36

Дополнения

Дополнения (add-ins) позволяют добавлять функциональность к приложению спустя некоторое время после его сборки. Вы можете создать некоторый "скелет" приложения, который будет со временем получать все больше и больше функциональности, написанной как вашей командой разработчиков, так и независимыми поставщиками, которые пожелают расширить функциональность вашего приложения за счет дополнений.

Сегодня дополнения используются в разных приложениях, таких как Internet Explorer и Visual Studio. Internet Explorer — это хост-приложение, которое представляет каркас для дополнений, используемый многими компаниями, поставляющими расширения для просмотра Web-страниц. Shockware Flash Object позволяет просматривать Web-страницы с Flash-содержимым. Панель инструментов Google предоставляет специфические средства Google, которые могут быть быстро доступны из Internet Explorer. Visual Studio также обладает моделью дополнений, позволяющей расширять Visual Studio на разных уровнях.

Ваши собственные приложения также могут реализовать модель дополнений, динамически загружая и используя функциональность из внешних сборок. При использовании такой модели следует заранее подумать о многих проблемах. Как обнаруживать новые сборки? Как разрешать проблемы, связанные с конфликтами версий? Может ли дополнение нарушить стабильность принимающего приложения?

.NET Framework 3.5 предоставляет каркас для хостинга и создания дополнений в сборке `System.Addin`. Этот каркас также известен под именем MAF (Managed AddIn Framework — каркас управляемых сборок).

Дополнения также известны под разными терминами — "add-on" или "plug-in".

Темы, рассматриваемые в настоящей главе:

- архитектура `System.Addin`;
- созданием простого дополнения.

Архитектура `System.Addin`

Когда вы создаете приложение, позволяющее добавлять дополнения во время выполнения, вам приходится сталкиваться с определенными сложностями, например, как находить дополнения и как решать проблемы совместимости версий, когда принимающее приложение и дополнение развиваются независимо друг от друга. Есть несколько путей преодоления этих сложностей. В этом разделе вы узнаете кое-что о них и способах их преодоления, предусмотренных в архитектуре MAF:

- сложности, связанные с дополнениями;
- канальная архитектура;
- исследование;
- активизация;
- изоляция;
- жизненный цикл;
- поддержка версий.

Сложности, связанные с дополнениями

Создание принимающего (хост) приложения, которое динамически загружает сборки, добавляемые позднее, представляет ряд сложностей, которые приходится преодолевать (табл. 36.1).

Таблица 36.1. Сложности дополнений

Сложность дополнения	Описание
Исследование	Как принимающее приложение может находить дополнения? Есть несколько разных вариантов. Один из них — добавление информации о дополнении в конфигурационный файл приложения. Это имеет тот недостаток, что инсталляция 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 13.
Активизация	С динамическими загружаемыми сборками невозможно просто использовать операцию `new` для создания экземпляра. Вы можете создать такие сборки с помощью класса `Activator`. Также различные опции активизации могут применяться, если дополнение загружено внутри другого домена приложений или нового процесса. Сборки и домены приложений описаны в главе 17.
Изоляция	Дополнение может нарушить работу принимающего приложения, как вы, вероятно уже видели на примере Internet Explorer, терпящего крах из-за различных дополнений. В зависимости от типа принимающего приложения и способа интеграции дополнения, оно может быть загружено внутри другого домена приложения или также внутри другого процесса.
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 принимающего приложения может загружать старые дополнения, а старое приложение может иметь опцию для загрузки новых дополнений.

Теперь давайте рассмотрим архитектуру MAF, и то, как каркас решает перечисленные проблемы. Дизайн MAF определен следующими целями:

- дополнения должны легко разрабатываться;
- нахождение дополнений во время исполнения должно быть производительным;
- разработка принимающих приложений должна быть также простой, но не столь простой, как разработка дополнений;
- дополнение и принимающее приложение должны развиваться независимо.

Канальная архитектура

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

На рис. 36.1 показан канал, 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 вы можете 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 реализует интерфейса, определенного контрактом. Вместо этого они просто имеют ссылку на сборку-представление (view). Хост-приложение ссылается на представление хоста; дополнение ссылается на представления дополнения. Сами представления содержат абстрактные классы представлений, которые определяют методы и свойства, как они определены контрактом.

