ГЛАВА
9 Точки соединения

Обзор точек соединения

Объект реализует один или несколько интерфейсов, чтобы предоставить доступ к своей функциональности. Термином “точки соединения” (connection points) обозначают концептуально противоположные элементы, которые позволяют объекту вызывать один или несколько интерфейсов.

Метод `QueryInterface` позволяет объекту получить от объекта указатель на интерфейс, реализуемый объектом. Точки соединения позволяют клиенту выдавать объекту указатель на интерфейс, реализованный клиентом. В первом случае клиент использует полученный интерфейсный указатель для вызова методов, предоставляемых объектом. Во втором — объект использует полученный интерфейсный указатель для вызова методов, предоставляемых клиентом.

Более тщательное изучение обоих случаев показывает, что `QueryInterface` позволяет клиенту получить от объекта только интерфейсы, которые клиент способен вызвать. Точки соединения позволяют клиенту предоставить объекту только интерфейсы, которые объект способен вызвать.

Соединение (connection) состоит из двух частей: объекта, вызывающего методы определенного интерфейса (этот объект называется *источником* — source или *точкой соединения* — connection point), и объекта, реализующего эти интерфейсы и принимающего вызовы (этот объект называется *приемником* — sink). Общая структура показана на рис. 9.1. Используя терминологию из предыдущих абзацев, объект является источником и вызывает методы интерфейсов приемника. Клиент является приемником и реализует интерфейсы. Некоторые дополнительные сложности связаны с тем, что один источник может взаимодействовать с несколькими приемниками.

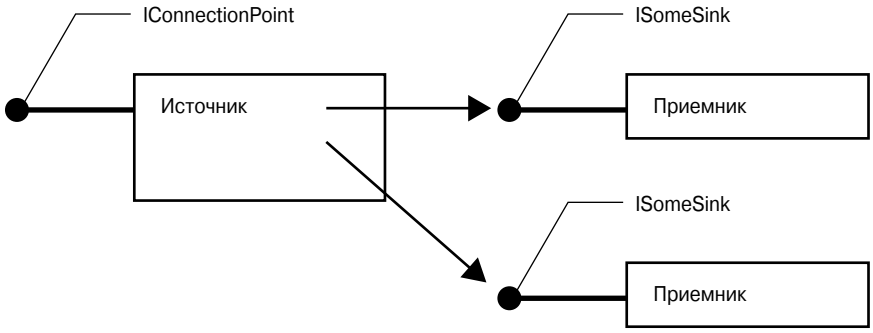

Рис. 9.1. Соединение с двумя приемниками

Интерфейс IConnectionPoint

Клиент использует реализацию интерфейса IConnectionPoint в источнике для установки соединения. Вот определение этого интерфейса.

```
interface IConnectionPoint : IUnknown {
    HRESULT GetConnectionInterface ([out] IID* pIID);
    HRESULT GetConnectionPointContainer (
        [out] IConnectionPointContainer** ppCPC);
    HRESULT Advise ([in] IUnknown* pUnkSink,
        [out] DWORD* pdwCookie);
    HRESULT Unadvise ([in] DWORD dwCookie);
    HRESULT EnumConnections ([out] IEnumConnections** ppEnum);
}
```

Метод GetConnectionInterface возвращает идентификатор интерфейса (IID), соответствующий интерфейсу приемника, к которому обращается точка соединения. В предыдущем примере вызов GetConnectionInterface вернет нам IID_ISomeSink. Клиент вызывает метод Advise, чтобы установить соединение. Клиент предоставляет соответствующий интерфейсный указатель приемника для точки соединения и получает токен, обозначающий соединение. Позже клиент может вызвать метод Unadvise, указывая токен для разрыва соединения. Метод EnumConnections возвращает стандартное перечисление COM, которое клиент может использовать для перебора всех соединений, содержащихся в точке соединения. Последний метод — GetConnectionPoint Container, с которым связана еще одна сложность.

Пока что наша структура позволяла источнику обращаться только к одному определенному интерфейсу. Источник хранит список клиентов, которые хотят получить вызовы от этого интерфейса. Когда источник определяет, что ему нужно вызвать один из методов интерфейса приемника, источник просматривает свой список объектов-приемников, вызывая этот метод для каждого объекта-приемника. Чего в нашей структуре нет (по крайней мере, на данный момент) — это возможности для объекта отправлять вызовы нескольким разным интерфейсам. Точнее говоря, наша проблема заключается в следующем: наш объект может поддерживать несколько соединений с одной точкой соединения, но как объект может поддерживать несколько разных точек соединения?

Решение данной проблемы — низвести объект-источник до уровня подобъекта и создать инкапсулирующий объект, который будет называться соединяемым и будет служить контейнером подобъектов-источников (как на рис. 9.2). Клиент использует метод GetConnectionPointContainer объекта-источника, чтобы получить указатель на соединяемый объект. Соединяемый объект реализует интерфейс IConnection PointContainer.

Реализация IConnectionPointContainer показывает, что объект COM поддерживает точки соединения, а точнее, что он может предоставить точку соединения (подобъект-источник) для каждого интерфейса приемника, который соединяемый объект может вызвать. В дальнейшем клиенты используют эту точку соединения так, как описано выше, чтобы установить соединение.

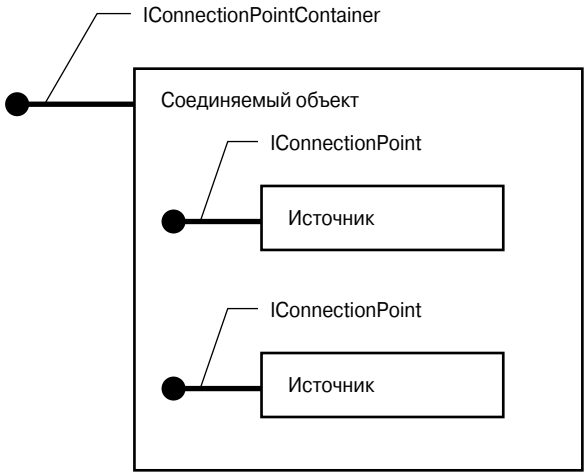

Рис. 9.2. Соединяемый объект

Интерфейс IConnectionPointContainer

Вот определение интерфейса `IConnectionPointContainer`.

```
interface IConnectionPointContainer : IUnknown {
  HRESULT EnumConnectionPoints (
    [out] IEnumConnectionPoints ** ppEnum);
  HRESULT FindConnectionPoint ([in] REFIID riid,
    [out] IConnectionPoint** ppCP);
}
```

Мы вызываем метод `FindConnectionPoint`, чтобы получить интерфейсный указатель `IConnectionPoint` подобъекта-источника, от которого исходят вызовы к интерфейсу, обозначенному идентификатором `riid`. Метод `EnumConnectionPoints` возвращает стандартный перечислитель COM, реализующий интерфейс `Ienum ConnectionPoints`. Этот интерфейс можно использовать для получения интерфейсных указателей `IConnectionPoint` для каждой точки соединения, поддерживаемой соединяемым объектом.

