

50

Платформа Managed Add-In Framework

В ЭТОЙ ГЛАВЕ...

- Архитектура управляемых дополнений (Managed Add-In Framework)
- Определение контракта
- Реализация конвейера
- Создание дополнений
- Хостинг дополнений

В этой главе подробно рассматривается архитектура управляемых дополнений (Managed Add-In Framework — MAF), а также создание и хостинг дополнений с использованием MAF. Вы узнаете о платформе MAF и проблемах, решаемых посредством размещения дополнений, в том числе о поддержке версий, обнаружении, активизации и изоляции. Затем будут описаны шаги, необходимые для создания конвейера MAF, используемого для подключения дополнения к хосту, а также показано, как строить дополнения и создавать размещаемые приложения, которые используют дополнения.

Архитектура MAF

В версии .NET 4 реализованы две технологии для создания и использования дополнений. В главе 28 рассказывалось о создании дополнений с помощью Managed Extensibility Framework (MEF). Во время выполнения приложения, использующие MEF, могут находить дополнения в каталоге или в сборке и подключать их, используя атрибуты. Отличие от MAF состоит в том, что MEF не предлагает отделения дополнения от размещаемого приложения с помощью доменов приложений или других процессов. И здесь вступает в действие MAF. Платой за достижение упомянутого разделения в MAF будет дополнительная сложность. Чтобы воспользоваться преимуществами обеих технологий — MEF и MAF, — их можно комбинировать. Разумеется, это также принесет сложность.

При создании приложения, позволяющего добавлять дополнения во время выполнения, приходится сталкиваться с определенными сложностями, например, как находить дополнения и каким образом решать проблемы совместимости версий, когда размещаемое приложение и дополнение развиваются независимо друг от друга. В этом разделе вы узнаете кое-что о них и способах их преодоления, предусмотренных в архитектуре MAF.

При создании размещаемого приложения, которое динамически загружает сборки, добавляемые позднее, возникает ряд сложностей, которые приходится преодолевать (табл. 50.1).

Таблица 50.1. Сложности, связанные с дополнениями

Сложность	Описание
Обнаружение	Как размещаемое приложение может находить дополнения? Существует несколько вариантов. Один из них — добавление информации о дополнении в конфигурационный файл приложения. Недостаток этого подхода в том, что установка нового дополнения требует внесения изменений в существующий конфигурационный файл. Другой вариант заключается в простом копировании сборки, содержащей дополнение, в предопределенный каталог с чтением информации о нем средствами рефлексии. О рефлексии рассказывается в главе 14.
Активизация	С динамическими загружаемыми сборками невозможно просто использовать операцию <code>new</code> для создания экземпляра. Такие сборки можно создавать с помощью класса <code>Activator</code> . Также различные опции активизации могут применяться, если дополнение загружено внутри другого домена приложений или нового процесса. Сборки и домены приложений описаны в главе 18.
Изоляция	Дополнение может нарушить работу размещаемого приложения, как вы, вероятно, уже видели на примере Internet Explorer, терпящего крах из-за различных дополнений. В зависимости от типа размещаемого приложения и способа интеграции дополнения, оно может быть загружено внутри другого домена приложения или также внутри другого процесса.

Сложность	Описание
Жизненный цикл	Очистка объектов — работа для сборщика мусора. Однако сборщик мусора здесь не поможет, поскольку дополнения могут быть активизированы в другом домене приложений или в другом процессе. Другой способ удержания объекта в памяти — счетчик ссылок или механизм аренды и спонсирования.
Поддержка версий	Поддержка версий является значительной сложностью для дополнений. Обычно допускается, что новая версия размещаемого приложения может загружать старые дополнения, а старое приложение может иметь опцию для загрузки новых дополнений.

Теперь давайте рассмотрим архитектуру MAF, и то, как платформа решает перечисленные проблемы. При проектировании платформы MAF преследовались следующие цели:

- дополнения должны легко разрабатываться;
- нахождение дополнений во время выполнения должно быть производительным;
- разработка размещаемых приложений должна быть также простой, но не настолько простой, как разработка дополнений;
- дополнение и размещаемое приложение должны развиваться независимо.

Платформа MAF решает проблемы, используя конвейер, в основе которого заложены контракты. В главе речь пойдет о том, как MAF справляется с обнаружением дополнений, каким образом дополнения активируются, как они поддерживаются в актуальном состоянии и каким образом осуществляется поддержка версий.

Конвейеры

Архитектура MAF основана на конвейере из семи сборок. Этот конвейер позволяет решать проблемы поддержки версий дополнений. Поскольку сборки, участвующие в конвейере, имеют между собой очень слабую зависимость, можно обеспечить совершенно независимое развитие версий контракта, хостинга и дополнений.

На рис. 50.1 показан конвейер, лежащий в основе архитектуры MAF. В центре находится сборка контракта. Эта сборка содержит интерфейс контракта, перечисляющий методы и свойства, которые должны быть реализованы дополнением и могут быть вызваны хостом. Левая часть контракта — это сторона хоста, а правая — сторона дополнения. На рисунке можно видеть зависимости между сборками. Сборка хоста, показанная слева, не имеет реальной зависимости от сборки контракта; то же касается и сборки дополнения. Ни та, ни другая в действительности не реализуют интерфейса, определенного контрактом. Вместо этого они просто имеют ссылку на сборку представления. Размещаемое приложение ссылается на представление хоста; дополнение ссылается на представление дополнения. Сами представления содержат абстрактные классы представлений, которые определяют методы и свойства, как они определены контрактом.



Рис. 50.1. Конвейер MAF

На рис. 50.2 представлены отношения между классами конвейера. Класс хоста имеет ассоциацию с абстрактным представлением хоста и вызывает его методы. Абстрактный класс представления реализован адаптером хоста. Адаптеры осуществляют соединение между представлением и контрактом. Адаптер дополнения реализует методы и свойства контракта. Этот адаптер содержит ссылку на представление дополнения и переадресует вызовы со стороны хоста к представлению дополнения. Класс адаптера хоста определяет конкретный класс, унаследованный от абстрактного базового класса представления хоста, реализующего его методы и свойства. Этот адаптер включает ссылку на контракт для переадресации вызовов от представления к контракту.

