчанию	Описание
must_reboot_required	False	Останов программы в случае установки компонента, требующего перезагрузки.
dialog_show_installed	True	Отображение при инсталляции установленных ранее компонентов.
dialog_show_required	True	Отображение необходимых компонентов в списке без возможности их отключения.
dialog_show_uninstalled	True	Отображение при деинсталляции удаленных ранее компонентов.
allow_continue_on_error	True	Продолжение работы после ошибки при установке компонента. При любом значении будет отображен диалог с сообщением failed_exec_command_continue. При True есть возможность продолжить, иначе – выход с сообщением об ошибке. В зависимости от требуемого поведения установите текст свойства.
auto_close_in_installed	True	Закрытие программы в случае, если при ее запуске все компоненты уже установлены.
auto_close_on_error	False	Закрытие программы в случае ошибки при установке компонента.
auto_continue_on_reboot	False	Автоматическое продолжение установки после выполнения требуемой перезагрузки.
auto_start	False	Автоматическое начало установки без ожидания действий пользователя.
supports_install	True	Поддержка режима инсталляции.

supports_uninstall	True	Поддержка режима деинсталляции.
wait_for_complete_command	True	Необходимость ожидания завершения команды complete_command – дополнительное действие после окончания установки.
cab_path_autodelete	True	Автоматическое удаление временных пакетов, извлекаемых при установке.

Здесь мы не будем рассматривать создание программы установки, поддерживающей сразу несколько языков: она сводится к созданию отдельного узла install для каждого языка с дополнительным указанием значений для свойств из группы Language.

Установив основные свойства программы, мы получим рисунок, аналогичный приведенному на рисунке 7.1.

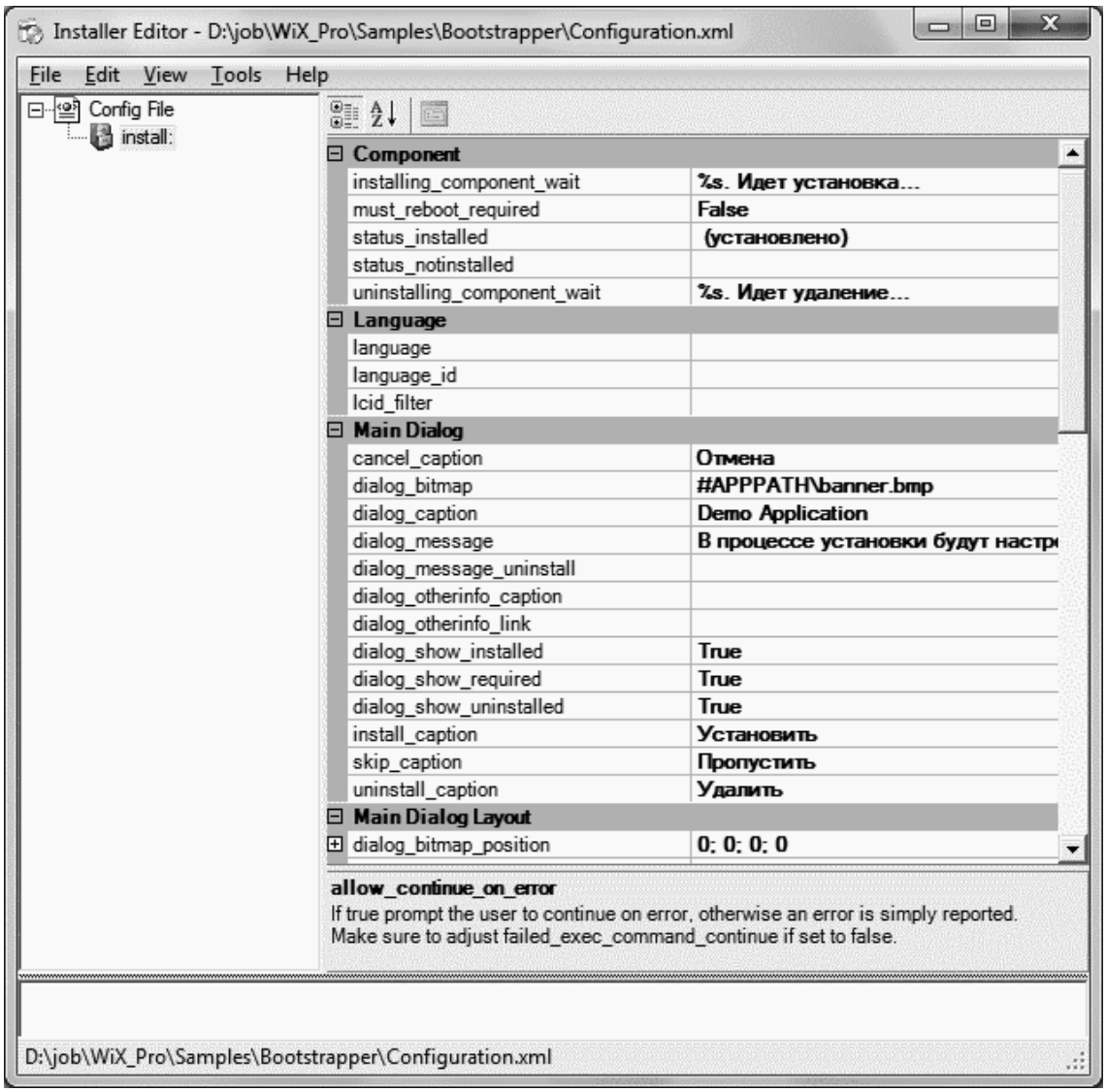

Рисунок 7.1 Основные свойства программы установки.

Установив общие свойства, перейдем к добавлению msi-пакета. Нажмем правой кнопкой мыши на элементе install и выберем пункты Add -> Components -> MSI Component. Для нового узла мы установим только несколько обязательных свойств: отображаемое имя (display_name), идентификатор (id) и путь к пакету (package). В качестве идентификатора можно использовать имя

пакета, но это не является обязательным. Местонахождение пакета, описываемое атрибутом package, в данном случае будет иметь вид #APPPATH\SimpleSetupProject.msi. Отметим, что в текущем виде нам потребуется поставлять msi-пакет вместе с программой установки, так как по умолчанию он не будет внедрен внутрь.

Здесь же можно задать ряд дополнительных свойств:

- статус компонента – необходим (required) и выбран по умолчанию (selected);
- проверка операционной системы и архитектуры процессора – os_filter_XXX и processor_architecture_filter;
- передача дополнительных параметров пакету при установке (cmdparameters) и удалении (uninstall_cmdparameters);
- необходимость перезагрузки и требуемое поведение при этом (must_reboot_required, mustreboot);
- возможность продолжения установки в случае ошибки (allow_continue_on_error).

Соберем и запустим программу установки.

Замечание: перед выполнением сборки не забывайте сохранять изменения, нажимая Ctrl+S или File -> Save. В противном случае результата работы вы можете не увидеть.

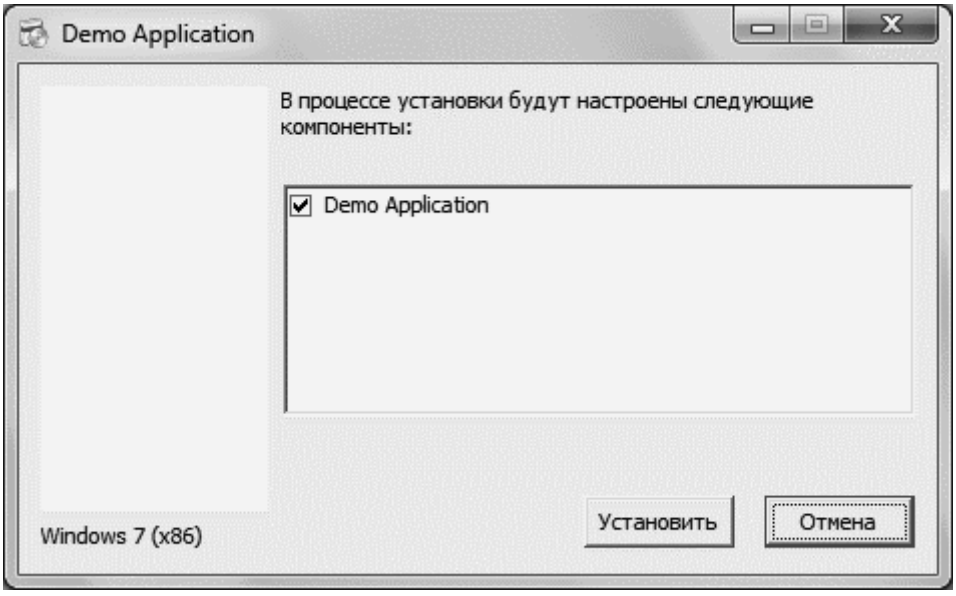

Рисунок 7.2 Простейшая программа установки.