Чаще всего клиент, который хочет установить соединение с соединяемым объектом, выполняет следующие операции (обработка ошибок удалена для краткости).

```
CComPtr<IUnknown>
  spSource = /* Задаем источник событий */ ;
CComPtr<_ISpeakerEvents>
  spSink = /* Приемник, реагирующий на события */ ;
DWORD dwCookie ;

CComPtr<IConnectionPointContainer> spcpc;
HRESULT hr = spSource.QueryInterface (&spcpc);

CComPtr<IConnectionPoint> spcp ;
hr = spcpc->FindConnectionPoint(__uuidof(_ISpeakerEvents),
  &spcp);
hr = spcp->Advise (spSink, &dwCookie) ; // Устанавливаем соединение
// Время идет, приходят обратные вызовы...
hr = spcp->Unadvise (dwCookie) ;        // Разрываем соединение
```

Собственно говоря, ATL предоставляет две полезные функции, устанавливающие и разрывающие соединение между источником и приемником. Функция `AtlAdvise` создает соединение между точкой соединения соединяемого объекта (заданной параметрами `pUnkCP` и `iid`) и реализацией интерфейса приемника (заданной параметром `pUnk`), а затем возвращает регистрационный код для соединения в параметре `pdw`. Функция `AtlUnadvise` приказывает точке соединения соединяемого объекта разорвать соединение, указанное в параметре `dw`.

```
ATLAPI AtlAdvise(IUnknown* pUnkCP, IUnknown* pUnk,
                 const IID& iid, LPDWORD pdw);
ATLAPI AtlUnadvise(IUnknown* pUnkCP, const IID& iid, DWORD dw);
```

Эти функции используются следующим образом.

```
DWORD dwCookie;
// Устанавливаем соединение
hr = AtlAdvise(spSource, spSink, __uuidof(_ISpeakerEvents),
  &dwCookie);
// ... Получаем обратные вызовы ...
// Разрываем соединение
hr = AtlUnadvise(spSource, __uuidof(_ISpeakerEvents), dwCookie);
```

Как видите, чтобы установить соединение, нужен интерфейсный указатель на соединяемый объект, интерфейсный указатель на объект, реализующий интерфейс приемника, и ID интерфейса приемника.

Точки соединения широко применяются в элементах управления ActiveX. Эти элементы управления часто являются источниками событий. Источник событий реализует возможности обратных вызовов для событий в виде вызовов методов в указанном интерфейсе приемника событий. Обычно контейнер элементов управления ActiveX реализует интерфейс приемника событий, чтобы получать определенные события от содержащихся в этом контейнере элементов. Используя точки соединения, контейнер элементов управления соединяет источник событий в элементе управления (точке соединения) и приемник событий в контейнере. Когда происходит событие, точка соединения вызывает соответствующий этому событию метод интерфейса приемника в каждом приемнике, соединенном с точкой соединения.

Следует заметить, что точки соединения — очень обобщенный механизм, поэтому их использование во многих случаях не очень эффективно[1]. Точки соединения наиболее эффективны, если неизвестно, сколько клиентов могут установить соединения с различными интерфейсами приемников. Кроме того, протоколы работы с точками соединения хорошо известны; соответственно, их могут использовать для связи между собой объекты, не знающие о внутренней структуре друг друга. Если вы создаете и источники, и приемники, то можете создать собственный протокол обмена интерфейсными указателями, который будет проще, чем протокол точек соединения.

[1] Для элементов управления ActiveX и других объектов, работающих в рамках процессов, точки соединения вполне эффективны, но если важны переходы из процесса в процесс, их использование дает плохие результаты. Если хотите получить более подробную информацию, почитайте, к примеру, книгу *Effective COM* Дона Бокса, Кейт Браун (Keith Brown), Тима Эвальда (Tim Ewald) и Криса Селлса (Chris Sells) — издательство Addison-Wesley, 1998 год.

Создание соединяемого объекта с помощью ATL

Пример проблемы и ее решение

Давайте создадим COM-объект Demagogue, изображающий оратора. Основанный на ATL класс CDemagogue реализует интерфейс ISpeaker. При получении запроса Speak оратор может шептать, говорить или кричать, в зависимости от значения параметра Volume.

```
interface ISpeaker : IDispatch {
    [propget, id(1)] HRESULT Volume([out, retval] long *pVal);
    [propput, id(1)] HRESULT Volume([in] long newVal);
    [propget, id(2)] HRESULT Speech([out, retval] BSTR *pVal);
    [propput, id(2)] HRESULT Speech([in] BSTR newVal);
    [id(3)] HRESULT Speak();
};
```

Произнесение речей генерирует уведомления о событиях в интерфейсе диспетчеризации событий _ISpeakerEvents, и получатели уведомлений будут "слышать" эти речи. Многие клиенты могут получать уведомления о событиях только в том случае, если интерфейс источника является интерфейсом диспетчеризации.

```
dispinterface _ISpeakerEvents {
properties:
methods:
    [id(1)] void OnWhisper(BSTR bstrSpeach);
    [id(2)] void OnTalk(BSTR bstrSpeach);
    [id(3)] void OnYell(BSTR bstrSpeach);
};
```

Знак подчеркивания в начале имени интерфейса — это условное обозначение, обнаруживая которое многие программы просмотра библиотек типов не отображают имя этого интерфейса. Поскольку интерфейс событий — это деталь реализации, обычно такой интерфейс не стоит открывать для доступа из скриптовых языков.

Обратите внимание на то, что компилятор MIDL (Microsoft Interface Definition Language) добавляет префикс DIID_ к имени интерфейса dispinterface, когда генерирует именованный GUID. Так что для данного интерфейса именованный GUID — это DIID_ISpeakerEvents.

Соответственно, приведенное ниже определение coclass описывает объекты Demagogue. Я добавил в него интерфейс, позволяющий нам задать новое имя объекта, если нам не нравится используемое по умолчанию (Demosthenes).

```
coclass Demagogue {
    [default]          interface     IUnknown;
                       interface     ISpeaker;
                       interface     INamedObject;
    [default, source] dispinterface _ISpeakerEvents;
};
```

Начнем с класса CDemagogue. Это основанный на ATL создаваемый класс для однопоточных апартаментов, предназначенный для представления объектов Demagogue.

```
class ATL_NO_VTABLE CDemagogue :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CDemagogue, &CLSID_Demagogue>,
```

```
    public ISupportErrorInfo,
    public IDispatchImpl<ISpeaker, &IID_ISpeaker,
        &LIBID_ATLINTERNALSLib>,
    ...
{ ... };
```

Семь шагов к соединяемому объекту

Для создания соединяемого объекта с помощью ATL нужно выполнить следующие действия.

1. Реализовать интерфейс `IConnectionPointContainer`.
2. Реализовать метод `QueryInterface` так, чтобы он отвечал на обращения к интерфейсу `IID_IConnectionPointContainer`.
3. Реализовать интерфейс `IConnectionPoint` для каждого интерфейса источника, поддерживаемого соединяемым объектом.
4. Предоставить карту соединений, которая представляет собой таблицу, сопоставляющую IID с реализациями точек соединения.
5. Обновить определение `coclass` для соединяемого объекта в файле IDL, указав в нем все интерфейсы источников. Для каждого интерфейса источника необходим атрибут `[source]`. Основной интерфейс источника должен иметь атрибуты `[default, source]`.
6. Обычно нужно реализовать вспомогательные методы, вызывающие методы всех подсоединенных приемников.
7. Вызывать вспомогательные методы в требуемые моменты времени.

Добавление требуемого базового класса в соединяемый объект

Чтобы объект мог генерировать события с помощью протокола точек соединения, этот объект должен быть соединяемым. А это значит, что объект должен реализовывать интерфейс `IConnectionPointContainer`. Можно воспользоваться предоставляемой ATL реализацией `IConnectionPointContainer` и `IConnectionPoint`, унаследовав соответствующие классы в классе соединяемого объекта.