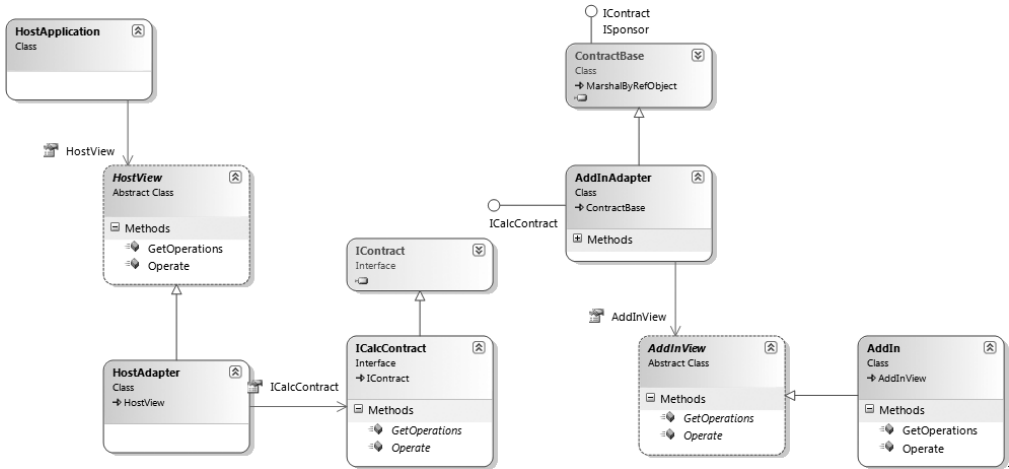


Рис. 50.2. Отношения между классами конвейера

Эта модель обеспечивает независимость между стороной дополнения и стороной хоста. Адаптации подлежит только уровень отображения. Например, если появляется новая версия хоста, которая использует совершенно новые методы и свойства, то контракт остается прежним и только адаптер подлежит изменению. Также можно определить новый контракт. Адаптеры могут изменяться, или несколько контрактов могут использоваться параллельно.

Обнаружение

Каким образом размещаемое приложение может находить новые дополнения? Архитектура MAF использует предопределенную структуру каталогов для нахождения дополнений и других сборок конвейера. Компоненты конвейера должны располагаться в следующих каталогах:

- HostSideAdapters
- Contracts
- AddInSideAdapters
- AddInViews
- AddIns

Все эти каталоги, за исключением AddIns, непосредственно содержат сборку определенной части конвейера. Каталог AddIns содержит подкаталоги для каждой сборки дополнения. Дополнения также можно хранить в каталогах, полностью независимых от других компонентов конвейера.

Сборки конвейера не загружаются динамически для получения всей информации о дополнении с помощью рефлексии. Для многих дополнений это значительно бы увеличило время запуска размещаемого приложения. Вместо этого MAF использует кэш с информацией о компонентах конвейера. Этот кэш создается программой, устанавливающей дополнение, или размещаемым приложением, если оно имеет доступ на запись в каталог конвейера.

Кэшированная информация о компонентах конвейера создается вызовами методов класса `AddInStore`. Метод `Update()` находит новую сборку, которая еще не указана в списке хранилища. Метод `Rebuild()` перестраивает полностью двоичный файл хранилища с информацией о дополнениях. В табл. 50.2 перечислены методы класса `AddInStore`.

Таблица 50.2. Методы класса `AddInStore`

Метод	Описание
<code>Rebuild()</code> <code>RebuildAddIns()</code>	Метод <code>Rebuild()</code> перестраивает кэш со всеми компонентами конвейера. Если дополнение хранится в другом каталоге, то метод <code>RebuildAddIns()</code> может быть использован для перестройки кэша дополнений.
<code>Update()</code> <code>UpdateAddIns()</code>	В то время как метод <code>Rebuild()</code> перестраивает целиком кэш конвейера, метод <code>Update()</code> просто обновляет кэш информацией о новых компонентах конвейера. Метод <code>UpdateAddIns()</code> обновляет только кэш дополнений.
<code>FindAddIn()</code> <code>FindAddIns()</code>	Эти методы используются для нахождения дополнений с использованием кэша. Метод <code>FindAddIns()</code> возвращает коллекцию всех дополнений, которые соответствуют представлению хоста. Метод <code>FindAddIn()</code> возвращает определенное дополнение.

Активизация и изоляция

Метод `FindAddIns()` класса `AddInStore` возвращает коллекцию объектов `AddInToken`, представляющую дополнение. Через класс `AddInToken` можно получить доступ к информации о сборке, такой как имя, описание, издатель и версия. Вы можете активизировать дополнение методом `Activate()`. В табл. 50.3 перечислены свойства и методы класса `AddInToken`.

Таблица 50.3. Свойства и методы класса `AddInToken`

Член	Описание
<code>Name</code> <code>Publisher</code> <code>Version</code> <code>Description</code>	Свойства <code>Name</code> , <code>Publisher</code> , <code>Version</code> и <code>Description</code> класса <code>AddInToken</code> возвращают информацию о дополнении, которая назначается дополнению с помощью атрибута <code>AddInAttribute</code> .
<code>AssemblyName</code>	<code>AssemblyName</code> возвращает имя сборки, содержащей дополнение.
<code>EnableDirectConnect</code>	В свойстве <code>EnableDirectConnect</code> можно установить значение, указывающее на то, что хост должен непосредственно подключаться к дополнению вместо использования компонентов конвейера. Это возможно только в том случае, если дополнение и хост запускаются в одном домене приложений, и типы и представления дополнения и представления хоста совпадают. К тому же требуется присутствие всех компонентов конвейера.

Член	Описание
<code>QualificationData</code>	Дополнение может пометить домен приложения и требования безопасности атрибутом <code>QualificationDataAttribute</code> . Дополнение может перечислять требования к безопасности и изоляции. Например, <code>[QualificationData("Isolation", "NewAppDomain")]</code> означает, что дополнение должно быть размещено в новом процессе. Можно прочитать эту информацию из <code>AddInToken</code> для активизации дополнения со специфицированными требованиями. Вдобавок к требованиям домена приложений и безопасности, этот атрибут можно использовать для передачи специальной информации по конвейеру.
<code>Activate()</code>	Дополнение активизируется методом <code>Activate()</code> . С помощью параметров этого метода можно определить, должно дополнение загружаться внутри нового домена приложения или нового процесса. Можно также указать полномочия, которые получит дополнение.