Как видно на рисунке 7.2, полученный результат выглядит не слишком привлекательно, но уже сейчас мы можем настроить последовательную установку нескольких msi-пакетов, не заставляя пользователя контролировать очередность их установки.

Внедрение файла внутрь сборки

На предыдущем шаге мы сделали простую программу установки, которая представлена в виде двух файлов: SimpleSetupProject.exe и SimpleSetupProject.msi. Такой подход не доставляет неудобств, когда программа установки поставляется на отдельном носителе, допустим, DVD-диске

и запускается автоматически. Однако часто программа установки скачивается из сети и в этом случае существенно удобнее, когда она представлена в виде единственного исполняемого файла.

Доработаем наш проект и внедрим msi-пакет внутрь исполняемого файла. Допускается внедрение как отдельных файлов, так и каталогов. Нажмем правой кнопкой мыши на элементе, описывающем msi-пакет, и выберем пункты меню Add -> Embed -> Embed File. У данного элемента всего два свойства – sourcefilepath и targetfilepath. Первый указывает путь к внедряемому файлу, а второй позволяет указать путь во временной папке, куда будет извлекаться указанный файл в процессе установки. Установим для свойства sourcefilepath значение #APPPATH\SimpleSetupProject.msi.

Замечание: обратим внимание, что в данном случае используется относительный путь к файлу (#APPPATH\), который должен быть разрешен компоновщиком при сборке. Если при сборке вы получаете ошибку, в которой говорится об отсутствии внедряемого файла, вернитесь к началу раздела, где описывается создание файла сценария для запуска компоновки.

Ничто не мешает нам добавить элемент уровнем выше – непосредственно внутрь элемента install. Все ресурсы внедряются внутрь исполняемого файла и это только способ структурировать файл конфигурации.

Теперь нам осталось только исправить ссылку на файл пакета: атрибут package в элементе, описывающем msi-пакет. Заменим #APPPATH\SimpleSetupProject.msi на #CABPATH\SimpleSetupProject.msi. Переменная #CABPATH указывает каталог, куда извлекаются ресурсы в процессе установки. Это все, сохраним изменения и пересоберем проект. Теперь мы имеем несколько увеличенную в объеме, но полностью независимую программу установки, представленную в виде единственного файла SimpleSetupProject.exe.

Зависимости от сторонних компонентов

Нередко наши продукты для работы требуют различных установленных ранее пакетов. Так, разрабатываемые на управляемом коде приложения для работы требуют .NET Framework различных версий. Доработаем проект таким образом, чтобы он проверял наличие в системе пакета .NET Framework версии 3.5 и устанавливал его в случае необходимости.

Сначала нам потребуется распространяемый производителем пакет, выполняющий установку .NET Framework 3.5 (или другого компонента, в зависимости от наших потребностей). Как правило, данные пакеты доступны на сайте производителя, их также можно найти по фразе «redistributable package», например: «crystal reports redistributable package». Для .NET предоставляются на выбор различные варианты развертывания. Так, .NET 3.5 SP1 можно скачать в виде полного пакета, занимающего около 230 Мб, или в виде загрузчика размером около 3 Мб. Более того, последний умеет самостоятельно выполнять все проверки, поэтому проще всего встроить его в проект и передать ему управление. Так и поступим.

Поместим файл dotNetFx35setup.exe в каталог с другими ресурсами проекта. Откроем редактор и, нажав правой кнопкой мыши на узле install, выберем пункты меню Add -> Components -> Command Component. Данный элемент позволяет выполнить команду, в том числе и запускающую исполняемый файл. Набор свойств данного элемента практически повторяет свойства, устанавливаемые для msi-пакета. Нам потребуется установить свойства display_name и id. Свойству required присвоим True, так как данный пакет должен выполняться всегда, а

allow_continue_on_error – False. И, наконец, наиболее важное свойство – непосредственно текст команды в свойстве command. Заменим предлагаемое по умолчанию значение (cmd.exe /C "#APPPATH\mysetup.exe") на "#APPPATH\dotNetFx35setup.exe". Поскольку мы добавили узел Command после пакета msi, его вызов будет выполнен последним. Переместим его вверх. Нажмем на узле правой кнопкой мыши и в контекстном меню выберем Move -> Up.

Соберем проект и запустим его на выполнение. Файл dotNetFx35setup.exe нам потребуется положить в тот же каталог, где находится сама программа установки. Чтобы создать единую программу установки, воспользуемся внедрением файла, описанным в предыдущем разделе.

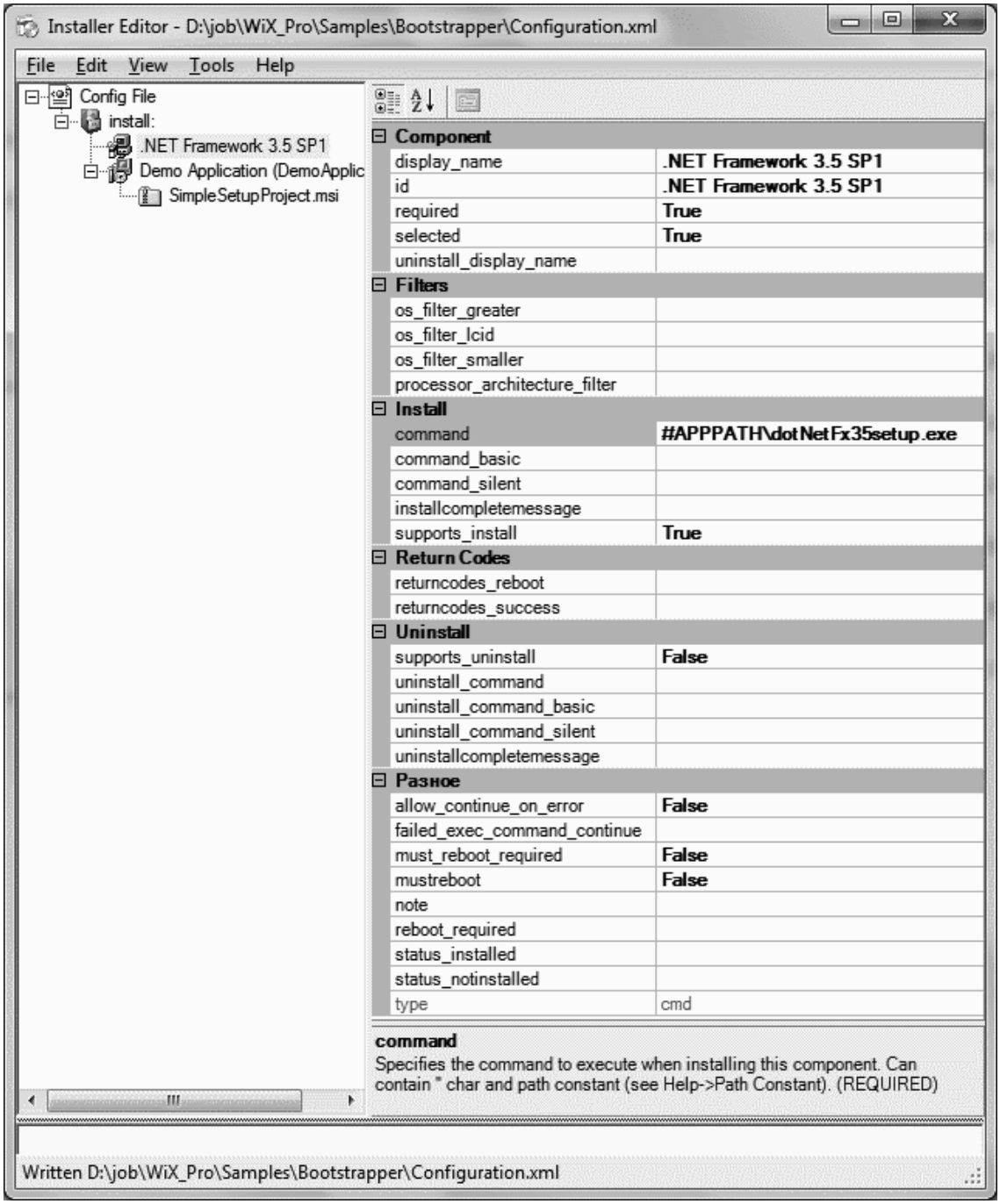

Рисунок 7.3 Свойства, используемые для внедрения разрешающего зависимость пакета.