Рис. 36.1. Канальная архитектура MAF

На рис. 36.2 представлены отношения между классами канала. Класс хоста имеет ассоциацию с абстрактным представлением хоста и вызывает его методы. Абстрактный класс представления реализован адаптером хоста. Адаптеры выполняют 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, наследуемый от абстрактного базового класса представления хоста, реализуя его методы и свойства. Этот адаптер включает ссылку на контракт для переадресации вызовов от представления к контракту.

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.

Исследование

Каким образом принимающее хост-приложение может находить новые дополнения? Архитектура MAF использует предопределенную структуру каталогов для нахож-

дения дополнений и других сборок канала. Компоненты канала должны располагаться в следующих каталогах:

- HostSideAdapters
- Contracts
- AddInSideAdapters
- AddInViews
- AddIns

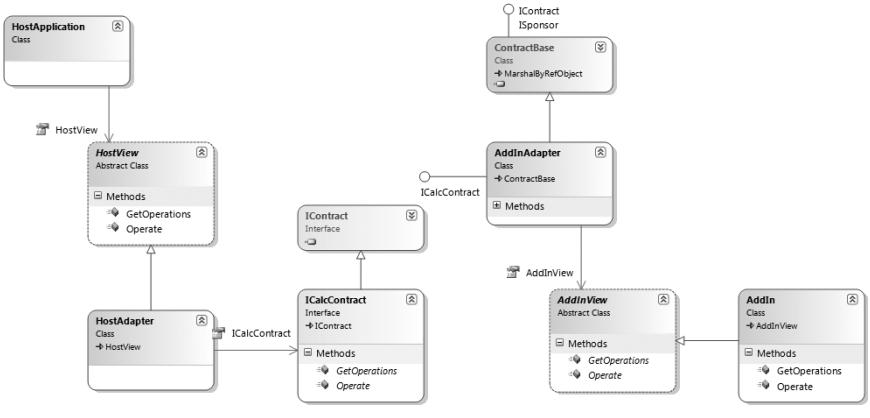

Рис. 36.2. Отношения между классами канала

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

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

Кэшированная информация о компонентах канала 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. 36.2 перечислены методы класса `AddInStore`.

Активизация и изоляция

Метод `FindAddIns()` класса `AddInStore` возвращает коллекцию объектов `AddInToken`, представляющую дополнение. Через класс `AddInToken` вы можете получить доступ к информации о сборке — такой как имя, описание, издатель и версия. Вы можете активизировать дополнения методом `Activate()`. В табл. 36.3 перечислены свойства и методы класса `AddInToken`.

Таблица 36.2. Методы класса `AddInStore`

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

Таблица 36.3. Свойства и методы класса `AddInToken`

Член `AddInToken`	Описание

`Name`
`Publisher`
`Version`
`Description`
	Свойства `Name`, `Publisher`, `Version` и `Description` класса `AddInToken` возвращают информацию о дополнении, которая присваивается дополнению атрибутом `AddInAttribute`.

`AssemblyName`	`AssemblyName` возвращает имя сборки, содержащей дополнение.
`EnableDirectConnect`	В свойстве `EnableDirectConnect` вы можете установить значение, указывающее на то, что хост должен непосредственно подключаться к дополнению вместо использования компонентов канала. 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, чтобы все компоненты канала присутствовали.
`QualificationData`	Дополнение может пометить домен приложения и требования безопасности атрибутом `QualificationDataAttribute`. Дополнение может перечислять требования к безопасности и изоляции. Например, `[QualificationData("Isolation", "NewAppDomain")]` означает, что дополнение должно быть размещено в новом процессе. Вы можете прочесть эту информацию из `AddInToken` для активизации дополнения со специфицированными требованиями. В дополнение к требованиям домена приложений и безопасности, вы можете использовать этот атрибут для передачи специальной информации по каналу.
`Activate()`	Дополнение активизируется методом `Activate()`. С параметрами этого метода вы можете определить, должно ли дополнение загружаться внутри нового домена приложения или нового процесса. Вы можете также определить привилегии, которые получит дополнение.

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

Метод `Activate()` класса `AddInToken` имеет несколько перегрузок, которым вы можете передать среду, куда желаете загрузить дополнение. Допустимые варианты перечислены в табл. 36.4.