Одно дополнение может разрушить все приложение. Вам наверняка приходилось видеть, как терпит крах Internet Explorer из-за сбоя дополнения. В зависимости от типа приложения и типа дополнения этого можно избежать, позволив дополнению работать в другом домене приложений или внутри другого процесса. Новый домен приложения может также иметь ограниченные полномочия.

Метод `Activate()` класса `AddInToken` имеет несколько перегрузок, которым можно передавать среду, куда должно загружаться дополнение. Допустимые варианты перечислены в табл. 50.4.

Таблица 50.4. Параметры метода `AddInToken.Activate()`

Параметр	Описание
<code>AppDomain</code>	Можно передать новый домен приложения, в который должно быть загружено дополнение. Подобным образом его можно сделать независимым от размещаемого приложения и, кроме того, оно может быть затем выгружено этим доменом приложения.
<code>AddInSecurityLevel</code>	Если дополнение должно быть запущено с другими уровнями безопасности, можно передать значение перечисления <code>AddInSecurityLevel</code> . Возможные значения — <code>Internet</code> , <code>Intranet</code> , <code>FullTrust</code> и <code>Host</code> .
<code>PermissionSet</code>	Если предопределенные уровни безопасности недостаточно специфичны, можно также присвоить <code>PermissionSet</code> домену приложения дополнения.
<code>AddInProcess</code>	Дополнение также может быть запущено внутри другого процесса размещаемого приложения. Методу <code>Activate()</code> необходимо передать новый процесс <code>AddInProcess</code> . Новый процесс может завершаться, когда все дополнения выгружены, или продолжать выполнение. Это устанавливается с помощью свойства <code>KeepAlive</code> .
<code>AddInEnvironment</code>	Передача объекта <code>AddInEnvironment</code> — еще один вариант определения домена приложения, куда должно быть загружено приложение. Конструктору <code>AddInEnvironment</code> можно передать объект <code>AppDomain</code> . Можно также получить существующий <code>AddInEnvironment</code> дополнения из свойства <code>AddInEnvironment</code> класса <code>AddInController</code> .



Домены приложений описаны в главе 18.

Тип приложения может ограничивать доступный выбор. Дополнения WPF не поддерживают пересекающиеся процессы. В Windows Forms невозможно иметь элементы управления Windows, подключаемые из разных доменов приложений.

Ниже перечислены действия, выполняемые при вызове метода `Activate()` объекта `AddInToken`.

1. Создается домен приложения с указанными полномочиями.
2. Сборка дополнения загружается в новый домен приложения методом `Assembly.LoadFrom()`.
3. Средствами рефлексии вызывается конструктор дополнения по умолчанию. Поскольку дополнение унаследовано от базового класса, который определен в представлении дополнения, сборка представления также загружается.
4. Конструируется экземпляр адаптера стороны дополнения. Экземпляр дополнения передается конструктору адаптера, так что адаптер может подключить контракт к дополнению. Адаптер дополнения унаследован от базового класса `MarshalByRefObject`, так что он может вызываться через границы доменов приложений.
5. Код активизации возвращает прокси-объект адаптеру дополнения в домен размещаемого приложения. Поскольку адаптер дополнения реализует интерфейс контракта, прокси-объект содержит методы и свойства интерфейса контракта.
6. Конструируется экземпляр адаптера стороны хоста в домене размещаемого приложения. Конструктору передается прокси-объект адаптера стороны дополнения. Активизация находит тип адаптера стороны хоста из маркера дополнения.

Адаптер стороны хоста возвращается размещаемому приложению.

Контракты

Контракты определяют границы между стороной хоста и стороной дополнения архитектуры MAE. Контракты определяются интерфейсом, который должен наследоваться от базового интерфейса `Contract`. Контракт должен быть тщательно продуман, чтобы обеспечить необходимую гибкость сценариев применения дополнений.

Контракты не имеют версий и могут не изменяться, так что предыдущие реализации дополнения могут запускаться на более новых хостах. Новые версии создаются определением нового контракта.

Существуют некоторые ограничения типов, которые разрешено использовать с контрактом. Они обусловлены сложностями, связанными с версиями, а также пересечением доменов приложений хоста и дополнения. Типы должны быть безопасными и согласованными по версиям, а также иметь возможность преодолевать границы (доменов приложений или процессов), чтобы передаваться между хостами и дополнениями.

Допустимые типы, которые могут передаваться с контрактом:

- примитивные типы;
- другие контракты;
- сериализуемые системные типы;
- простые сериализуемые специальные типы, состоящие из примитивных типов или контрактов, которые могут не иметь реализации.
- Члены интерфейса `Contract` описаны в табл. 50.5.

Таблица 50.5. Члены интерфейса IContract

Член	Описание
<code>QueryContract()</code>	С помощью <code>QueryContract()</code> можно запросить контракт проверить, реализован ли также другой контракт. Дополнение может поддерживать несколько контрактов.
<code>RemoteToString()</code>	Параметр <code>QueryContract()</code> требует строкового представления контракта. <code>RemoteToString()</code> возвращает строковое представление текущего контракта.
<code>AcquireLifetimeToken()</code> <code>RevokeLifetimeToken()</code>	Клиент вызывает <code>AcquireLifetimeToken()</code> для сохранения ссылки на контракт. <code>AcquireLifetimeToken()</code> увеличивает счетчик ссылок. <code>RevokeLifetimeToken()</code> уменьшает счетчик ссылок.
<code>RemoteEquals()</code>	<code>RemoteEquals()</code> может использоваться для сравнения двух ссылок на контракты.

Интерфейсы контрактов определены в пространствах имен `System.AddIn.Contract`, `System.AddIn.Contract.Collections` и `System.AddIn.Contract.Automation`.

В табл. 50.6 перечислены интерфейсы контрактов, которые можно использовать с контрактом.