В случае отсутствия загрузчика нам придется проверять факт установки пакета самостоятельно. dotNetInstaller предоставляет возможность выполнения таких проверок, опираясь на наличие записей в реестре, файлов, каталогов, поиск продуктов также может производиться по идентификаторам. Такая проверка потребует наличия дополнительных знаний о структуре устанавливаемого пакета. Так, например, наличие .NET Framework 3.5 SP1 на компьютере можно проверить по разделу реестра HKLM\SOFTWARE\Microsoft\NET Framework Setup\NDP\v3.5, используя ключи Version и SP. Ключ Version содержит строку с текущей версией, например 3.5.30729.4926, а SP – номер установленного пакета обновления, например 1.

Выполним простую проверку, опираясь на ключ SP, нас устроит значение большее или равное 1. Добавим проверку по ключу реестра. Нажмем правой кнопкой мыши на узле .NET Framework 3.5 SP1 и выберем пункты меню Add -> Checks -> Installed Check Registry. Выберем добавленный узел и установим необходимые свойства: rootkey – в HKEY_LOCAL_MACHINE, path – в HKLM\SOFTWARE\Microsoft\NET Framework Setup\NDP\v3.5, fieldname – в SP, fieldvalue – в 1. Поле comparision может принимать различные значения, позволяющие как проверять наличие ключей и значений, так и выполнять поиск подстрок и сравнение числовых значений. Мы используем вариант version_ge (greater or equal – больше или равен), возвращающий истинное значение для пакетов обновления 1 и старше. Снова соберем проект и запустим. На рисунке 7.4 видно, что на тестовой машине данный ключ найден, поэтому пакет в списке помечен как установленный – следовательно, повторную установку производить не потребуется.

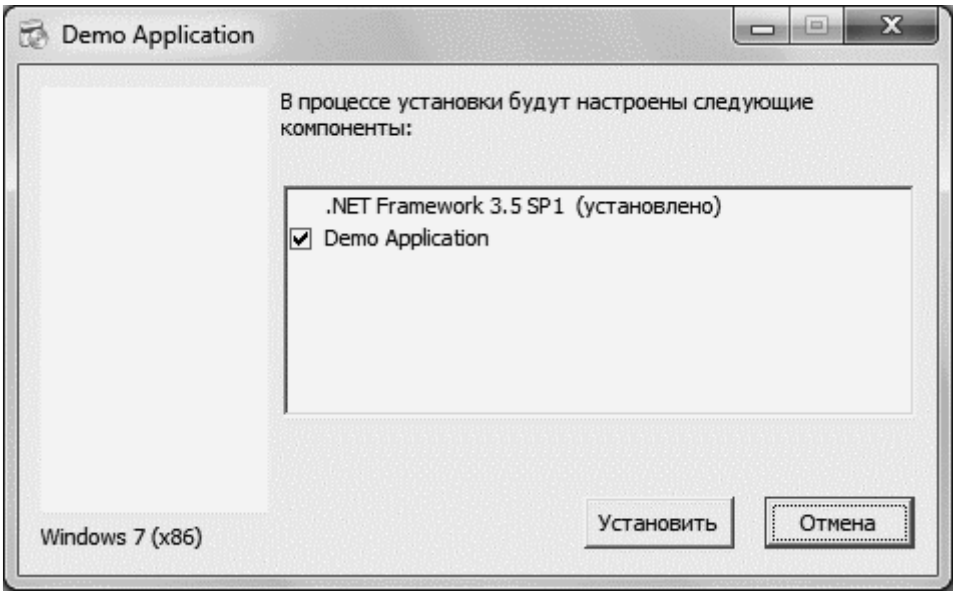

Рисунок 7.4 Отображение установленного компонента.

Загрузка отсутствующих пакетов из сети

Допустим, распространяемый пакет имеет достаточно большой размер. Более того, он может быть уже установлен на компьютере пользователя и нет необходимости поставлять его вместе с программой установки. Как пример – полный пакет установки .NET Framework 3.5 SP1 объемом около 230 Мб. На этот случай загрузчик имеет возможность скачивать отсутствующие пакеты из сети.

И снова доработаем предыдущий проект. Мы не будем поставлять исполняемый файл dotNetFx35setup.exe вместе с программой установки, а позволим загрузчику при необходимости скачать его с сайта производителя. Добавим диалог загрузки. Для этого нажмем правой кнопкой на узле .NET Framework 3.5 SP1 и выберем пункты меню Add -> Download -> Download Dialog. Для диалога установим значения элементов интерфейса в соответствии с рисунком 7.5.

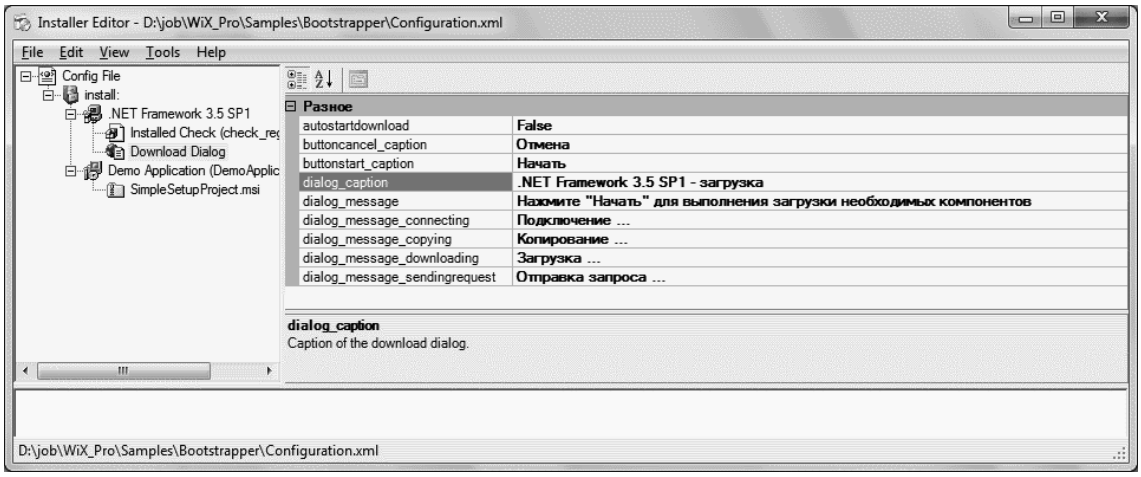

Рисунок 7.5 Свойства диалога загрузки файлов.

Для наглядности установим в false значение свойства autostartdownload – это позволит загрузчику задать нам дополнительный вопрос, а нам – увидеть в действии доступные возможности. Теперь добавим в данный диалог элемент, описывающий загружаемый файл. В контекстном меню элемента Download Dialog выберем пункты меню Add -> Download -> Download File. У этого элемента не так много свойств, основными из которых являются sourceurl, destinationpath и alwaysdownload. Первое содержит ссылку на загружаемый ресурс, второе – путь, куда будет помещен результат скачивания. Свойство alwaysdownload позволяет управлять поведением – если True, то загрузка будет выполнена даже при наличии файла в локальном каталоге. Установим для sourceurl значение http://download.microsoft.com/download/7/0/3/703455ee-a747-4cc8-bd3e-98a615c3aedb/dotNetFx35setup.exe; этот путь был найден на сайте производителя, но ничто не мешает нам указать адрес нашего собственного веб-сервера. Для destinationpath рекомендуется использовать подкаталог #TEMPPATH\, но для наглядности мы заменим его на #APPPATH. Обратите внимание, что указанный здесь путь также потребуется прописать в свойстве command элемента, выполняющего запуск исполняемого файла. Также укажем для свойства componentname значение .NET Framework 3.5 SP 1, оно будет отображаться в диалоге.

Дополнительно потребуется изменить свойства проверяемого ключа реестра или полностью удалить его, чтобы .NET Framework 3.5 SP1 считался неустановленным. Можно изменить значение свойства fieldvalue на 2 – в этом случае программа установки будет «ожидать» наличие второго или более старшего пакета обновления.

Соберем проект и запустим. В процессе работы мы увидим диалог загрузки, приведенный на рисунке 7.6, после чего будет произведена установка.

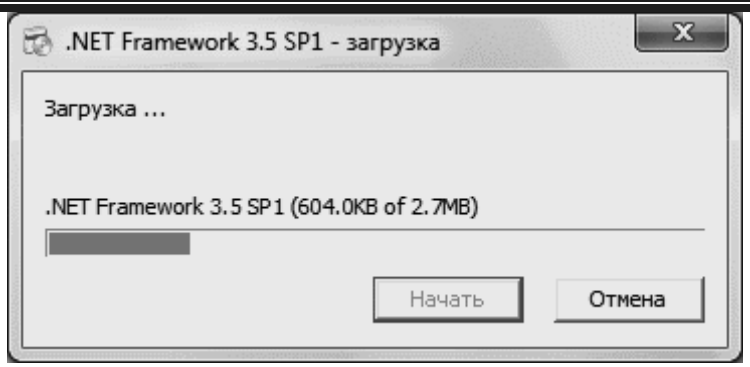

Рисунок 7.6 Скачивание отсутствующего пакета из сети.