Таблица 36.4. Параметры метода `AddInToken.Activate()`

Параметр `AddInToken.Activate()`	Описание
`AppDomain`	Вы можете передать новый домен приложения, в который должно быть загружено дополнение. Таким образом, вы можете сделать его независимым от хост-приложения и, кроме того, оно может быть затем выгружено этим доменом приложения.
`AddInSecurityLevel`	Если дополнение должно быть запущено с другими уровнями безопасности, вы можете передать значение перечисления `AddInSecurityLevel`. Возможные значения — `Internet`, `Intranet`, `FullTrust` и `Host`.
`PermissionSet`	Если предопределенные уровни безопасности недостаточно специфичны, вы можете также присвоить `PermissionSet` домену приложения дополнения.
`AddInProcess`	Дополнение также может быть запущено внутри другого процесса принимающего хост-приложения. Вы можете передать новый `AddInProcess` методу `Activate()`. Новый процесс может завершиться, когда все дополнения выгружены, или продолжать выполнение. Эта опция, которая может быть установлена свойством `KeepAlive`.
`AddInEnvironment`	Передача объекта `AddInEnvironment` — еще один вариант определения домена приложения, куда должно быть загружено приложение. Конструктору `AddInEnvironment` вы можете передать объект `AppDomain`. Можно также получить существующий `AddInEnvironment` дополнения из свойства `AddInEnvironment` класса `AddInController`.

Домены приложений описаны в главе 17.

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

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

1. Создается домен приложения с указанными привилегиями.
2. Сборка дополнения загружается в новый домен приложения методом `Assembly.LoadFrom()`.
3. Средствами рефлексии вызывается конструктор дополнения по умолчанию. Поскольку дополнение наследуется от базового класса, который определен в представлении дополнения, сборка представления также загружается.
4. Далее конструируется экземпляр адаптера стороны дополнения. Экземпляр дополнения передается конструктору адаптера, так что адаптер может подключить контракт к дополнению. Адаптер дополнения наследуется от базового класса `MarshalByRefObject`, так что он может вызываться через границы доменов приложений.
5. Код активизации возвращает прокси адаптеру дополнения в домен принимающего приложения. Поскольку адаптер дополнения реализует интерфейс контракта, прокси содержит методы и свойства интерфейса контракта.

6. Конструируется экземпляр адаптера стороны хоста в домене принимающего приложения. Прокси адаптера стороны дополнения передается конструктору. Активизация находит тип адаптера стороны хоста из маркера дополнения.

Адаптер стороны хоста возвращается принимающему приложению.

Контракты

Контракты определяют границы между стороной хоста и стороной дополнения. Контракты определяются интерфейсом, который должен наследоваться от базового интерфейса `IContract`. 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 вы можете 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, контрактов и не имеющие реализации.

Члены интерфейса `IContract` описаны в табл. 36.5.

Таблица 36.5. Члены интерфейса `IContract`

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

Интерфейсы контрактов определены в пространстве имен `System.AddIn.Contract`, `System.AddIn.Contract.Collections` и `System.AddIn.Contract.Automation`. В табл. 36.6 перечислены интерфейсы контрактов, которые вы можете использовать с контрактом.

Таблица 36.6. Интерфейсы контрактов

Контракт	Описание
`IListContract<T>`	`IListContract<T>` может использоваться для возврата списка контрактов.
`IEnumeratorContract<T>`	`IEnumeratorContract<T>` применяется для перечисления элементов `IListContract<T>`.
`IServiceProviderContract`	Дополнение может предоставлять службы другим дополнениям. Дополнение, предоставляющее службу, называется поставщиком службы и реализует интерфейс `IServiceProviderContract`. Посредством метода `QueryService()` дополнение, реализующее этот интерфейс, может быть опрошено на предмет предоставляемых служб.
`IProfferServiceContract`	`IProfferServiceContract` — интерфейс, предоставляемый поставщиком службы в сочетании с `IServiceProviderContract`. Интерфейс `IProfferServiceContract` определяет методы `ProfferService()` и `RevokeService()`. Метод `ProfferService()` добавляет `IServiceProviderContract` к предоставляемым службам, а `RevokeService()` удаляет его.
`INativeHandleContract`	Этот интерфейс предоставляет доступ к "родным" дескрипторам Windows через метод `GetHandle()`. 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 разрешения этих вопросов. Один вариант заключается в использовании счетчика ссылок. Каждое использование дополнения увеличивает значение счетчика. Когда значение счетчика уменьшается до нуля, дополнение может быть выгружено. Другой вариант — применять сборщик мусора. Если запускается сборщик мусора, и на объект не остается никаких ссылок, то этот объект подлежит уничтожению сборщиком. .NET Remoting использует механизм аренды и спонсорства для удержания объектов в живом состоянии. Как только истекает время аренды, спонсоры запрашивают, должен ли объект оставаться живым.