Таблица 50.6. Интерфейсы контрактов

Интерфейс контракта	Описание
<code>IListContract<T></code>	<code>IListContract<T></code> может использоваться для возврата списка контрактов.
<code>IEnumeratorContract<T></code>	<code>IEnumeratorContract<T></code> применяется для перечисления элементов <code>IListContract<T></code> .
<code>IServiceProviderContract</code>	Дополнение может предоставлять службы другим дополнениям. Дополнение, предоставляющее службу, называется поставщиком службы и реализует интерфейс <code>IServiceProviderContract</code> . С помощью метода <code>QueryService()</code> дополнение, реализующее этот интерфейс, может быть опрошено на предмет предоставляемых служб.
<code>IProfferServiceContract</code>	<code>IProfferServiceContract</code> — интерфейс, предоставляемый поставщиком службы в сочетании с <code>IServiceProviderContract</code> . Интерфейс <code>IProfferServiceContract</code> определяет методы <code>ProfferService()</code> и <code>RevokeService()</code> . Метод <code>ProfferService()</code> добавляет <code>IServiceProviderContract</code> к предоставляемым службам, а <code>RevokeService()</code> удаляет его.
<code>INativeHandleContract</code>	Этот интерфейс предоставляет доступ к внутренним дескрипторам Windows через метод <code>GetHandle()</code> . Этот контракт применяется с хостами WPF для использования дополнений WPF.

Жизненный цикл

Насколько долго дополнение должно пребывать в загруженном состоянии? Сколько времени оно используется? Когда можно выгрузить домен приложения? Есть несколько вариантов решения этих вопросов. Один вариант заключается в применении счетчика ссылок. Каждое использование дополнения увеличивает значение счетчика. Когда значение счетчика уменьшается до нуля, дополнение может быть выгружено. Другой вариант предусматривает применение сборщика мусора. Если запускается сборщик мусора и на объект не остается никаких ссылок, то этот объект подлежит уничтожению сборщиком. Для удержания объектов в активном состоянии в .NET Remoting используется механизм аренды и спонсорства. Как только истекает время аренды, спонсоры запрашивают, должен ли объект оставаться активным.



Домены приложений описаны в главе 18.

Что касается дополнений, то здесь возникает определенная сложность с выгрузкой, потому что они могут запускаться в разных доменах приложений и разных процессах. Сборщик мусора не может работать через границы процессов. Для управления жизненным циклом в MAF используется смешанная модель. В пределах одного домена приложения применяется сборщик мусора. В пределах конвейера используется неявное спонсорство, но счетчик ссылок доступен извне для контроля спонсора.

Рассмотрим сценарий, когда дополнение загружается в другой домен приложения. Внутри размещаемого приложения (хоста) сборщик мусора очищает представление хоста и адаптер стороны хоста, когда надобность в ссылке отпадает. Для стороны дополнения контракт определяет методы `AcquireLifetimeToken()` и `RevokeLifetimeToken()` для увеличения и уменьшения значения, которое может привести к слишком раннему освобождению объекта, если одна часть станет вызывать метод `RevokeLifetimeToken()` слишком часто. Вместо этого `AcquireLifetimeToken()` возвращает идентификатор маркера жизненного цикла, и этот идентификатор должен применяться для вызова метода `RevokeLifetimeToken()`. Таким образом, эти методы всегда вызываются парами.

Обычно вам не придется иметь дело с методами `AcquireLifetimeToken()` и `RevokeLifetimeToken()`. Вместо этого можно использовать класс `ContractHandle`, вызывающий `AcquireLifetimeToken()` в конструкторе и `RevokeLifetimeToken()` — в финализаторе.



Финализаторы рассматриваются в главе 13.

В сценариях с загрузкой дополнения в новый домен приложения можно избавиться от загруженного кода, когда дополнение уже не нужно. В MAF используется простая модель для определения одного дополнения на домен приложения, чтобы выгружать его, когда необходимость в дополнении отпадает. Дополнение выступает владельцем домена приложения, если домен приложения создается при активизации дополнения. Домен приложения не выгружается автоматически, если он был создан ранее.

Класс `ContractHandle` используется в адаптере стороны хоста для добавления счетчика ссылки на дополнение. Члены этого класса описаны в табл. 50.7.

Таблица 50.7. Члены `ContractHandle`

Член	Описание
<code>Contract</code>	При конструировании класса <code>ContractHandle</code> объект, реализующий <code>ISContract</code> , может быть присвоен для сохранения ссылки на него. Свойство <code>Contract</code> возвращает этот объект.
<code>Dispose()</code>	Метод <code>Dispose()</code> может быть вызван вместо ожидания, когда сборщик мусора выполнит финализацию для аннулирования маркера жизненного цикла.
<code>AppDomainOwner()</code>	<code>AppDomainOwner()</code> — статический метод класса <code>ContractHandle</code> , возвращающий адаптер дополнения, если он владеет доменом приложения, переданным в этот метод.
<code>ContractOwnsAppDomain()</code>	С помощью статического метода <code>ContractOwnsAppDomain()</code> можно проверить, является ли специфицированный контракт владельцем домена приложения. Таким образом, домен приложения выгружается при освобождении контракта.

Поддержка версий

Поддержка версий является очень большой проблемой для дополнений. Размещаемое приложение разрабатывается заранее с учетом будущих дополнений. К дополнению предъявляется требование, чтобы была возможность загрузки старых версий дополнения новыми версиями приложения. Это также должно работать и в противоположном направлении: старые приложения должны запускать новые версии дополнений. Но что, если изменится контракт?

Сборка `System.AddIn` полностью независима от реализаций размещаемого приложения и дополнений. Это обеспечивается концепцией конвейера, состоящего из семи частей.

Пример дополнения

Начнем с простого примера размещаемого приложения, которое может загружать дополнения калькулятора. Дополнения могут поддерживать разные вычислительные операции, предоставленные дополнениями.

Необходимо создать решение с шестью проектами библиотек и одним консольным приложением. Проекты примера приложения представлены в табл. 50.8. В таблице перечислены сборки, ссылки на которые понадобятся. Из-за ссылок на другие проекты внутри решения понадобится установить свойство `Copy Local` (Копировать локально) в `False`, чтобы сборки не копировались. Исключением является консольный проект `HostApp`, которому нужна ссылка на проект `HostView`. Эта сборка должна быть скопирована, чтобы ее можно было найти из размещаемого приложения. Также нужно будет изменить выходной путь сгенерированных сборок, чтобы сборки копировались в правильные каталоги конвейера.