Настройка интерфейса программы установки

Выше мы рассмотрели основные возможности, предоставляемые нам загрузчиком. Однако до передачи программы установки пользователям ей следует придать завершенный внешний вид. dotNetInstaller позволяет произвести некоторую настройку интерфейса, что мы и выполним сейчас. В редакторе нельзя увидеть внешний вид диалога, поэтому подгонку элементов интерфейса придется производить многократным перестроением пакета. Настраиваются положение и размер диалогового окна, кнопок и других стандартных элементов управления; есть возможность установить собственное изображение, а также добавить некоторые дополнительные элементы управления: статический текст, текстовое поле, флажок, гиперссылку и кнопку диалога открытия файла. Кроме того, можно добавить лицензионное соглашение.

В приведенном примере изменены значения некоторых параметров. Кроме того, для демонстрации заранее был подготовлен файл изображения Banner.bmp размером 135 на 360 пикселей, который будет отображаться в диалоговом окне слева. Если для какого-либо параметра оставлено нулевое значение, будет использоваться значение по умолчанию. Положение и размеры элементов управления приведены в таблице 7.4, значения приведены в формате (координата X; координата Y; ширина; высота).

Таблица 7.4 Размер и положение элементов управления в приведенном примере.

Название свойства	Значение	Описание
dialog_bitmap_position	1; 1; 135; 360	Основное изображение
dialog_cancel_button_position	380; 325; 100; 24	Кнопка «Отмена»
dialog_components_list_position	150; 70; 330; 140	Список устанавливаемых компонентов
dialog_install_button_position	270; 325; 100; 24	Кнопка «Установить»
dialog_message_position	150; 20; 0; 0	Поясняющая надпись (в нашем случае – «В процессе установки будут настроены…»)
dialog_osinfo_position	480; 1; 1; 1	Текст, отображающий версию операционной системы и тип процессора (скрыт за счет размера в 1 на 1 пиксель)
dialog_otherinfo_link_position	150; 230; 0; 0	Гиперссылка с дополнительной информацией (содержимое определяется свойствами dialog_otherinfo_caption и dialog_otherinfo_link)
dialog_position	0; 0; 500; 390	Диалог
dialog_skip_button_position	0; 0; 0; 0	Кнопка «Пропустить» (в примере не используется и не отображается)

Теперь добавим к программе установки файл изображения. Для этого поместим его в каталог с остальными ресурсами. Используемый файл указывается в свойстве dialog_bitmap элемента

install. Значение, устанавливаемое по умолчанию (#APPPATH\banner.bmp) устроит нас, так как созданный файл изображения имеет имя Banner.bmp.

Теперь изменим файл сценария, добавив в него ключ /Banner:

```
[Путь к утилитам]InstallerLinker.exe /AppPath:[Путь к каталогу с ресурсами]
/Output:Output\DemoApplicationSetup.exe /Template:[Путь к утилитам]dotNetInstaller.exe
/Configuration:[Путь к каталогу с ресурсами]Configuration.xml /Verbose+ /Banner:Banner.bmp
```

Если вы выполняете сборку из редактора, то можете указать путь к файлу изображения в диалоге, отображаемом при выборе пунктов меню File -> Create Exe… Вне зависимости от способа сборки, результат построения должен выглядеть, как показано на рисунке 7.7.

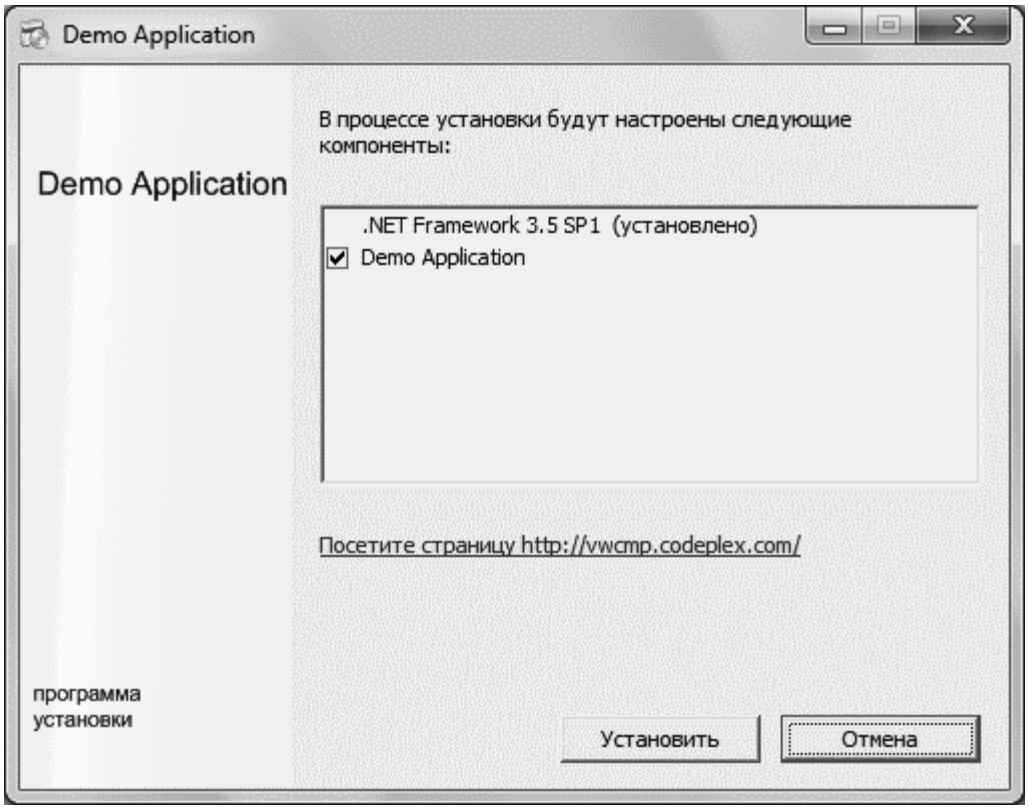

Рисунок 7.7 Измененный интерфейс программы установки.

Можно также дополнительно указать изображение, отображаемое при запуске в течение трех секунд (Splash Bitmap) – ключ /Splash, файл пиктограммы для программы установки – ключ /Icon и файл манифеста, если используемый по умолчанию вас не устраивает – ключ /Manifest.

Таким образом, предоставляемые пакетом dotNetInstaller возможности достаточно велики. Его использование позволяет снять ряд встроенных ограничений, накладываемых использованием технологии Windows Installer.

Анализ и декомпиляция msi-пакетов

Технология Windows Installer предоставляет в распоряжение разработчика огромный набор возможностей, но не всегда в документации легко найти описание необходимых функций. Как правило, интересный функционал можно встретить в дистрибутивах, созданных кем-то другим.

И в этой ситуации не может не радовать тот факт, что структура базы данных Windows Installer является открытой; кроме того, существуют замечательные инструменты, позволяющие изучать и изменять содержимое msi-сборок. Сначала мы рассмотрим основные возможности утилиты Orca, поставляемой в комплекте с Windows SDK, а затем научимся пользоваться утилитой Dark.exe – декомпилятором, входящим в состав Windows Installer XML.

Просмотр и модификация содержимого пакетов

Фактически пакеты Windows Installer представляют собой реляционную базу данных, где информация хранится в виде набора таблиц. Поставляемая в составе пакета Windows SDK утилита Orca позволяет читать просматривать и изменять содержимое сборки. Orca недоступна для отдельного скачивания, поэтому предварительно потребуется установить полный набор инструментов из Windows SDK. Как правило, инструменты могут быть найдены в каталоге «%ProgramFiles%\Microsoft SDKs\Windows\[Номер версии]\bin\». Там же находится и пакет Orca.msi, позволяющий установить указанный инструмент на свой компьютер.

Запустив утилиту, есть смысл сразу открыть какой-нибудь msi-пакет. Слева представлен полный список доступных таблиц, справа – содержимое выделенной таблицы. На рисунке 7.8 открыто содержимое таблицы Control. Рассмотрение структуры базы данных Windows Installer выходит за рамки данной книги, поэтому мы кратко рассмотрим основные и наиболее востребованные возможности утилиты при условии, что мы используем Windows Installer XML. Строго говоря, из всех инструментов, предназначенных для создания программ установки, WiX наиболее близок к структуре таблиц – соответствующие элементы схемы проецируются в процессе сборки на таблицы Windows Installer. В этом случае WiX упрощает работу, позволяя контролировать целостность данных на основании схемы, а также повышает производительность, так как один элемент может проецировать данные сразу на несколько связанных таблиц.