Шаг 1. Сделаем класс `CDemagogue` производным от класса-шаблона `IConnection PointContainerImpl`. Этот шаблон использует единственный параметр — имя производного класса. Это наследование даст нам соединяемые объекты с реализацией интерфейса `IConnectionPointContainer`.

```
class ATL_NO_VTABLE CDemagogue :
    ...
    public IConnectionPointContainerImpl<CDemagogue>,
    ...
```

Изменения в COM_MAP соединяемого объекта

Шаг 2. Добавляя в класс реализацию нового интерфейса, нужно добавлять поддержку этого интерфейса в метод `QueryInterface`. ATL реализует `QueryInterface` для объектов, просматривая таблицу `COM_MAP` в поисках записи с соответствующим IID. Чтобы показать, что объект поддерживает `IConnectionPointContainer`, добавьте в эту таблицу макрос `COM_INTERFACE_ENTRY` для данного интерфейса.

```
BEGIN_COM_MAP(CDemagogue)
    ...
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
    ...
END_COM_MAP ()
```

Добавление точек соединения

В контейнере точек соединения должна содержаться коллекция этих точек (в противном случае вообще непонятно, зачем этот контейнер нужен). Для каждого интерфейса источника, поддерживаемого соединяемым объектом, необходим подобъект — точка соединения. Этот подобъект с логической точки зрения является отдельным объектом, реализующим интерфейс `IConnectionPoint`.

Шаг 3. Чтобы создать подобъекты точек соединения, нужно унаследовать в классе соединяемого объекта шаблон `IConnectionPointImpl` (один или несколько раз — по одному разу для каждого интерфейса источника, поддерживаемого соединяемым объектом). Это наследование предоставляет соединяемому объекту одну или несколько реализаций интерфейса `IConnectionPoint` в подобъектах с независимыми счетчиками ссылок. Шаблон `IConnectionPointImpl` использует три параметра: имя класса соединяемого объекта, IID интерфейса источника точки соединения и (если оно задано) имя класса, управляющего коллекциями.

```
class ATL_NO_VTABLE CDemagogue :
  ...
  public IConnectionPointContainerImpl<CDemagogue>,
  public IConnectionPointImpl<CDemagogue, &DIID__ISpeakerEvents>
  ...
```

Расположение точек соединения

Для любой реализации `IConnectionPointContainer` требуется определенная базовая информация: список объектов точек соединения и IID, поддерживаемых каждым таким объектом. ATL-реализация использует таблицу, называемую *картой точек соединения* (connection point map), в которой задается вся требуемая информация. Эта таблица определяется в объявлении класса соединяемых объектов с помощью трех макросов ATL.

Макрос `BEGIN_CONNECTION_POINT_MAP` обозначает начало таблицы. Единственный его параметр — имя класса соединяемого объекта. Каждый макрос `CONNECTION_POINT_ENTRY` добавляет в таблицу одну запись, соответствующую одной точке соединения. Единственный параметр этого макроса — IID интерфейса, поддерживаемого этой точкой соединения.

Заметьте, что макрос `CONNECTION_POINT_ENTRY` использует значение IID, а макрос `COM_INTERFACE_ENTRY` — имя класса интерфейса. Исторически для получения GUID интерфейса к имени класса этого интерфейса всегда добавлялся префикс `IID_`. Ранние версии ATL получали IID именно таким способом.

Однако у интерфейсов источников нет таких строгих стандартов для имен. В разных версиях MFC, MKTYPLIB и MIDL к `dispinterface` добавляются разные префиксы. Макрос `CONNECTION_POINT_ENTRY` не полагается на наличие определенного префикса, поэтому IID нужно указывать явно. По умолчанию ATL получает IID класса с помощью ключевого слова `__uuidof`.

Макрос END_CONNECTION_POINT_MAP генерирует маркер конца таблицы и код, возвращающий адрес этой таблицы и ее размер.

Шаг 4. Вот карта точек соединения для класса CDemagogue.

```
BEGIN_CONNECTION_POINT_MAP(CDemagogue)
    CONNECTION_POINT_ENTRY(__uuidof(_ISpeakerEvents))
END_CONNECTION_POINT_MAP()
```

Обновление определения coclass для поддержки интерфейсов источников

Шаг 5. Клиенты часто просматривают библиотеки типов, в которых описываются объекты-источники событий, чтобы узнать некоторые детали реализации, например интерфейсы источников. Нужно, чтобы в описании coclass для объекта-источника была запись об интерфейсе источника.

```
coclass Demagogue
{
    [default]          interface     IUnknown;
                       interface     ISpeaker;
                       interface     INamedObject;
    [default, source] dispinterface _ISpeakerEvents;
};
```

Механизмы генерации событий

На данный момент у нас есть соединяемый объект Demagogue, являющийся контейнером для точек соединения, и одна точка соединения. Реализация, рассмотренная до сих пор, позволяет клиенту зарегистрировать интерфейс, связывающийся с точкой соединения. Все перечислители будут работать. Клиент даже сможет при желании разорвать соединение. Однако соединяемый объект никогда не будет генерировать события. А это значит, что пока что мы не достигли никаких практически полезных результатов — нашу работу нужно закончить. Соединяемому объекту требуется возможность вызывать методы интерфейсов приемников, т.е. генерировать события.

Чтобы сгенерировать событие, нужно вызвать соответствующий метод интерфейса приемника через каждый интерфейсный указатель, зарегистрированный для точки соединения. Эта задача довольно сложна, и вы, вероятно, посчитаете удобным создать в классе соединяемого объекта специальные вспомогательные методы для генерации событий. Вам понадобится по одному вспомогательному методу для каждого метода каждого интерфейса, связанного с точками соединения вашего объекта.

Чтобы сгенерировать событие, нужно вызвать соответствующий метод требуемого интерфейса приемника. Это делается для каждого интерфейса приемника, зарегистрированного для точки соединения. А это значит, что нужно просматривать список таких зарегистрированных интерфейсов и вызывать метод через каждый интерфейсный указатель. Вам интересно, где и как хранится этот список интерфейсов? Сейчас мы в этом разберемся.

У каждого объекта базового класса IConnectionPointImpl (т.е. у каждой точки соединения) есть поле m_vec, которое ATL объявляет как вектор указателей IUnknown. Однако вам не нужно вызывать QueryInterface, чтобы извлечь указатели из этого вектора; реализация метода IConnectionPoint::Advise, предоставляемая ATL, уже сделала это за вас. Например, вектор в точке соединения, связанной с DIID_ISpeakerEvents, на самом деле содержит указатели _ISpeakerEvents.

По умолчанию m_vec — это объект класса CComDynamicUnkArray, т.е. динамически выделяемый массив интерфейсных указателей IUnknown, каждый из которых является интерфейсным указателем для приемника, связанного с точкой соединения. Размер этого массива увеличивается по мере надобности, поэтому реализация по умолчанию поддерживает неограниченное количество соединений.

В качестве альтернативы, выбирая IConnectionPointImpl в качестве базового класса, можно указать, что m_vec — это объект класса CComUnkArray, в котором будет храниться фиксированное количество интерфейсных указателей. Используйте эту возможность, если вы хотите поддерживать определенное максимальное количество соединений. Кроме того, в ATL есть явный шаблон, CComUnkArray<1>, специально предназначенный для поддержки единственного соединения.

Шаг 6. Чтобы сгенерировать событие, мы просматриваем массив и для каждой записи со значением, отличающимся от NULL, вызываем метод интерфейса приемника, связанный с этим событием. Вот простой вспомогательный метод, генерирующий событие OnTalk интерфейса _ISpeakerEvents.