Контракт дополнения

Начнем с реализации сборки контракта. Сборки контракта содержат интерфейс контракта, определяющий протокол для коммуникаций между хостом и дополнением.

В приведенном ниже коде показан контракт, определенный для примера приложения — калькулятора. Приложение определяет контракт с методами `GetOperations()` и `Operate()`. Метод `GetOperations()` возвращает список математических операций, поддерживаемых дополнением калькулятора.

Таблица 50.8. Описание решения

Проект	Ссылки	Выходной путь	Описание
CalcContract	System.AddIn.Contract	..\Pipeline\Contracts\	Эта сборка содержит контракт для взаимодействия с дополнением. Контракт оп-ределен интерфейсом.
CalcView	System.AddIn	..\Pipeline\AddInViews\	Сборка CalcView содержит абстрактный класс, на который ссылается дополнение. Это сторона контракта, относящаяся к дополнению.
CalcAddIn	System.AddIn CalcView	..\Pipeline\AddIns\CalcAddIn\	CalcAddIn — проект дополнения, ссы-лающийся на сборку представления до-полнения. Эта сборка содержит реализа-цию дополнения.
CalcAddInAdapter	System.AddIn System.AddIn.Contract CalcView CalcContract	..\Pipeline\AddInSideAdapters\	CalcAddInAdapter соединяет пред-ставление дополнения и сборку контракта и отображает контракт на представление дополнения.
HostView			Сборка, содержащая абстрактный класс представления хоста, не нуждается в ссылке на любую сборку дополнения, а также не имеет ссылок на другой проект в решении.
HostAdapter	System.AddIn System.AddIn.Contract HostView CalcContract	..\Pipeline\HostSideAdapters\	Адаптер хоста отображает представление хоста на контракт. Таким образом, он нуж-дается в ссылках на оба эти проекта.
HostApp	System.AddIn HostView		Размещаемое приложение, активизирую-щее дополнение.

Операция определена интерфейсом `IOperationContract`, представляющим собой контракт. `IOperationContract` определяет доступные только для чтения свойства `Name` и `NumberOperands`.

Метод `Operate()` вызывает операцию из дополнения и требует операции, определенной интерфейсом `IOperation`, и операндов в массиве `double`.

С таким контрактом возможна поддержка дополнением любых операций, принимающих любое количество операндов `double` и возвращающих значение `double`.

Атрибут `AddInContract` используется `AddInStore` для построения кэша. Атрибут `AddInContract` помечает класс как интерфейс контракта дополнения.



Используемый здесь контракт функционирует подобно контракту дополнения из главы 28. Однако к контрактам MEF не предъявляется таких требований, как к контрактам MAF. В этом отношении контракты MAF более ограничены из-за границ домена приложений и процесса между хостом и дополнением. Однако в архитектуре MEF не используются разные домены приложений.



```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    [AddInContract]
    public interface ICalculatorContract: IContract
    {
        IListContract<IOperationContract> GetOperations();
        double Operate(IOperationContract operation, double[] operands);
    }
    public interface IOperationContract: IContract
    {
        string Name { get; }
        int NumberOperands { get; }
    }
}
```

Фрагмент кода `CalcContract\ICalculatorContract.cs`

Представление дополнения калькулятора

Представление дополнения переопределяет контракт так, как он выглядит для дополнения. Этот контракт определен интерфейсами `ICalculatorContract` и `IOperationContract`. Для этого представление дополнения определяет абстрактный класс `Calculator` и конкретный класс `Operation`.

Для `Operation` не существует определенной реализации, которая требовалась бы каждому дополнению. Вместо этого класс уже реализован в сборке представления дополнения. Этот класс описывает операцию для математических вычислений с помощью свойств `Name` и `NumberOperands`.

Абстрактный класс `Calculator` определяет методы, которые должны быть реализованы дополнениями. В то время как контракт определяет типы параметров и возврата, которые должны передаваться через границы доменов приложений и процессов, это не касается представления дополнения. Здесь можно использовать типы, которые облегчают написание дополнений разработчику. Метод `GetOperations()` возвращает `IList<Operation>` вместо `IListOperation<IOperationContract>`, как вы уже видели в сборке контракта. Атрибут `AddInBase` идентифицирует класс как представление дополнения для хранения.

```

using System.AddIn.Pipeline;
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    [AddInBase]
    public abstract class Calculator
    {
        public abstract IList<Operation> GetOperations();
        public abstract double Operate(Operation operation, double[] operand);
    }
    public class Operation
    {
        public string Name { get; set; }
        public int NumberOperands { get; set; }
    }
}

```

Фрагмент кода *CalcView\CalculatorView.cs*

Адаптер дополнения калькулятора

Адаптер дополнения отображает контракт на представление дополнения. Эта сборка имеет ссылки как на сборку контракта, так и на сборку представления дополнения. Реализация адаптера требует отображения метода `IListContract<IOperationContract> GetOperations()` на метод представления `IList<Operation> GetOperations()`.

Сборка включает классы `OperationViewToContractAddInAdapter` и `CalculatorViewToContractAddInAdapter`. Эти классы реализуют интерфейсы `IOperationContract` и `ICalculatorContract`. Методы базового интерфейса `IContract` могут быть реализованы наследованием от базового класса `ContractBase`. Этот класс предоставляет реализацию по умолчанию. `OperationViewToContractAddInAdapter` реализует другие члены интерфейса `IOperationContract` и просто переадресует вызовы к представлению `Operation`, присвоенному в конструкторе.

Класс `OperationViewToContractAddInAdapter` также содержит статические вспомогательные методы `ViewToContractAdapter()` и `ContractToViewAdapter()`, отображающие `Operation` на `IOperationContract` и наоборот.

```

using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationViewToContractAddInAdapter: ContractBase,
        IOperationContract
    {
        private Operation view;
        public OperationViewToContractAddInAdapter(Operation view)
        {
            this.view = view;
        }
        public string Name
        {
            get { return view.Name; }
        }
        public int NumberOperands
        {
            get { return view.NumberOperands; }
        }
        public static IOperationContract ViewToContractAdapter(Operation view)
        {
            return new OperationViewToContractAddInAdapter(view);
        }
    }
}

```

```

public static Operation ContractToViewAdapter(IOperationContract contract)
{
    return (contract as OperationViewToContractAddInAdapter).view;
}
}

```


Фрагмент кода `CalcAddInAdapter\OperationViewToContractAddInAdapter.cs`

Класс `CalculatorViewToContractAddInAdapter` очень похож на `OperationViewToContractAddInAdapter`: он происходит от `ContractBase`, наследуя реализацию интерфейса `IContract` по умолчанию, и реализует интерфейс контракта. На этот раз интерфейс `ICalculatorContract` реализован методами `GetOperations()` и `Operate()`.