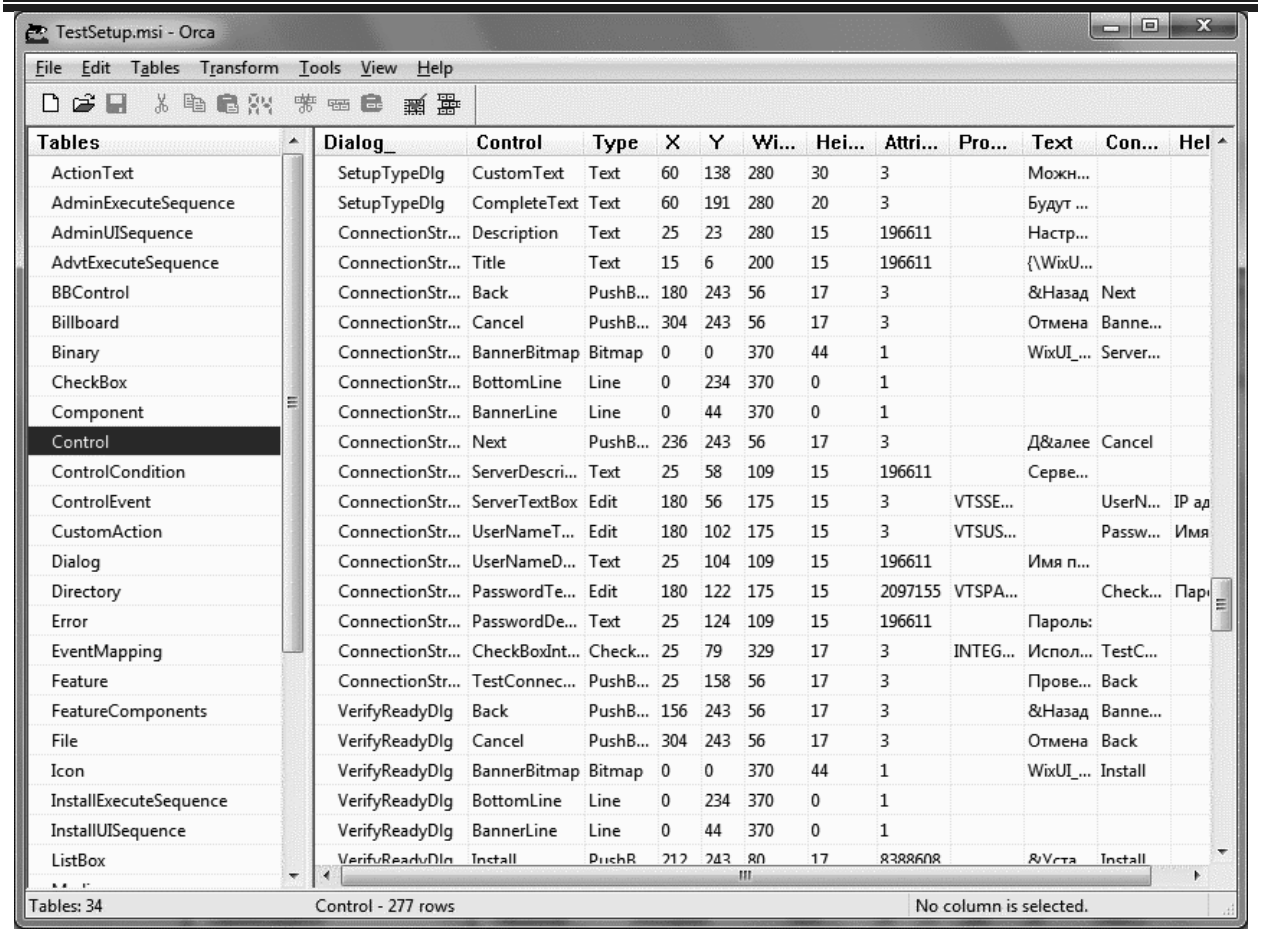

Рисунок 7.8 Анализ установочного пакета с помощью утилиты Orca.

Что именно может пригодиться нам при работе с Windows Installer XML? В первую очередь, мы можем увидеть значения локализованных ресурсов, отсутствующих при подготовке файла сценария, но подключаемых компоновщиком. Достаточно просмотреть содержимое таблиц данных для того, что увидеть случаи, требующие, например, замены предлагаемого по умолчанию значения. Здесь же мы можем увидеть имена двоичных ресурсов и даже извлечь их, но для этой цели лучше всего подходит декомпилятор, возможности которого мы рассмотрим ниже.

Также Orca предоставляет возможности по созданию преобразований (Transform), позволяющих модифицировать существующие пакеты, и просмотру результата применения патчей к ним. Но и эти задачи удобнее решать через декомпиляцию пакета. А вот какую способность данной утилиты сложно переоценить, так это возможность предварительного просмотра внешнего вида диалогов, включая отображение простых взаимосвязей между ними. Для вызова диалога предварительного просмотра предназначено меню Tools -> Dialog Preview… Мы увидим окно, похожее на приведенное на рисунке 7.9.

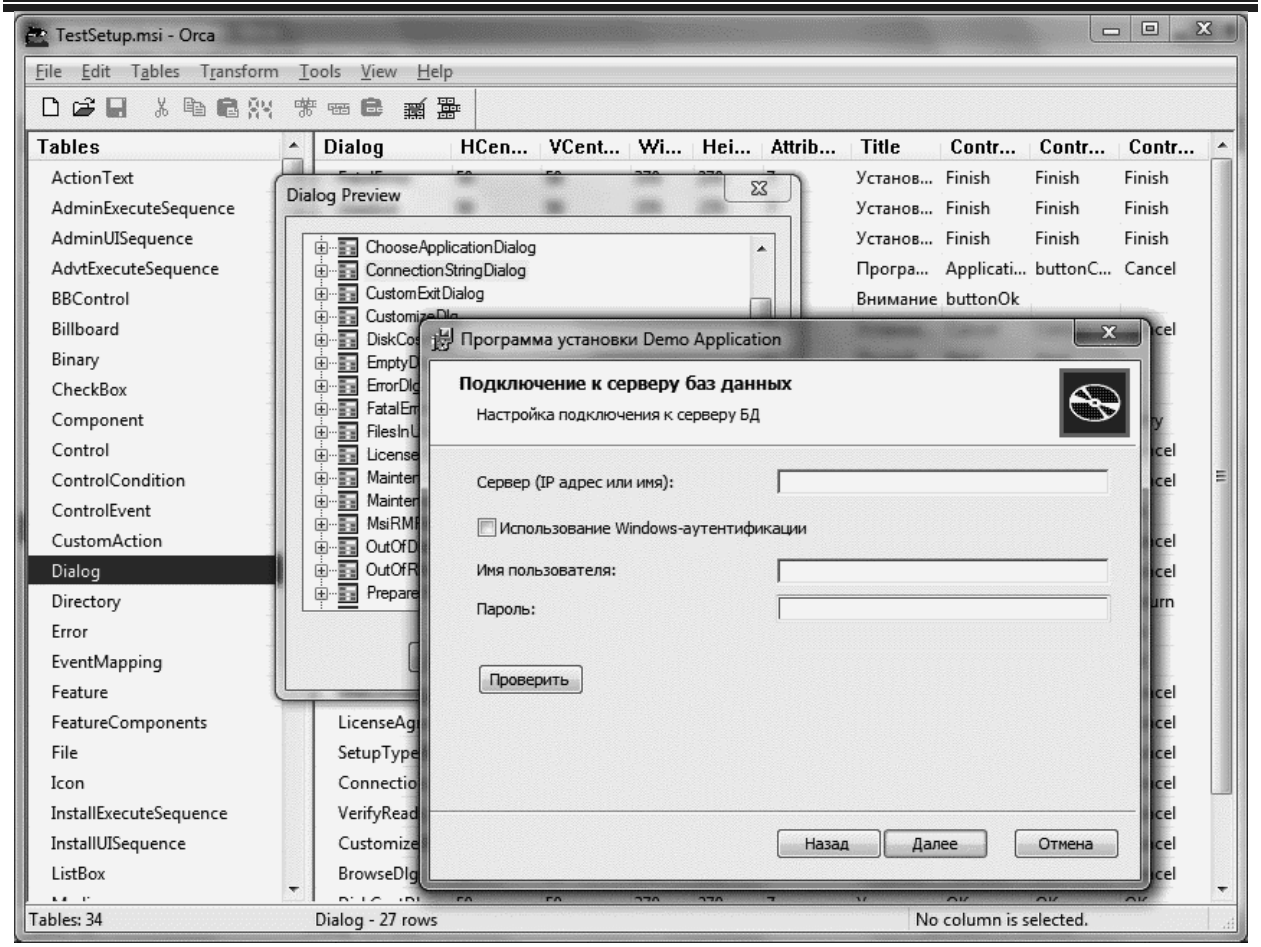

Рисунок 7.9 Просмотр интерфейса диалоговых окон.