Что касается дополнений, 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. MAF использует смешанную модель для управления жизненным циклом. В пределах одного домена приложения используется сборщик мусора. В пределах канала применяется неявное спонсорство, но счетчик ссылок доступен извне для контроля спонсора.

Рассмотрим сценарий, когда дополнение загружается в другой домен приложения. Внутри принимающего 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 revoke слишком части. Вместо этого `AcquireLifetimeToken()` возвращает идентификатор маркера жизненного цикла, и этот идентификатор должен быть использован для вызова метода `RevokeLifetimeToken()`. Таким образом, эти методы всегда вызываются парами.

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

Финализатор объясняется в главе 12.

В сценариях с загрузкой дополнения в новый домен приложения можно избавиться от загруженного кода, когда дополнение уже не нужно. MAF использует простую 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. 36.7.

Таблица 36.7. Члены `ContractHandle`

Член `ContractHandle`	Описание
`Contract`	При конструировании класса `ContractHandle` объект, реализующий `IContract`, может быть присвоен для сохранения ссылки на него. Свойство Contract возвращает этот объект.
`Dispose()`	Метод `Dispose()` может быть вызван вместо ожидания, когда сборщик мусора выполнит финализацию для аннулирования маркера жизненного цикла.
`AppDomainOwner()`	`AppDomainOwner()` — статический метод класса `ContractHandle`, возвращающий адаптер дополнения, если он владеет доменом приложения, переданным в этот метод.
`ContractOwnsAppDomain()`	С помощью статического метода `ContractOwnsAppDomain()` вы можете 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. Принимающее приложение разрабатывается заранее с учетом будущих дополнений. К дополнению предъявляется требование, чтобы была возможность загрузки старых версий дополнения новыми версиями приложения. Также эта пара должна работать и в противоположном направлении: старые приложения должны запускать новые версии дополнений. Но что, если изменится контракт?

`System.AddIn` полностью независим от реализаций хост-приложения и дополнений. Это обеспечивается концепцией канала, состоящего из семи частей.

Пример дополнения

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

Вам нужно создать решение с шестью проектами библиотек и одним консольным приложением. Проекты простого приложения представлены в табл. 36.8. В таблице

перечислены сборки, ссылки на которые понадобятся. Со ссылками на другие проекты внутри решения вам понадобится установить свойство `Copy Local` 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 хост-приложения. Также вам нужно будет изменить выходной путь сгенерированных сборок, чтобы сборки копировались в правильные каталоги канала.

Таблица 36.8. Описание решения

Проект	Ссылки	Выходной путь	Описание
`CalcContract`	`System.AddIn.Contract`	`..\Pipeline\Contracts\`	Эта сборка содержит контракт для взаимодействия с дополнением. Контракт определен 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, ссылающийся на сборку представления дополнения. Эта сборка содержит реализацию дополнения.

`CalcAddInAdapter`	`System.AddIn`
	`System.AddIn.Contract`
	`CalcView`
	`CalcContract`
		`..\Pipeline\AddInSideAdapters\`	`CalcAddInAdapter` соединяет представление 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, и 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, нуждается в ссылках на оба эти проекта.

`HostApp`	`System.AddIn` `HostView`		Принимающее хост-приложение, активизирующее дополнение.

Контракт калькулятора

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

В следующем коде вы можете видеть контракт, определенный для примера приложения — калькулятора. Приложение определяет контракт с методами `GetOperations()` и `Operate()`. Метод `GetOperations()` возвращает список математических операций,

поддерживаемых дополнением калькулятора. 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.