Метод `Operate()` адаптера вызывает метод `Operate()` класса представления `Calculator`, где `IOperationContract` должен быть преобразован в `Operation`. Это делается вспомогательным статическим методом `ContractToViewAdapter()`, определенным в классе `OperationViewToContractAddInAdapter`.

Реализация метода `GetOperations` нуждается в преобразовании коллекции `IListContract<IOperationContract>` в `IList<Operation>`. Для таких преобразований коллекции класс `CollectionAdapters` определяет методы преобразования `ToIList()` и `ToIListContract()`. Здесь для преобразования используется метод `ToIListContract()`.

Атрибут `AddInAdapter` идентифицирует класс как адаптер стороны дополнения.

 `using System.AddIn.Contract;`
`using System.AddIn.Pipeline;`
`namespace Wrox.ProCSharp.MAF`
`{`
`[AddInAdapter]`
`internal class CalculatorViewToContractAddInAdapter: ContractBase,`
`ICalculatorContract`
`{`
`private Calculator view;`
`public CalculatorViewToContractAddInAdapter(Calculator view)`
`{`
`this.view = view;`
`}`
`public IListContract<IOperationContract> GetOperations()`
`{`
`return CollectionAdapters.ToIListContract<Operation,`
`IOperationContract>(view.GetOperations(),`
`OperationViewToContractAddInAdapter.ViewToContractAdapter,`
`OperationViewToContractAddInAdapter.ContractToViewAdapter);`
`}`
`public double Operate(IOperationContract operation, double[] operands)`
`{`
`return view.Operate(`
`OperationViewToContractAddInAdapter.ContractToViewAdapter(`
`operation), operands);`
`}`
`}`
`}`

Фрагмент кода `CalcAddInAdapter\CalculatorViewToContractAddInAdapter.cs`



Поскольку классы адаптеров вызываются системой рефлексии .NET, с этими классами можно использовать модификатор доступа `internal`. Поскольку эти классы относятся к деталям реализации, применять его будет хорошей идеей.

Дополнение калькулятора

Теперь дополнение содержит реализацию некоторой функциональности. Дополнение реализовано классом `CalculatorV1`. Сборка дополнения имеет зависимость от сборки представления дополнения, поскольку это необходимо для реализации абстрактного класса `Calculator`.

Атрибут `AddIn` помечает класс как дополнение для хранилища дополнений и добавляет информацию об издателе, версии, а также описание. На стороне хоста эта информация доступна из `AddInToken`.

`CalculatorV1` возвращает список поддерживаемых операций в методе `GetOperations()`. Метод `Operate()` вычисляет операнды в зависимости от операции.

```
using System;
using System.AddIn;
using System.Collections.Generic;

namespace Wrox.ProCSharp.MAF
{
    [AddIn("CalculatorAddIn", Publisher="Wrox Press", Version="1.0.0.0",
        Description="Sample AddIn")]
    public class CalculatorV1: Calculator
    {
        private List<Operation> operations;
        public CalculatorV1()
        {
            operations = new List<Operation>();
            operations.Add(new Operation() { Name = "+", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "-", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "/", NumberOperands = 2 });
            operations.Add(new Operation() { Name = "*", NumberOperands = 2 });
        }

        public override IList<Operation> GetOperations()
        {
            return operations;
        }

        public override double Operate(Operation operation, double[] operand)
        {
            switch (operation.Name)
            {
                case "+":
                    return operand[0] + operand[1];
                case "-":
                    return operand[0] - operand[1];
                case "/":
                    return operand[0] / operand[1];
                case "*":
                    return operand[0] * operand[1];
                default:
                    throw new InvalidOperationException(
                        String.Format("invalid operation {0}", operation.Name));
            }
        }
    }
}
```

Фрагмент кода *CalcAddIn\Calculator.cs*

Представление хоста калькулятора

Продолжим рассмотрение и обратимся к представлению хоста на стороне хоста. Подобно представлению дополнения, представление хоста определяет абстрактный класс с методами, подобными контракту. Однако определенные здесь методы вызываются размещаемым приложением.

Оба класса — `Calculator` и `Operation` — являются абстрактными, а их члены реализованы адаптером хоста. Классы здесь должны реализовать интерфейс, чтобы их могло использовать размещаемое приложение.


```
using System.Collections.Generic;
namespace Wrox.ProCSharp.MAF
{
    public abstract class Calculator
    {
        public abstract IList<Operation> GetOperations();
        public abstract double Operate(Operation operation,
            params double[] operand);
    }
    public abstract class Operation
    {
        public abstract string Name { get; }
        public abstract int NumberOperands { get; }
    }
}
```

Фрагмент кода *HostView\CalculatorHostView.cs*

Адаптер хоста калькулятора

Сборка адаптера хоста ссылается на представление хоста и контракт для отображения представления контракта. Класс `OperationContractToViewHostAdapter` реализует члены абстрактного класса `Operation`. Класс `CalculatorContractToViewHostAdapter` реализует члены абстрактного класса `Calculator`.