Обратите внимание на то, что m_vec будет однозначной, только если мы будем использовать единственную точку соединения.

```
HRESULT Fire_OnTalk(BSTR bstrSpeech)
{
    CComVariant arg, varResult;
    int nIndex, nConnections = m_vec.GetSize();

    for (nIndex = 0; nIndex < nConnections; nIndex++) {
        CComPtr<IUnknown> sp = m_vec.GetAt(nIndex);
        IDispatch* pDispatch =
            reinterpret_cast<IDispatch*>(sp.p);
        if (pDispatch != NULL) {
            VariantClear(&varResult);
            arg = bstrSpeech;
            DISPPARAMS disp = { &arg, NULL, 1, 0 };
            pDispatch->Invoke(0x2, IID_NULL, LOCALE_USER_DEFAULT,
                DISPATCH_METHOD, &disp, &varResult, NULL, NULL);
       }
    }
    return varResult.scode;
}
```

Генератор заместителей точек соединения в ATL

Писать вручную вспомогательные методы для вызова методов интерфейсов приемников скучно, к тому же при этом легко можно наделать ошибок. Дополнительные сложности вносит то, что интерфейс приемника может быть нестандартным интерфейсом COM или dispinterface. Работать с интерфейсом dispinterface (с помощью IDispatch::Invoke) гораздо сложнее, чем сделать обратный вызов через vtbl. К сожалению, обратные вызовы для dispinterface встречаются чаще всего, поскольку это единственный механизм, поддерживаемый в большинстве скриптовых языков, Internet Explorer и контейнерах элементов управления ActiveX.

В IDE Visual Studio 2005 есть инструмент для генерации исходного кода, создающий классы точек соединений со всеми необходимыми вспомогательными методами для выполнения обратных вызовов в указанных интерфейсах точек соединения. Чтобы вос-

пользоваться этим инструментом, в панели Class View в Visual Studio 2005 щелкните правой кнопкой мыши на значке класса C++, который хотите сделать источником событий. В появившемся контекстном меню щелкните на пункте Add Connection Point. Откроется окно мастера Implement Connection Point Wizard, показанное на рис. 9.3.

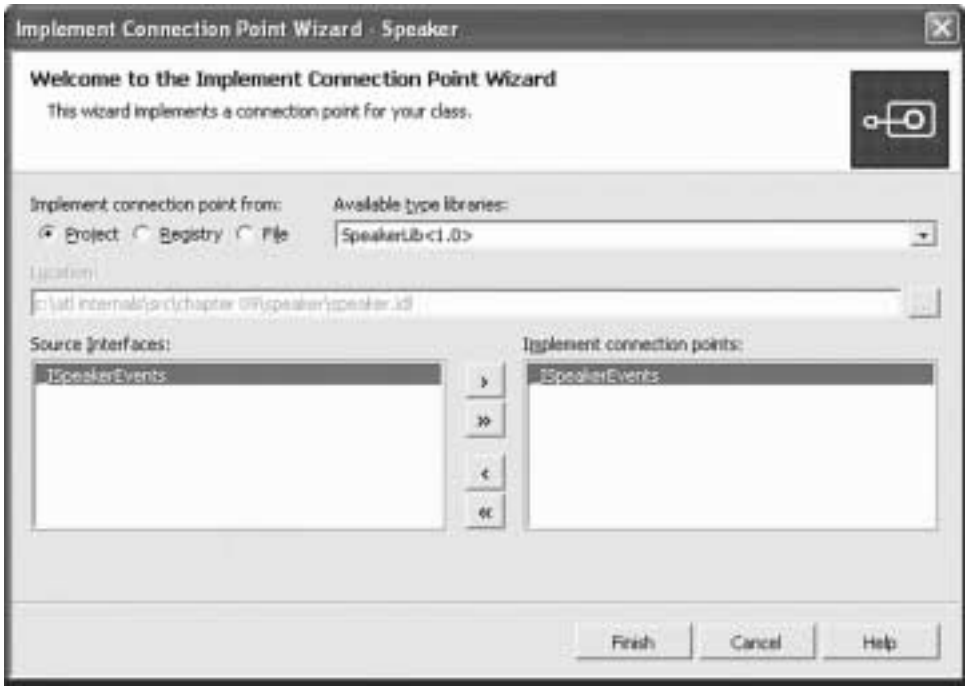

Рис. 9.3. Диалоговое окно Implement Connection Point Wizard

Этот мастер создает один или несколько классов (объявляемых и определяемых в заданном заголовочном файле), которые представляют указанные интерфейсы и их методы. Чтобы воспользоваться генератором кода, вам потребуется библиотека типов, описывающая требуемый интерфейс событий. Генератор кода просматривает эту библиотеку и генерирует класс, производный от `IConnectionPointImpl`, в котором есть вспомогательные методы, генерирующие события для каждого метода интерфейса. Вам нужно только задать имя генерируемого класса в качестве одного из базовых классов в классе вашего соединяемого объекта. Этот базовый класс реализует конкретную точку соединения и содержит все необходимые вспомогательные методы для генерации событий.

Код, сгенерированный мастером

Генератор кода создает класс-шаблон с именем `CProxy_<ИмяИнтерфейсаПриемника>`. Этот класс-заместитель использует один параметр — имя класса соединяемого объекта. Класс-заместитель является производным от специализации `IConnectionPointImpl` для интерфейса источника.

Вот код, сгенерированный мастером Implement Connection Point Wizard для описанного выше интерфейса `_ISpeakerEvents`.

```
#pragma once
template<class T>

class CProxy_ISpeakerEvents :
    public IConnectionPointImpl<T, &__uuidof(_ISpeakerEvents)> {
public:
    HRESULT Fire_OnWhisper(BSTR bstrSpeech) {
        HRESULT hr = S_OK;
        T * pThis = static_cast<T*>(this);
        int cConnections = m_vec.GetSize();

        for (int iConnection = 0;
            iConnection < cConnections;
            iConnection++)
        {
            pThis->Lock();
            CComPtr<IUnknown> punkConnection =
                m_vec.GetAt(iConnection);
            pThis->Unlock();

            IDispatch * pConnection =
                static_cast<IDispatch*>(punkConnection.p);

            if (pConnection) {
                CComVariant avarParams[1];
                avarParams[0] = bstrSpeech;
                DISPPARAMS params = { avarParams, NULL, 1, 0 };
                hr = pConnection->Invoke(DISPID_ONWHISPER,
                    IID_NULL,
                    LOCALE_USER_DEFAULT,
                    DISPATCH_METHOD,
                    &params, NULL, NULL, NULL);
          }
        }
        return hr;
    }

// Другие методы такие же, они удалены для краткости

};
```

Использование кода заместителя точки соединения

Ниже приведены некоторые основные элементы объявления класса соединяемых объектов CDemagogue. Единственное отличие этого объявления от ранее рассмотренных примеров — использование в качестве базового класса для точки соединения не IConnectionPointImpl, а сгенерированного класса заместителя, CProxy_ISpeakerEvents<CDemagogue>.

```
class ATL_NO_VTABLE CDemagogue :
...
    public IConnectionPointContainerImpl<CDemagogue>,
    public CProxy_ISpeakerEvents<CDemagogue>, ... {

BEGIN_COM_MAP(CDemagogue)
...
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
```

```
...
END_COM_MAP()

BEGIN_CONNECTION_POINT_MAP(CDemagogue)
    CONNECTION_POINT_ENTRY(__uuidof(_ISpeakerEvents))
END_CONNECTION_POINT_MAP()
...
};
```

Генерация событий

Шаг 7. Последнее, что остается сделать, чтобы все заработало — это сгенерировать нужные события в нужные моменты времени. Эти моменты зависят от приложения, но вот один пример.