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

Представление дополнения калькулятора

Представление дополнения переопределяет контракт, 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 математических вычислений со свойствами 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 вы можете 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.AddIns
{
  [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; }
  }
}
```

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 `Calculator ViewToContractAddInAdapter`. Эти классы реализуют интерфейсы `IOperationContract` и `ICalculatorContract`. Методы базового интерфейса `IContract` могут быть реализованы порождением 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.AddIns
{
  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;
    }
  }
}
```

Класс `CalculatorViewToContractAddInAdapter` очень похож на `OperationViewT oContractAddInAdapter`: он происходит от `ContractBase`, наследуя реализацию ин-

терфейса 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.AddIns
{
  [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);
    }
  }
}
```

> Поскольку классы адаптеров вызываются системой рефлексии .NET, с этими классами можно использовать модификатор доступа internal. Поскольку эти классы — деталь реализации, применять его будет хорошей идеей.

Дополнение калькулятора

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 и 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.AddIns
{
  [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));
      }
    }
  }
}
```

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 методы, определенные здесь, вызываются приложением хоста.

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

```
using System.Collections.Generic;
namespace Wrox.ProCSharp.AddIns
{
  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; }
  }
}
```

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 принимающее хост-приложение.

```
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
  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);
    }
  }
}
```

Класс `CalculatorContractToViewHostAdapter` реализует методы абстрактного класса представления хоста `Calculator` и переадресует вызов контракту. Опять-таки, вы можете видеть `ContractHandle`, хранящий ссылку на контракт, что подобно преобразованиям типа адаптера стороны дополнения. На этот раз преобразование типа идет в другом направлении, чем в случае адаптера дополнения.

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

```
using System.Collections.Generic;
using System.AddIn.Pipeline;
namespace Wrox.ProCSharp.AddIns
{
  [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);
    }
  }
}
```

Хост калькулятора

Простое принимающее хост-приложение использует технологию WPF. Вы можете видеть пользовательский интерфейс этого приложения на рис. 36.3. Вверху находится список доступных дополнений. Слева — операции активного дополнения. Когда вы выбираете операцию, которая должна быть вызвана, отображаются операнды. После ввода значений операндов может быть вызвана операция дополнения.

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

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

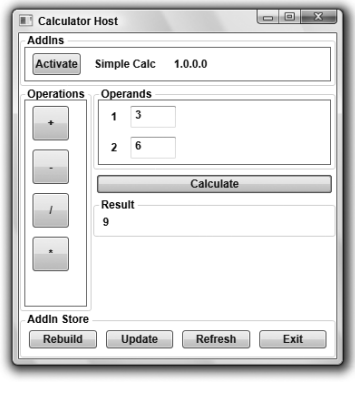

Рис. 36.3. Хост калькулятора

Информацию о шаблонах элементов вы найдете 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>
```

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. Первый параметр метода `AddInStore.FindAddIns()` передает абстрактный класс `Calculator`, определенный представлением хоста, чтобы найти все дополнения из хранилища, применимые к контракту. Второй параметр передает каталог канала, прочитанный из конфигурационного файла приложения. В примере приложения, доступном на прилагаемом компакт-диске, вы должны изменить каталог в конфигурационном файле приложения, чтобы он соответствовал вашей структуре каталогов.

```
using System;
using System.AddIn.Hosting;
using System.AddIn.Pipeline;
using System.IO;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using Wrox.ProCSharp.AddIns.Properties;
namespace Wrox.ProCSharp.AddIns
{
  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();
  }
}
//...
```

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 канала 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. 36.3 вы можете видеть рядом со списком доступных дополнений кнопку 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 выбрать 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 канала и хост-приложение созданы. Канал работает, и добавление к хост-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
```

Резюме

В этой главе были представлены концепции новой технологии .NET 3.5: каркаса управляемых дополнений (Managed Add-In Framework — MAF).

MAF использует концепцию канала для обеспечения независимости между сборками принимающего хост-приложения и дополнений. Четко определенный контракт отделяет представление хоста от представления дополнения. Адаптеры обеспечивают возможность обеим сторонам изменяться независимо друг от друга.

Следующая глава — первая из трех, посвященных разработке пользовательского интерфейса с ASP.NET.