Это позволяет просмотреть существующий пакет, не запуская его, и принять решение о заимствовании части интерфейсных решений. Подробнее об этом – в разделе, посвященном декомпиляции.

Декомпиляция пакетов

Наличие декомпилятора дает возможность при необходимости воспользоваться чужими наработками. Мы можем увидеть примеры использования различных функций, извлечь ресурсы и даже готовый интерфейс. Для декомпиляции пакета следует воспользоваться утилитой командной строки Dark.exe, генерирующей код в формате Windows Installer XML. Для преобразования необходимо указать путь к исходному пакету и имя создаваемого файл результата:

```
PS C:\Program Files\Windows Installer XML v3.5\bin> ./dark.exe D:\Samples\testsetup.msi -v -out c:\temp\generalsample.wxs
```

Мы также добавили ключ «-v», включающий отладочный вывод. Данный пример создаст выходной файл generalsample.wxs, но в него не будут помещены двоичные ресурсы. Для извлечения также и ресурсов дополнительно укажите параметр «-x» и путь к каталогу, где они будут сохранены:

```
PS C:\Program Files\Windows Installer XML v3.5\bin> ./dark.exe D:\Samples\testsetup.msi -v -out c:\temp\generalsample.wxs -x c:\temp\gsbinaries
```

Данную возможность следует применять осмысленно, так как извлечение всех ресурсов из крупных файлов может занять значительное время.

Перед декомпиляцией полезно предварительно проанализировать содержимое msi-пакета, открыв его с помощью Orca: именно там ресурсы удобно разложены по соответствующим таблицам. Можно получить огромный выигрыш во времени, извлекая из существующих пакетов изображения и описания целых диалогов. Конечно, исключительно в учебных целях.

Вопросы отладки

Даже самые простые установочные пакеты нередко содержат ошибки, а создание достаточно сложных программ установки непременно потребует известной доли отладки. В качестве самого простого примера скажем о том, что значения всех вычисляемых при установке свойств, за исключением закрытых, помещаются в журнал. В этом разделе рассмотрены только наиболее часто применимые вопросы, для получения дополнительной информации следует воспользоваться описываемыми в приложении материалами.

Включение ведения журналов

При создании msi-пакета проще всего запустить установку из командной строки. Параметр /l включает ведение журнала, а вместе с дополнительными аргументами выглядит как /l*vx, что позволяет получить наиболее подробный вывод, которым мы и пользуемся. Для пакета TestSetup.msi команда будет выглядеть как:

```
msiexec.exe /i TestSetup.msi /l*vx install.log
```

где результаты помещаются в файл install.log.

Чтение журналов Windows Installer

Получив файл журнала, мы становимся перед задачей чтения и интерпретации его содержимого. Прежде всего, процесс установки выполняется в двух различных процессах: в клиентском и серверном. Клиентский процесс взаимодействует с интерфейсом пользователя, извлекает значения свойств, а затем передает полученные значения выполняющему непосредственно установку серверному процессу. Указанное разделение не принципиально, но понимание используемых принципов лишним не будет.

Основная часть записей в журнале начинается с похожих конструкций:

```
MSI (c) (60:30) [19:16:13:320]: Baseline: New baseline 1.0.10 from transaction.

MSI (s) (34:F4) [19:16:13:539]: Baseline: Sorted order Native: Order 0.
```

Действия, выполненные клиентским процессом, отличаются наличием в записи конструкции «(c)», в то время как последовательность «(s)» описывает действие серверного процесса. Далее в круглых скобках указаны через двоеточие идентификаторы процесса и потока, создавшие данную запись. Причем от идентификаторов мы имеем только две последние шестнадцатеричные цифры.

Так, (34:F4) в нашем случае может означать процесс с ID 134 и поток с ID 2F4. Затем в квадратных скобках приведено время записи с точностью до миллисекунд.

Из журнала можно извлечь:

- содержание ошибок, произошедших в процессе установки;
- значения переменных и все их изменения;
- результаты вызова расширенных операций;
- сведения о запросах на перезагрузку, отмену установки и факт отката.

Наиболее важно научиться находить в журнале информацию о произошедших ошибках. При этом надо учесть, что существенная часть сообщений не является ошибками в привычном смысле, нарушающими процесс установки, а представляет собой информационные сообщения.

Рассмотрим пример:

```
MSI (c) (C0:6C) [22:17:24:953]: Note: 1: 1402 2: HKEY_CURRENT_USER\Software\Microsoft\MS Setup (ACME)\User Info 3: 2
```

После начального блока идет лексема Note, свидетельствующая о наличии дополнительного сообщения. Далее идут параметры, помеченные цифрами:

```
1: 1402

2: HKEY_CURRENT_USER\Software\Microsoft\MS Setup (ACME)\User Info

3: 2
```

Первый параметр всегда наиболее важен – это код ошибки, по которому можно получить ее описание. Полный перечень находится в библиотеке MSDN, проще всего его найти, набрав в строке поиска «windows installer error messages» или воспользоваться ссылкой, приведенной в приложении. Открываем указанную страницу и находим описание ошибки 1402:

```
Could not open key: [2]. System error [3].
```

Параметры уже пронумерованы, нам осталось собрать их вместе и получить результирующее сообщение:

```
Could not open key: HKEY_CURRENT_USER\Software\Microsoft\MS Setup (ACME)\User Info. System error 2.
```

Здесь также присутствует информация о некоторой системной ошибке с кодом 2. Узнать подробнее о ней можно, заглянув в файл Winerror.h (при наличии у вас Windows SDK), или набрав в командной строке:

```
PS C:\> net helpmsg 2

The system cannot find the file specified.
```

При наличии локализованной версии Windows, сообщение может быть отображено на знакомом языке:

```
Не удается найти указанный файл.
```

Таким образом, данное сообщение свидетельствует о невозможности открытия ключа реестра, так как он не был найден. Записи, касающиеся файлов и ключей реестра, Windows Installer делает однообразно. В зависимости от логики пакета установки данная ошибка может быть ожидаемой и обрабатываемой или недопустимой.

В случае использования расширенных операций (Custom Action) мы получим в журнале несколько отличающиеся записи. При отсутствии других операций записи в нем появятся отметки о результатах вызова операций. Если вы используете для разработки англоязычную версию Windows, то указанные записи можно найти по строке «Return value N.», русифицированная версия возвращает нам запись вида: «Код возврата N.», где N принимает значения:

- 0 – функция не может быть выполнена;
- 1, 5 – действие успешно завершено;
- 2 – пользователь отменил установку;
- 3 – ошибка;
- 4 – установка приостановлена, не завершена;
- 6 – дескриптор (handle) в некорректном состоянии;
- 7 – некорректные данные;
- 8 – уже выполняется другая установка.

Если мы получили код результата 3, необходимо вплотную заняться отладкой расширенной операции.

Использование утилиты WiLogUtl для обработки журналов

Файлы журнала могут иметь огромные размеры и регулярно проводить время за их чтением не слишком интересно. Для упрощения этой задачи предназначена утилита WiLogUtl из комплекта поставки Windows SDK. Найти ее можно в подкаталоге bin в папке, в которую установлен Windows SDK. Как правило, это путь «C:\Program Files\Microsoft SDKs\Windows\[номер версии]\bin\». Запускаем утилиту, нажимаем кнопку Browse и выбираем файл журнала, затем нажмем кнопку Analyze, как представлено на рисунке 7.10.