Объект класса `CDemagogue` произносит речь, когда клиент вызывает его метод `Speak`. Этот метод, в зависимости от значения параметра `volume`, должен шептать, говорить или кричать. Он делает это, вызывая методы событий `OnWhisper`, `OnTalk` или `OnYell` всех клиентов, связанных с интерфейсом `_ISpeakerEvents`.

```
STDMETHODIMP CDemagogue::Speak() {
    if (m_nVolume <= -100)
        return Fire_OnWhisper(m_bstrSpeech);

    if (m_nVolume >= 100)
        return Fire_OnYell(m_bstrSpeech);

    return Fire_OnTalk(m_bstrSpeech);
}
```

Завершающие штрихи

Сделанные к данному моменту изменения полностью реализуют протокол точки соединения. Однако стоит сделать еще одно изменение, которое упростит использование нашего соединяемого объекта клиентами. Соединяемый объект должен предоставлять удобный доступ к информации о поддерживаемых этим объектом интерфейсах.

Точнее говоря, многие клиенты, которые хотят получать уведомления о событиях от соединяемого объекта, могут запросить у объекта интерфейс `IProvideClassInfo2`. Это делают, например, Microsoft Internet Explorer, Visual Basic и основанные на ATL контейнеры для элементов управления ActiveX. Клиент вызывает метод `GetGUID` этого интерфейса с параметром `GUIDKIND_DEFAULT_SOURCE_DISP_IID`, чтобы получить IID основного интерфейса `dispinterface`, поддерживаемого соединяемым объектом. Это IID интерфейса, указанного в описании `coclass` объекта с атрибутами `[default, source]`.

Поддержка интерфейса `IProvideClassInfo2` дает произвольным клиентам возможность определить IID основного источника событий и с помощью этого IID установить соединение. Заметьте, что IID, получаемый в результате вызова `GetGUID`, должен соответствовать интерфейсу `dispinterface`; он не может соответствовать стандартному интерфейсу, работающему через `vtbl` (производному от `IUnknown`).

Если соединяемый объект не может ответить на запрос интерфейса `Iprovide ClassInfo2`, некоторые клиенты запрашивают интерфейс `IProvideClassInfo`.

Клиент может использовать этот интерфейс, чтобы получить указатель ITypeInfo для класса соединяемого объекта. Приложив определенные усилия, клиент может с помощью этого указателя определить, какой интерфейс источника по умолчанию поддерживает соединяемый объект. Интерфейс IProvideClassInfo2 является производным от IProvideClassInfo, так что, реализовав первый из них, вы автоматически реализуете и второй.

Поскольку большинство соединяемых объектов должно реализовывать интерфейс IProvideClassInfo2, ATL предоставляет класс-шаблон для его реализации. Этот класс-шаблон называется IProvideClassInfo2Impl. Он предоставляет реализацию по умолчанию для всех методов интерфейсов IProvideClassInfo2 и IProvideClassInfo. Определение класса выглядит так.

```
template <const CLSID* pcoclsid, const IID* psrcid,
    const GUID* plibid = &CAtlModule::m_libid,
    WORD wMajor = 1, WORD wMinor = 0,
    class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IProvideClassInfo2Impl
    : public IProvideClassInfo2
{ ... }
```

Чтобы использовать эту реализацию в соединяемом объекте, вам нужно унаследовать класс соединяемого объекта от класса IProvideClassInfo2Impl. Последние два параметра шаблона в последующем примере — это номера версии и подверсии библиотеки типов компонента. По умолчанию для них задаются значения 1 и 0. Если вы изменяете номер версии библиотеки типов, нужно соответственно изменить и номер в вызове шаблона. Если вы этого не сделаете, компилятор возмущаться не станет, но работать код не будет.

Всегда указывая явно номер версии, вы защитите себя от неприятностей, возникших из-за того, что вы забыли поменять этот номер.

```
#define LIBRARY_MAJOR 1
#define LIBRARY_MINOR 0

class ATL_NO_VTABLE CDemagogue :
...
    public IConnectionPointContainerImpl<CDemagogue>,
    public CProxy_ISpeakerEvents<CDemagogue>,
    public IProvideClassInfo2Impl<&CLSID_Demagogue,
        &__uuidof(_ISpeakerEvents),
        &LIBID_ATLINTERNALSLib, LIBRARY_MAJOR, LIBRARY_MINOR>,
...
};
```

Кроме того, нужно обновить таблицу интерфейсов, чтобы QueryInterface мог отвечать на запросы интерфейсов IProvideClassInfo и IProvideClassInfo2.

```
BEGIN_COM_MAP(CDemagogue)
...
    COM_INTERFACE_ENTRY(IProvideClassInfo2)
    COM_INTERFACE_ENTRY(IProvideClassInfo)
END_COM_MAP()
```

Наконец, в листинге ниже показаны вместе все изменения, связанные с соединяемым объектом.

```
#define LIBRARY_MAJOR    1
#define LIBRARY_MINOR    0

// dispinterface для событий
dispinterface _ISpeakerEvents {
properties:
methods:
    [id(1)] void OnWhisper(BSTR bstrSpeech);
    [id(2)] void OnTalk(BSTR bstrSpeech);
    [id(3)] void OnYell(BSTR bstrSpeech);
};

// Класс соединяемого объекта
coclass Demagogue {
    [default]         interface      IUnknown;
                      interface      ISpeaker;
                      interface      INamedObject;
    [default, source] dispinterface _ISpeakerEvents;
};

// Класс реализации для coclass Demagogue
class ATL_NO_VTABLE CDemagogue :
...
    public IConnectionPointContainerImpl<CDemagogue>,
    public CProxy_ISpeakerEvents<CDemagogue>,
    public IProvideClassInfo2Impl<&CLSID_Demagogue,
        &__uuidof(_ISpeakerEvents),
        &LIBID_ATLINTERNALSLib, LIBRARY_MAJOR, LIBRARY_MINOR>,
    ... {
public:
BEGIN_COM_MAP(CDemagogue)
    COM_INTERFACE_ENTRY(IConnectionPointContainer)
    COM_INTERFACE_ENTRY(IProvideClassInfo2)
    COM_INTERFACE_ENTRY(IProvideClassInfo)
...
END_COM_MAP()

BEGIN_CONNECTION_POINT_MAP(CDemagogue)
    CONNECTION_POINT_ENTRY(__uuidof(_ISpeakerEvents))
END_CONNECTION_POINT_MAP()
...
};
```

Создание объекта, принимающего события

Теоретически создать объект, принимающий события по одному интерфейсу, несложно. Для этого нужно определить класс, реализующий этот интерфейс, и соединить объект с источником событий. У нас есть класс `Demagogue`, генерирующий события для интерфейса диспетчеризации событий `_ISpeakerEvents`. Давайте определим класс `CEarPolitic`, реализующий этот интерфейс.

```
coclass EarPolitic {
    [default] dispinterface _ISpeakerEvents;
};
```

А теперь реализуем класс с помощью ATL как класс `CEarPolitic`.

```
class ATL_NO_VTABLE CEarPolitic :
  ...
  public _ISpeakerEvents,
  ... {
public:
  ...
BEGIN_COM_MAP(CEarPolitic)
  COM_INTERFACE_ENTRY(IDispatch)
  COM_INTERFACE_ENTRY(_ISpeakerEvents)
...
END_COM_MAP()

// _ISpeakerEvents
  STDMETHOD(GetTypeInfoCount)(UINT* pctinfo);
  STDMETHOD(GetTypeInfo)(UINT itinfo, LCID lcid,
    ITypeInfo** pptinfo);
  STDMETHOD(GetIDsOfNames)(REFIID riid, LPOLESTR* rgszNames,
    UINT cNames, LCID lcid, DISPID* rgdispid);
  STDMETHOD(Invoke)(DISPID dispidMember, REFIID riid, LCID lcid,
    WORD wFlags, DISPPARAMS* pdispparams, VARIANT* pvarResult,
    EXCEPINFO* pexcepinfo, UINT* puArgErr);
};
```

К сожалению, интерфейс событий — это обычно `dispinterface`, поэтому обычный способ реализации интерфейсов для него не подходит. В результате работы компилятора MIDL для интерфейса `_ISpeakerEvents` вы получите только такой фрагмент.

```
MIDL_INTERFACE("A924C9DE-797F-430d-913D-93158AD2D801")
_ISpeakerEvents : public IDispatch
{
};
```

Вместо того чтобы просто реализовать методы интерфейса как обычные методы C++, мы должны реализовать интерфейс `IDispatch` и обеспечить поддержку, как минимум, его метода `Invoke`. Этот метод — довольно неприятная в реализации вещь для любого интерфейса событий.