В `OperationContractToViewHostAdapter` ссылка на контракт присваивается в конструкторе. Класс адаптера также содержит экземпляр `ContractHandle`, который добавляет ссылку времени жизни на `contract`, так что дополнение остается загруженным до тех пор, пока в нем нуждается размещаемое приложение.

```
 using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
    internal class OperationContractToViewHostAdapter: Operation
    {
        private ContractHandle handle;
        public IOperationContract Contract { get; private set; }
        public OperationContractToViewHostAdapter(IOperationContract contract)
        {
            this.Contract = contract;
            handle = new ContractHandle(contract);
        }
        public override string Name
        {
            get
            {
                return Contract.Name;
            }
        }
    }
}
```



```


public override int NumberOperands
{
    get
    {
        return Contract.NumberOperands;
    }
}
}
internal static class OperationHostAdapters
{
    internal static IOperationContract ViewToContractAdapter(Operation view)
    {
        return ((OperationContractToViewHostAdapter)view).Contract;
    }
    internal static Operation ContractToViewAdapter(
        IOperationContract contract)
    {
        return new OperationContractToViewHostAdapter(contract);
    }
}
}

```

Фрагмент кода *HostAdapter\OperationContractToViewHostAdapter.cs*

Класс `CalculatorContractToViewHostAdapter` реализует методы абстрактного класса представления хоста `Calculator` и переадресует вызов контракту. Опять-таки, здесь присутствует `ContractHandle`, который хранит ссылку на контракт, что подобно преобразованию типа адаптера стороны дополнения. На этот раз преобразование типа осуществляется в другом направлении, чем в случае адаптера дополнения.

Атрибут `HostAdapter` помечает класс как адаптер, который должен быть установлен в каталоге `HostSideAdapters`.

 using System.Collections.Generic;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.MAF
{
 [HostAdapter]
 internal class CalculatorContractToViewHostAdapter: Calculator
 {
 private ICalculatorContract contract;
 private ContractHandle handle;
 public CalculatorContractToViewHostAdapter(ICalculatorContract contract)
 {
 this.contract = contract;
 handle = new ContractHandle(contract);
 }
 public override IList<Operation> GetOperations()
 {
 return CollectionAdapters.ToIList<IOperationContract, Operation>(
 contract.GetOperations(),
 OperationHostAdapters.ContractToViewAdapter,
 OperationHostAdapters.ViewToContractAdapter);
 }
 public override double Operate(Operation operation, double[] operands)
 {
 return contract.Operate(OperationHostAdapters.ViewToContractAdapter(
 operation), operands);
 }
 }
}

Фрагмент кода *HostAdapter\CalculatorContractToViewHostAdapter.cs*

Хост калькулятора

Простое размещаемое приложение строится с использованием технологии WPF. Пользовательский интерфейс этого приложения показан на рис. 50.3. Вверху находится список доступных дополнений, а слева — операции активного дополнения. При выборе операции, которая должна быть вызвана, отображаются операнды. После ввода значений операндов может быть вызвана операция дополнения.

Кнопки в нижнем ряду служат для перестройки и обновления хранилища дополнений и для выхода из приложения.

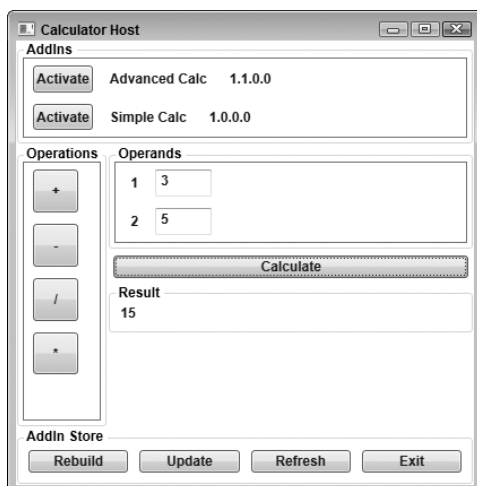


Рис. 50.3. Пользовательский интерфейс хоста калькулятора

Приведенный ниже код XAML демонстрирует дерево объектов пользовательского интерфейса. С элементами `ListBox` используются разные стили шаблонов, чтобы дать специфическое представление списка дополнений, списка операций и списка операндов.



Сведения о шаблонах элементов можно найти в главе 35.



```
<DockPanel>
  <GroupBox Header="AddIn Store" DockPanel.Dock="Bottom">
    <UniformGrid Columns="4">
      <Button x:Name="rebuildStore" Click="RebuildStore"
        Margin="5"> Rebuild </Button>
      <Button x:Name="updateStore" Click="UpdateStore"
        Margin="5"> Update </Button>
      <Button x:Name="refresh" Click="RefreshAddIns"
        Margin="5"> Refresh </Button>
      <Button x:Name="exit" Click="App_Exit" Margin="5"> Exit </Button>
    </UniformGrid>
  </GroupBox>
  <GroupBox Header="AddIns" DockPanel.Dock="Top">
    <ListBox x:Name="listAddIns" ItemsSource="{Binding}"
      Style="{StaticResource listAddInsStyle}" />
  </GroupBox>
  <GroupBox DockPanel.Dock="Left" Header="Operations">
```

```


<ListBox x:Name="listOperations" ItemsSource="{Binding}"
        Style="{StaticResource listOperationsStyle}" />
</GroupBox>
<StackPanel DockPanel.Dock="Right" Orientation="Vertical">
    <GroupBox Header="Operands">
        <ListBox x:Name="listOperands" ItemsSource="{Binding}"
                Style="{StaticResource listOperandsStyle}">
        </ListBox>
    </GroupBox>
    <Button x:Name="buttonCalculate" Click="Calculate" IsEnabled="False"
            Margin="5"> Calculate </Button>
    <GroupBox DockPanel.Dock="Bottom" Header="Result">
        <Label x:Name="labelResult" />
    </GroupBox>
</StackPanel>
</DockPanel>

```

Фрагмент кода *HostAppWPF\CalculatorHostWindow.xaml*

В приведенном ниже коде метод `FindAddIns()` вызывается в конструкторе `Window`. Метод `FindAddIns()` использует класс `AddInStore` для получения коллекции объектов `AddInToken` и передачи их свойству `DataContext` элемента `ListBox` по имени `listAddIns` для отображения. В первом параметре метода `AddInStore.FindAddIns()` передается абстрактный класс `Calculator`, определенный представлением хоста, чтобы найти все дополнения из хранилища, применимые к контракту. Во втором параметре передается каталог конвейера, прочитанный из конфигурационного файла приложения. В примере приложения, доступном на прилагаемом компакт-диске, вы должны изменить каталог в конфигурационном файле приложения, чтобы он соответствовал существующей структуре каталогов.

```

 using System;
using System.AddIn.Hosting;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using Wrox.ProCSharp.MAF.Properties;

namespace Wrox.ProCSharp.MAF
{
    public partial class CalculatorHostWindow: Window
    {
        private Calculator activeAddIn = null;
        private Operation currentOperation = null;

        public CalculatorHostWindow()
        {
            InitializeComponent();
            FindAddIns();
        }

        void FindAddIns()
        {
            try
            {
                this.listAddIns.DataContext =
                    AddInStore.FindAddIns(typeof(Calculator),
                    Settings.Default.PipelinePath);
            }
        }
    }
}

```

```

        catch (DirectoryNotFoundException ex)
        {
            MessageBox.Show("Verify the pipeline directory in the " +
                "config file");
            Application.Current.Shutdown();
        }
    }
    //...