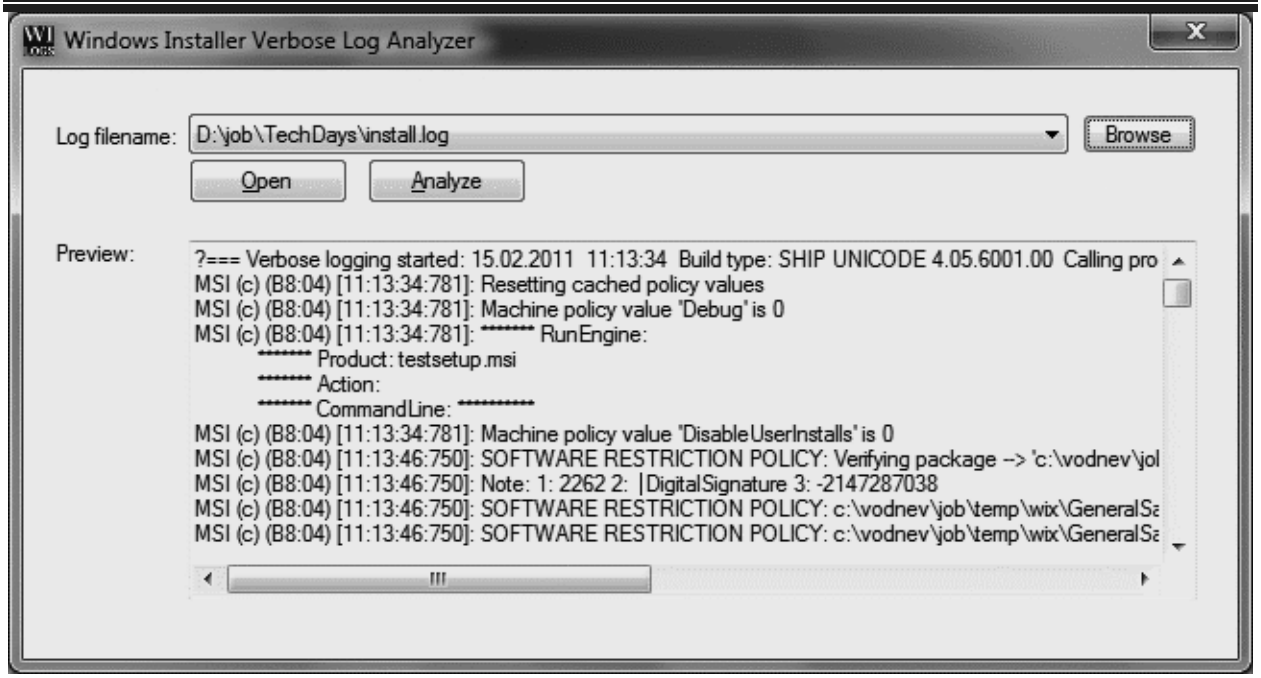

Рисунок 7.10 Открытие файла журнала с помощью утилиты WiLogUtl.

Открывается диалог, более насыщенный элементами управления (рисунок 7.11). Вверху перечислены все найденные в файле ошибки, между которыми можно перемещаться нажатием клавиш Previous и Next.

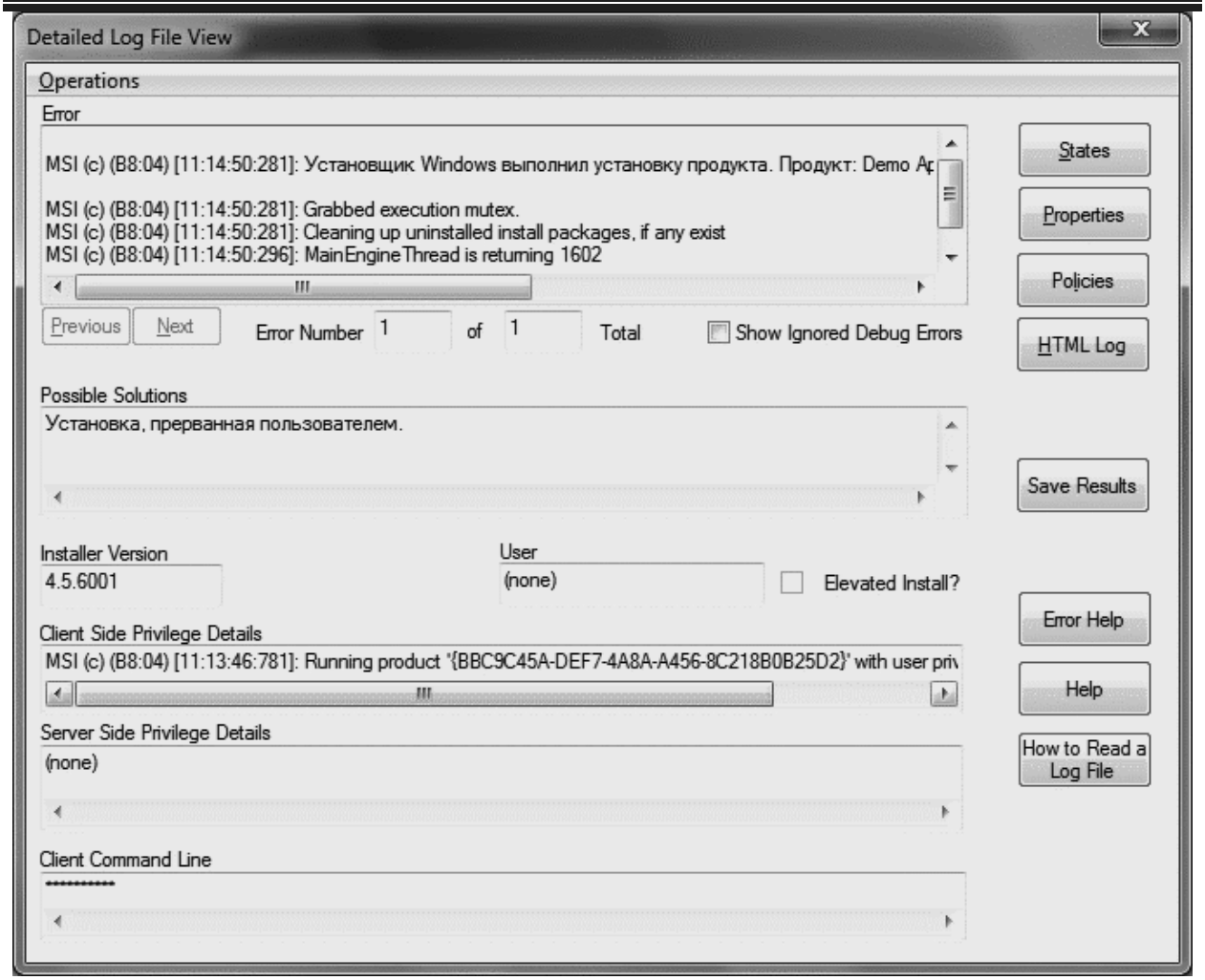

Рисунок 7.11 Основное окно анализа журналов.

Внизу справа располагаются кнопки Error Help, Help и How to Read a Log File. Первая выдает перечень ошибок Windows Installer, что удобно в случае отсутствия доступа к сети, вторая и третья открывают статьи, посвященные чтению файлов журнала и использованию данной утилиты.

Вверху справа располагаются кнопки States, Properties, Policies и HTML Log. States отображает диалог с информацией об установке наборов и компонентов, Properties показывает значения всех свойств, а Policies – текущие настройки Windows Installer. И, наконец, кнопка HTML Log создает удобный вывод журнала в виде web-страницы, где события подсвечены в зависимости от категорий, а также добавлены возможности навигации (рисунок 7.12).