Другой возможный подход — использовать реализацию `IDispatch`, предоставляемую классом-шаблоном `IDispatchImpl`. К сожалению, класс-шаблон требует параметров, описывающих двойной интерфейс, а не `dispinterface`. Чтобы использовать `IDispatchImpl`, нам придется определить ложный двойной интерфейс, содержащий те же методы диспетчеризации, идентификаторы диспетчеризации и сигнатуры функций, что и интерфейс событий `dispinterface`.

У такого подхода будет больше последствий, чем может показаться. Интерфейсы `dispinterface`, в отличие от обычных интерфейсов COM, не являются неизменными. Если вы не управляете определением интерфейса `dispinterface`, оно может изменяться от версии к версии (вряд ли оно изменится, но все же это возможно). А это значит, что вашему ложному двойному интерфейсу тоже придется изменяться, следовательно, нельзя документировать двойной интерфейс — если вы опишете его в документации, кто-то из клиентов может им воспользоваться. А поскольку нельзя будет описать двойной интерфейс в библиотеке типов (она документирует этот интерфейс), вы не сможете использовать универсальный маршалер, основанный на библиотеке типов. Следовательно, вам понадобятся специальные заглушка и замес-

титель для удаленных вызовов двойного интерфейса. Все это по большей части теоретические рассуждения, поскольку в нашем случае двойной интерфейс является деталью реализации, специфичной для класса реализации, но описанных проблем достаточно, чтобы поискать другое решение.

Можно посмотреть на эту проблему с другой точки зрения. Что если нам нужно принимать одинаковые события из нескольких источников и знать, из какого источника мы приняли событие? Предположим, что нам нужно реализовать класс `EarPolitic`, выполняющий роль судьи, слушающего объекты `Defendant` и `Plaintiff` через интерфейс `_ISpeakerEvents`. Каждый из объектов может быть источником событий `OnWhisper`, `OnSpeak` и `OnYell`, но судья должен следить за тем, кто и что говорит.

Для этого придется реализовать интерфейс `_ISpeakerEvents` несколько раз: по одному разу для каждого источника событий. Чтобы создать в классе несколько реализаций одного интерфейса, нужно, чтобы каждая реализация размещалась в отдельной сущности COM. Два стандартных решения этой проблемы — поэлементная композиция (каждая реализация — во вложенном классе) и нечто наподобие отделяемых интерфейсов (каждая реализация — в отдельном классе).

Классы IDispEventImpl и IDispEventSimpleImpl

Чтобы избавиться от только что описанных проблем, ATL предоставляет два класса-шаблона, `IDispEventImpl` и `IDispEventSimpleImpl`, предоставляющих реализацию интерфейса `IDispatch` для объектов ATL COM. Обычно один из этих классов используется в объекте, который должен принимать события. Оба класса реализуют интерфейс диспетчеризации как объект вложенного класса, образующего сущность COM, отдельную от производного класса. Это значит, что можно наследовать эти классы несколько раз, если нужно реализовать несколько интерфейсов диспетчеризации событий.

Класс `IDispEventImpl` требует использования библиотеки типов, описывающей интерфейс диспетчеризации. Этот класс использует во время выполнения программы оператор `typeinfo`, чтобы преобразовать полученные в методе `Invoke` параметры типа `VARIANT` в требуемые типы и структуру кадра стека при вызове метода-обработчика событий.

```
template <UINT nID, class T, const IID* pdiid = &IID_NULL,
          const GUID* plibid = &GUID_NULL,
          WORD wMajor = 0, WORD wMinor = 0,
          class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IDispEventImpl :
    public IDispEventSimpleImpl<nID, T, pdiid>
{ ... }
```

Класс `IDispEventSimpleImpl` не использует библиотеку типов, и работать с ним немного проще. Его стоит использовать, если у вас нет библиотеки типов или если вы хотите добиться большей эффективности при выполнении программы.

```
template <UINT nID, class T, const IID* pdiid>
class ATL_NO_VTABLE IDispEventSimpleImpl :
    public _IDispEventLocator<nID, pdiid>
{ ... }
```

Используя класс IDispEventSimpleImpl, нужно предоставлять структуру _ATL_FUNC_INFO, в которой содержится информация о параметрах обработчика событий.

```
struct _ATL_FUNC_INFO {
CALLCONV cc;           // Правила вызова
VARTYPE vtReturn;      // Тип VARIANT для возвращаемого значения
SHORT nParams;         // Количество параметров
VARTYPE pVarTypes[_ATL_MAX_VARTYPES]; // Массив параметров
                                      // Тип VARIANT
};
```

Заметьте, что класс IDispEventImpl является производным от IDispEvent SimpleImpl. Класс IDispEventSimpleImpl вызывает обработчик события, исходя из информации, хранящейся в структуре _ATL_FUNC_INFO. Эту структуру можно предоставлять статически (во время компиляции), ссылаясь на структуру в таблице приемников (которую мы рассмотрим позже в этой главе).

Если ссылки на структуру нет, класс IDispEventSimpleImpl вызывает виртуальный метод GetFuncInfoFromId, чтобы получить структуру _ATL_FUNC_INFO для обработчика событий, связанного с указанным DISPID. Чтобы предоставить эту структуру динамически, нужно переопределить метод GetFuncInfoFromId так, чтобы он возвращал эту структуру при вызове. Метод GetFuncInfoFromId необходимо использовать, если нужно вызывать разные обработчики событий в зависимости от специфичных данных, предоставляемых источниками событий.

Вот реализация по умолчанию, предоставляемая классом IDispEventSimpleImpl.

```
// Вспомогательный метод для определения индекса функции по DISPID
virtual HRESULT GetFuncInfoFromId(const IID& iid,
  DISPID dispidMember,
  LCID lcid,
  _ATL_FUNC_INFO& info) {
  ATLTRACE(_T("TODO: Classes using IDispEventSimpleImpl "
    "should override this method\n"));
  ATLASSERT(0);
  ATLTRACENOTIMPL(_T("IDispEventSimpleImpl::GetFuncInfoFromId"));
}
```

Класс IDispEventImpl переопределяет этот виртуальный метод, чтобы создать структуру из данных, полученных с помощью typeinfo из библиотеки типов.

Реализация приемника событий

Самый *простой* способ реализации одного или нескольких приемников событий в объекте ATL без атрибутов — один или несколько раз унаследовать в классе этого объекта класс IDispEventImpl. Наследовать IDispEventImpl нужно по одному разу для каждого интерфейса событий от каждого источника событий. Вот спецификация этого класса-шаблона.

```
template <UINT nID, class T, const IID* pdiid = &IID_NULL,
          const GUID* plibid = &GUID_NULL,
          WORD wMajor = 0, WORD wMinor = 0,
          class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IDispEventImpl :
    public IDispEventSimpleImpl<nID, T, pdiid>
{ ... }
```

Параметр nID задает идентификатор источника событий, уникальный для наследующего класса T. В главе 11, "Элементы управления ActiveX", вы увидите, что если источником событий является внутренний элемент управления, а приемником — составной элемент управления, то этим идентификатором является идентификатор дочернего окна внутреннего элемента управления.