```

Фрагмент кода HostAppWPF\CalculatorHostWindow.xaml.cs

Чтобы обновить кэш хранилища дополнений, методы `UpdateStore()` и `RebuildStore()` отображаются на события щелчка на кнопках **Update** (Обновить) и **Rebuild** (Перестроить). Внутри реализации этих методов используются методы `Rebuild()` и `Update()` класса `AddInStore`. Эти методы возвращают строковый массив предупреждений, если сборки оказываются в неправильных каталогах. Из-за сложности структуры конвейера высока вероятность, что у вас не сразу получится правильная конфигурация проекта для копирования сборок в правильные каталоги. Чтение информации, возвращенной этими методами, даст вам ясное объяснение того, что пошло не так. Например, сообщение “No usable AddInAdapter parts could be found in assembly Pipeline\AddInSideAdapters\CalcView.dll” ясно указывает на неправильное расположение сборки `CalcView`.

```

private void UpdateStore(object sender, RoutedEventArgs e)
{
    string[] messages = AddInStore.Update(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}
private void RebuildStore(object sender, RoutedEventArgs e)
{
    string[] messages =
        AddInStore.Rebuild(Settings.Default.PipelinePath);
    if (messages.Length != 0)
    {
        MessageBox.Show(string.Join("\n", messages),
            "AddInStore Warnings", MessageBoxButton.OK,
            MessageBoxImage.Warning);
    }
}

```

На рис. 50.3 рядом со списком доступных дополнений находится кнопка **Activate** (Активизировать). Щелчок на этой кнопке вызывает метод-обработчик `ActivateAddIn()`. При такой реализации дополнение активизируется вызовом метода `Activate()` класса `AddInToken`. Здесь дополнение загружается в новый процесс, созданный классом `AddInProcess`. Этот класс запускает процесс `AddInProcess32.exe`. Установка свойства `KeepAlive` процесса в `false` останавливает процесс, как только сборщик мусора уберет последнюю ссылку на дополнение. Параметр `AddInSecurityLevel.Internet` устанавливает ограниченные права запускаемому дополнению. Последний оператор `ActivateAddIn()` вызывает метод `LastOperations()`, который, в свою очередь, вызывает метод `GetOperations()` дополнения. `GetOperations()` присваивает возвращенный список контексту данных `ListBox listOperations` для отображения всех операций.

```

private void ActivateAddIn(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;

```

```

Trace.Assert(el != null, "ActivateAddIn invoked from the wrong " +
    "control type");

AddInToken addIn = el.Tag as AddInToken;
Trace.Assert(el.Tag != null, String.Format(
    "An AddInToken must be assigned to the Tag property " +
    "of the control {0}", el.Name);
AddInProcess process = new AddInProcess();
process.KeepAlive = false;
activeAddIn = addIn.Activate<Calculator>(process,
    AddInSecurityLevel.Internet);
ListOperations();
}
void ListOperations()
{
    this.listOperations.DataContext = activeAddIn.GetOperations();
}

```

После активизации дополнения и отображения списка операций в пользовательском интерфейсе пользователь может выбирать операцию. Событию Click элемента Button, показанного в категории Operations, назначается метод-обработчик OperationSelected(). В его реализации объект Operation, присвоенный свойству Tag элемента Button, извлекается для получения количества операндов, необходимых для выбранной операции. Чтобы позволить пользователю добавлять значения операндам, массив объектов OperandUI привязывается к ListBox listOperations.

```

private void OperationSelected(object sender, RoutedEventArgs e)
{
    FrameworkElement el = sender as FrameworkElement;
    Trace.Assert(el != null, "OperationSelected invoked from " +
        "the wrong control type");
    Operation op = el.Tag as Operation;
    Trace.Assert(el.Tag != null, String.Format(
        "An AddInToken must be assigned to the Tag property " +
        "of the control {0}", el.Name);
    currentOperation = op;
    ListOperands(new double[op.NumberOperands]);
}
private class OperandUI
{
    public int Index { get; set; }
    public double Value { get; set; }
}
void ListOperands(double[] operands)
{
    this.listOperands.DataContext =
        operands.Select((operand, index) =>
            new OperandUI()
            { Index = index + 1, Value = operand }).ToArray();
}

```

Метод Calculate() вызывается событием Click кнопки Calculate (Вычислить). Здесь операнды извлекаются из пользовательского интерфейса, операция и операнды передаются методу Operate() дополнения, а результат отображается в содержимом метки.


```

private void Calculate(object sender, RoutedEventArgs e)
{
    OperandUI[] operandsUI = (OperandUI[])this.listOperands.DataContext;
    double[] operands = operandsUI.Select(opui => opui.Value).ToArray();
    labelResult.Content = activeAddIn.Operate(currentOperation, operands);
}

```

Дополнительные дополнения

На этом вся трудная работа закончена. Компоненты конвейера и размещаемое приложение созданы. Конвейер работает, и добавление к размещаемому приложению новых дополнений, таких как Advanced Calculator, показанный в следующем фрагменте кода, становится простой задачей.

 [AddIn("Advanced Calc", Publisher = "Wrox Press", Version = "1.1.0.0",
Description = "Another AddIn Sample")]
public class AdvancedCalculatorV1: Calculator

Фрагмент кода *AdvancedCalcAddIn\AdvancedCalculator.cs*

Резюме

В этой главе были представлены концепции технологии, появившейся в .NET 3.5 — платформы управляемых дополнений (Managed Add-In Framework — MAF).

Для обеспечения независимости между сборками размещаемого приложения и дополнений в MAF используется концепция конвейера. Четко определенный контракт отделяет представление хоста от представления дополнения. Адаптеры обеспечивают возможность обеим сторонам изменяться независимо друг от друга.

В следующей главе рассматривается технология .NET Enterprise Services.