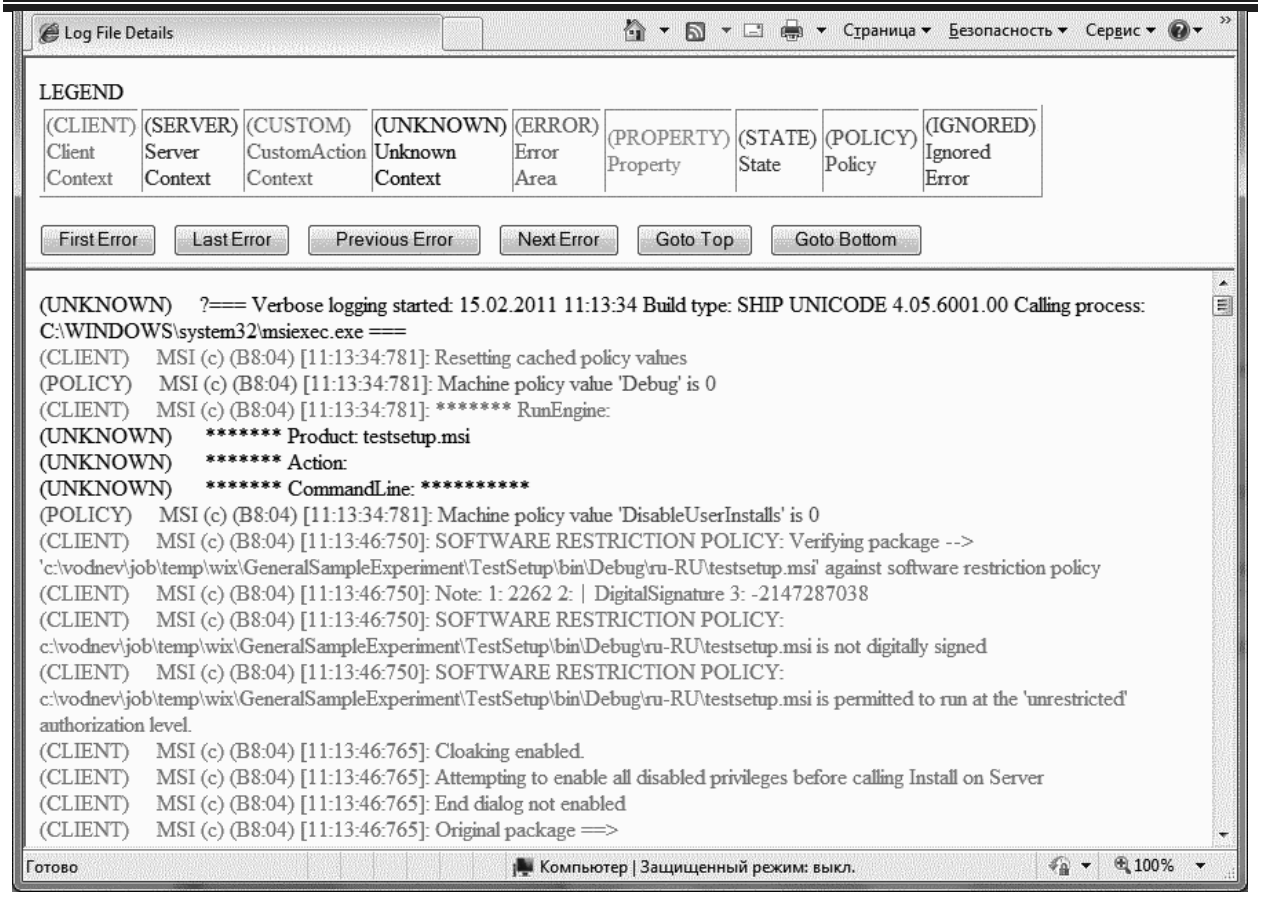

Рисунок 7.12 Форматированный вывод журнала событий.

Использование описанных выше подходов и утилиты WiLogUtl позволяет экономить время и усилия, направляемые на поиск и исправление ошибок.

Автоматизация сбора данных

Приложение может насчитывать сотни файлов. Если учесть рекомендации по наличию одного компонента на один развертываемый файл, то генерация для них кода потребует многократного повторения однообразных действий. В этом случае следует воспользоваться утилитой heat.exe, получившей свое название от слова Harvester. Этот инструмент выполняет автоматическую генерацию фрагментов кода, которые в дальнейшем могут быть использованы в тексте файла сценария.

Для обработки всех файлов в каталоге C:\FolderName\ и в его подкаталогах с выводом результата в файл Test.wxs следует ввести команду:

```
heat.exe dir C:\FolderName -out Test.wxs
```

Утилита может принимать значительное количество параметров, но основная их часть позволяет различным способом форматировать вывод или отключать его части. Достаточно полезным является дополнительное указание ключей –gg и –v. Первый включает автоматическую генерацию идентификаторов для создаваемых компонентов; второй – отладочный вывод.

Кроме каталогов, утилита позволяет обрабатывать отдельные файлы, счетчики производительности, содержимое веб-приложений IIS, а также результаты сборки проектов, создаваемых в Visual Studio.

Приложение.

Описание стандартных диалогов из расширения WixUIExtension

Название диалога	Описание диалога
AdvancedWelcomeEulaDlg	Отображает текст лицензионного соглашения. В отличие от LicenseAgreementDlg, данный диалог имеет кнопки «Расширенная» и «Установка» вместо «Далее» и «Назад». Используется в наборе WixUI_Advanced.
BrowseDlg	Позволяет пользователю произвести выбор каталог.
CancelDlg	Используется для подтверждения отмены процесса установки.
CustomizeDlg	Отображает дерево компонентов с информацией о выбранном наборе; позволяет изменить каталог установки.
DiskCostDlg	Отображает информацию об использовании дискового пространства для каждого тома.
ErrorDlg	Информация с сообщением об ошибке; предоставляет возможность повторить действие.
ExitDlg	Отображается после успешной установки.
FatalError	Информация о критической ошибке во время установки.
FeaturesDlg	Упрощенный диалог для выбора наборов для установки.
FilesInUse	Показывает список приложений, блокирующих обновляемые файлы. Содержит кнопки «Повтор», «Игнорировать» и «Выйти».
InstallDirDlg	Позволяет пользователю ввести в текстовом поле путь установки программы.
InstallScopeDlg	Используется для указания типа установки продукта: для всех пользователей или только для текущего.
InvalidDirDlg	Выводит сообщение об ошибке, если пользователь указывает некорректный каталог установки.
LicenseAgreementDlg	Отображает текст лицензионного соглашения.
MaintenanceTypeDlg	Содержит кнопки «Изменить», «Восстановить» и «Удалить», запускающие соответствующую функциональность для установленного продукта.
MaintenanceWelcomeDlg	Отображается при запуске программы установки в случае, когда продукт уже установлен.
MsiRMFilesInUse	Показывает список приложений, блокирующих обновляемые файлы. Позволяет закрыть их с последующим перезапуском или выполнить перезагрузку после окончания установки.
OutOfDiskDlg	Информирует пользователя о не достатке свободного места на диске.
OutOfRbDiskDlg	Диалог, похожий на OutOfDiskDlg. Дополнительно позволяет отключить функции отката Windows Installer в целях экономии дискового пространства, необходимого при установке.
PrepareDlg	Окно с индикатором прогресса, отображаемое во время инициализации программы установки.
ProgressDlg	Появляется во время установки. Отображает индикатор состояния и сообщения о текущих операциях.
ResumeDlg	Отображается при запуске программы установки при ее возобновлении.
SetupTypeDlg	Позволяет выбрать обычную, выборочную или полный установку.
UserExit	Отображается в случае отмены установки пользователем.
VerifyReadyDlg	Запрос подтверждения корректности данных. Отображается непосредственно перед началом установки.
WaitForCostingDlg	Отображается, когда не завершен расчет требований к дисковому пространству.
WelcomeDlg	Начальный диалог. Появляется при запуске программы установки продукта в первый раз.
WelcomeEulaDlg	Отображает текст лицензионного соглашения. Позволяет пользователю приступить к установке сразу после принятия соглашения. Используется только в наборе WixUI_Minimal.

Дополнительные ресурсы и материалы

Онлайн-руководство по WiX

http://www.tramontana.co.hu/wix/ - единственное на момент написания этой книги руководство по Windows Installer XML. Ориентировано на WiX версии 2, но основная часть материала применима и к версии 3+. Описывает работу с утилитами WiX из командной строки без использования IDE. На английском языке.

Справочные материалы из библиотеки MSDN

Основная часть справочных материалов находится в англоязычных разделах библиотеки MSDN, посвященных службам Windows Installer. Они располагаются по адресу: http://msdn.microsoft.com/en-us/library/cc185688(VS.85).aspx

Описание предустановленных свойств Windows Installer:

http://msdn.microsoft.com/en-us/library/aa370905(VS.85).aspx

Идентификаторы языковых культур и соответствующих кодовых страниц:

http://msdn.microsoft.com/en-us/library/aa369771(VS.85).aspx

Чтение журналов Windows Installer

Статья на английском языке, в которой рассматривается работа с журналами, генерируемыми Windows Installer. Обсуждаются вопросы включения журналирования: с использованием аргументов командной строки, через групповые политики, свойства или через Installer API. В остальной части статьи описывается анализ полученных файлов.

http://blogs.technet.com/richard_macdonald/archive/2007/04/02/How-to-Interpret-Windows-Installer-Logs.aspx

Описание ошибок Windows Installer:

http://msdn.microsoft.com/en-us/library/aa372835(VS.85).aspx

Результаты вызова расширенных операций:

http://msdn.microsoft.com/en-us/library/aa369778(VS.85).aspx

Bootstrapper из комплекта поставки Visual Studio

Статья из журнала MSDN Magazine за октябрь 2004 года. На русском языке. Если вы создаете программы установки с использованием стандартных средств Visual Studio – читать обязательно.

http://www.microsoft.com/Rus/Msdn/Magazine/2004/10/VS_2005.mspx

Онлайн-доклады на русском языке

Сайт http://www.techdays.ru содержит большое количество технических докладов, посвященных разработке и эксплуатации программного обеспечения, в том числе несколько – о создании установочных пакетов с помощью Windows Installer XML. Достаточно выполнить поиск по сайту по ключевому слову «wix».

Об авторе

Евгений Воднев – профессиональный разработчик, специалист в области создания и внедрения распределенных систем. Обладает статусами Microsoft Certified Application Developer, Microsoft Certified Professional Developer (Windows Developer, Windows Developer 3.5). Автор докладов по Windows Installer XML и облачным технологиям. Все вопросы, замечания и предложения по данной книге можно присылать на адрес e.vodnev@mail.ru.