Составной элемент управления может принимать для всех остальных параметров шаблона IDispEventImpl значения по умолчанию, но остальным объектам COM придется задавать все параметры, кроме последнего. Параметр piid указывает GUID интерфейса dispinterface, реализуемого этим классом. Этот интерфейс должен быть описан в библиотеке типов, указанной в параметре plibid, причем номера версии и подверсии должны быть указаны соответственно в параметрах wMajor и wMinor. Параметр tihclass указывает класс с информацией о типах для производного класса T. Обычно предлагаемый по умолчанию класс CComTypeInfoHolder вполне подходит.

Самый эффективный способ реализации одного или нескольких приемников событий в объекте ATL основан на использовании класса IDispEventSimpleImpl и не требует использования библиотеки типов. Однако этот способ требует наличия структуры _ATL_FUNC_INFO, описанной выше. Используя класс IDispEventSimpleImpl, вы должны только указать идентификатор источника событий в параметре nID, производный класс в параметре T и GUID интерфейса dispinterface в параметре piid.

```
template <UINT nID, class T, const IID* pdiid>
class ATL_NO_VTABLE IDispEventSimpleImpl :
    public _IDispEventLocator<nID, pdiid>
{ ... }
```

Чтобы использовать более простой способ, давайте переопределим класс CearPolitic, реализовав в нем интерфейс _ISpeakerEvents дважды: одна реализация будет предназначена для объекта Demagogue — истца и одна для объекта Demagogue — ответчика. У нас есть библиотека типов, так что мы будем использовать класс IDispEventImpl для реализации приемника вызовов от объекта-ответчика. Чтобы продемонстрировать альтернативный способ, для реализации приемника вызовов от объекта-истца мы воспользуемся классом IDispEventSimpleImpl. Для каждого интерфейса событий лучше использовать определение типа, чтобы свести к минимуму количество ошибок.

```
static const int DEFENDANT_SOURCE_ID = 0 ;
static const int PLAINTIFF_SOURCE_ID = 1 ;

class CEarPolitic;

typedef IDispEventImpl<DEFENDANT_SOURCE_ID,
  CEarPolitic,
  &__uuidof(_ISpeakerEvents),
  &LIBID_ATLINTERNALSLib,
  LIBRARY_MAJOR, LIBRARY_MINOR> DefendantEventImpl;

typedef IDispEventSimpleImpl<PLAINTIFF_SOURCE_ID,
  CEarPolitic, &__uuidof(_ISpeakerEvents)> PlaintiffEventImpl;
```

В этом примере мы выбрали значения 0 и 1 в качестве идентификаторов источников событий, хотя можно использовать любые другие числа. А теперь нам нужно унаследовать класс CEarPolitic от двух классов реализации событий.

```
class ATL_NO_VTABLE CEarPolitic :
    ...
    public DefendantEventImpl,
    public PlaintiffEventImpl {

// Здесь должна быть таблица приемников событий

};
```

Таблица приемников событий

Реализация метода Invoke в классе IDispEventSimpleImpl получает обратные вызовы, когда генерируются события. Когда источник событий вызывает метод Invoke, он указывает, какое событие произошло, в параметре DISPID. Реализация Invoke просматривает таблицу приемников событий, чтобы определить, какую функцию нужно вызвать, когда событие DISPID генерируется в интерфейсе диспетчеризации DIID источника SOURCE.

Начало таблицы приемников событий создается с помощью макроса BEGIN_SINK_MAP в объявлении класса, производного от IDispEventImpl или IDispEventSimpleImpl. Каждому уникальному сочетанию значений SOURCE, DIID и DISPID с помощью макросов SINK_ENTRY, SINK_ENTRY_EX и SINK_ENTRY_INFO ставится в соответствие метод-обработчик сообщения.

- SINK_ENTRY(SOURCE, DISPID, func). Этот макрос используется в составных элементах управления, чтобы указать обработчик для заданного события в интерфейсе событий по умолчанию во внутреннем элементе управления. Он подразумевает использование IDispEventImpl и вызов функции AtlAdviseSinkMap для установки соединения. Функция AtlAdviseSinkMap подразумевает, что ваш класс является производным от CWindow. В общем, макрос SINK_ENTRY не особенно полезен для объектов, которые хотят принимать события, но не относятся к пользовательскому интерфейсу.
- SINK_ENTRY_EX(SOURCE, DIID, DISPID, func). Этот макрос позволяет задавать обработчик конкретному событию в конкретном интерфейсе конкретного объекта. Он наиболее полезен для объектов, не относящихся к пользовательскому интерфейсу, которые хотят принимать события, а также для составных элементов управления, которые хотят принимать события от внутренних элементов управления через интерфейсы, отличающиеся от интерфейса по умолчанию.
- SINK_ENTRY_INFO(SOURCE, DIID, DISPID, func, info). Этот макрос аналогичен SINK_ENTRY_EX, но позволяет указывать структуру _ATL_FUNC_INFO, которую нужно использовать при вызове обработчика событий. Данный макрос обычно применяется при использовании класса IDispEventSimpleImpl.

Завершается таблица приемников событий макросом END_SINK_MAP. Выглядит эта таблица обычно примерно так.

```
BEGIN_SINK_MAP(CEarPolitic)
    SINK_ENTRY_EX(SOURCE, DIID, DISPID, EventHandlerFunc)
    SINK_ENTRY_INFO(SOURCE, DIID, DISPID, EventHandlerFunc, &info)
...
END_SINK_MAP()
```

В примере с классом CEarPolitic есть три события, поступающие через одинаковые интерфейсы, но от двух разных источников. Соответственно, в таблице приемников событий должно быть шесть записей. Для создания записей, соответствующих источнику Defendant (ответчик), можно использовать макросы SINK_ENTRY_EX. Нам не нужно указывать структуру _ATL_FUNC_INFO, поскольку базовый класс IDispEventImpl использует библиотеку типов, чтобы сформировать эту структуру во время выполнения программы. Для записей, соответствующих источнику Plaintiff (истец), нужно использовать макросы SINK_ENTRY_INFO и предоставлять структуры с информацией о функциях для каждой записи, поскольку для этого источника мы используем класс IDispEventSimpleImpl.

Каждая структура с информацией о функции описывает один метод. Структура содержит информацию о правилах вызова, типе VARIANT возвращаемого значения метода, количестве параметров и типе VARIANT каждого параметра. Следует использовать правила вызова CC_STDCALL, поскольку класс IDispEventSimpleImpl ожидает от обработчиков событий использования именно этих правил.

Вот прототипы функций для трех методов событий Plaintiff и структуры с информацией об этих методах (все эти структуры в данном примере идентичны). Обработчики событий обычно не возвращают значений, т.е. являются void-функциями. Замечание: void-функциям в поле vtReturn структуры _ATL_FUNC_INFO соответствует значение VT_EMPTY, а не VT_VOID.

```
void __stdcall OnHearPlaintiffWhisper(BSTR bstrText);
void __stdcall OnHearPlaintiffTalk(BSTR bstrText);
void __stdcall OnHearPlaintiffYell(BSTR bstrText);

static const int  DISPID_WHISPER = 1 ;
static const int  DISPID_TALK    = 2 ;
static const int  DISPID_YELL    = 3 ;

_ATL_FUNC_INFO OnHearWhisperInfo = {
  CC_STDCALL, VT_EMPTY, 1, { VT_BSTR }};
_ATL_FUNC_INFO OnHearTalkInfo = {
  CC_STDCALL, VT_EMPTY, 1, { VT_BSTR }};
_ATL_FUNC_INFO OnHearYellInfo = {
  CC_STDCALL, VT_EMPTY, 1, { VT_BSTR }};
```

Вот таблица приемников событий для объекта CEarPolitic.

```
BEGIN_SINK_MAP(CEarPolitic)
  SINK_ENTRY_EX(DEFENDANT_SOURCE_ID, __uuidof(_ISpeakerEvents),
    DISPID_WHISPER, OnHearDefendantWhisper)
  SINK_ENTRY_EX(DEFENDANT_SOURCE_ID, __uuidof(_ISpeakerEvents),
    DISPID_TALK, OnHearDefendantTalk)
  SINK_ENTRY_EX(DEFENDANT_SOURCE_ID, __uuidof(_ISpeakerEvents),
    DISPID_YELL, OnHearDefendantYell)
```

```
  SINK_ENTRY_INFO(PLAINTIFF_SOURCE_ID, __uuidof(_ISpeakerEvents),
    DISPID_WHISPER, OnHearPlaintiffWhisper, &OnHearWhisperInfo)
  SINK_ENTRY_INFO(PLAINTIFF_SOURCE_ID, __uuidof(_ISpeakerEvents),
    DISPID_TALK, OnHearPlaintiffTalk, &OnHearTalkInfo)
  SINK_ENTRY_INFO(PLAINTIFF_SOURCE_ID, __uuidof(_ISpeakerEvents),
    DISPID_YELL, OnHearPlaintiffYell, &OnHearYellInfo)
END_SINK_MAP()
```

Предупреждение: в таблице приемников событий содержатся жестко заданные значения DISPID для каждого из событий. А это значит, что таблица приемников событий привязана к определенной версии интерфейса dispinterface. COM разрешает значениям DISPID в интерфейсах dispinterface изменяться от версии к версии. Это случается нечасто, но если случается, то разработчик получает гневные послания от клиентов, работающих с созданными им элементами управления. И тем не менее это *может* случиться.

Единственный способ, позволяющий всегда получать правильные значения DISPID — при запуске запрашивать эти значения у объекта, реализующего интерфейс dispinterface. Для интерфейсов источников это невозможно, поскольку, когда вы создаете приемник, вы реализуете интерфейс, и объекта, у которого можно запросить DISPID, не существует. Так что единственный реалистичный вариант — просматривать библиотеку типов во время выполнения программы. ATL, Visual Basic и MFC этого не делают — все они ради сомнительного выигрыша в производительности подразумевают, что значения DISPID никогда не изменяются.

Методы обратного вызова

Методы обратного вызова, указанные в записях таблицы приемников событий, должны использовать правила вызова __stdcall. Параметры каждого такого метода должны соответствовать по количеству и типам описанию этого метода в библиотеке типов. Вот методы объекта Defendant; они идентичны методам объекта Plaintiff.

```
void __stdcall OnHearDefendantWhisper(BSTR bstrText);
void __stdcall OnHearDefendantTalk(BSTR bstrText);
void __stdcall OnHearDefendantYell(BSTR bstrText);
```

Осталось сделать два завершающих шага: реализовать методы обратного вызова и установить соединение между экземпляром класса CEarPolitic и объектами Demagogue, вызывающими методы _ISpeakerEvents.

Для каждого из обработчиков мы будем использовать следующую простую реализацию.

```
void __stdcall CEarPolitic::OnHearDefendantTalk(BSTR bstrText) {
  CComBSTR title ;
    CreateText(title, OLESTR("defendant"),
    OLESTR("talking"), m_defendant);

    MessageBox(NULL, COLE2CT(bstrText), COLE2CT(title), MB_OK);
}

void CEarPolitic::CreateText(CComBSTR& bstrText,
  LPCOLESTR strRole,
  LPCOLESTR strAction,
  LPUNKNOWN punk) {
    bstrText = m_bstrName;
```

```
    bstrText += OLESTR(" hears the ");
    bstrText += strRole;
    bstrText += OLESTR(" (");

    CComQIPtr<INamedObject> spno = punk;
    CComBSTR bstrName;
    HRESULT hr = spno->get_Name (&bstrName) ;

    bstrText.AppendBSTR(bstrName);
    bstrText += OLESTR(") ");

    bstrText += strAction;
}
```

Соединение источника и приемника событий

Если ваш класс — составной элемент управления, то нужно использовать функцию `AtlAdviseSinkMap` для создания и разрыва соединений между реализациями `IDispEventImpl` и перечисленными в таблице приемников событий интерфейсами источников во внутренних элементах управления. Этот метод использует идентификатор источника событий как идентификатор дочернего окна. С помощью метода `CWindow::GetDlgItem` функция `AtlAdviseSinkMap` переходит к дескриптору дочернего окна, а затем — к интерфейсу `IUnknown` внутреннего элемента управления. От интерфейса `IUnknown` она получает интерфейс `IConnectionPointContainer` и требуемую точку соединения, а затем вызывает метод `Advise`.

```
template <class T>
inline HRESULT AtlAdviseSinkMap(T* pT, bool bAdvise);
```

Функцию `AtlAdviseSinkMap` *обязательно нужно* использовать для установки соединений, если вы используете класс-шаблон `IDispEventImpl` и указываете только два первых параметра в шаблоне, используя для остальных параметров значения по умолчанию. Поскольку интерфейс источника не указан, подразумевается, что в качестве источника событий используется интерфейс источника по умолчанию. Функция `AtlAdviseSinkMap` определяет для каждого источника событий интерфейс источника по умолчанию, если он не определен, и создает соединение с этим интерфейсом.

Если ваш класс не является составным элементом управления, как в нашем примере, нужно явно вызвать метод `DispEventAdvise` каждого базового класса `IdispEventSimpleImpl` (или производного), чтобы соединить каждый источник событий с каждой реализацией приемника событий. Параметр `pUnk` метода `DispEventAdvise` — это интерфейс в источнике событий. Параметр `piid` — это GUID запрашиваемого интерфейса диспетчеризации источника. Метод `DispEventUnadvise` разрывает соединение.

```
template <UINT nID, class T, const IID* pdiid>
class ATL_NO_VTABLE IDispEventSimpleImpl : ... {
...
    HRESULT DispEventAdvise(IUnknown* pUnk, const IID* piid);
    HRESULT DispEventUnadvise(IUnknown* pUnk, const IID* piid);
...
}
```

Вот интерфейс IListener. Мы добавили его в coclass EarPolitic, чтобы дать возможность определять, может ли объект COM слушать объекты Defendant и Plaintiff. Кроме того, он предоставляет методы ListenTo и StopListen, позволяющие устанавливать и разрывать соединение между источником событий Speaker и приемниками событий Defendant и Plaintiff.

```
interface IListener : IDispatch {
  typedef enum SpeakerRole { Defendant, Plaintiff } SpeakerRole;

  [id(1)] HRESULT ListenTo(SpeakerRole role, IUnknown* pSpeaker);
  [id(2)] HRESULT StopListening(SpeakerRole role);
};
```

Реализация этих методов достаточно прямолинейна. Метод ListenTo вызывает метод DispEventAdvise для соответствующего приемника событий, чтобы установить соединение.

```
STDMETHODIMP CEarPolitic::ListenTo(SpeakerRole role,
  IUnknown *pSpeaker) {
  HRESULT hr = StopListening(role) ; // Validates role
  if (FAILED(hr)) return hr ;

  switch (role) {
    case Defendant:
      hr = DefendantEventImpl::DispEventAdvise (pSpeaker,
        &DIID__ISpeakerEvents);
      if (SUCCEEDED(hr))
        m_defendant = pSpeaker;
      else
        Error(OLESTR("The defendant does not support listening"),
          __uuidof(IListener), hr);
    break;
    case Plaintiff:
      hr = PlaintiffEventImpl::DispEventAdvise (pSpeaker,
        &DIID__ISpeakerEvents);
      if (SUCCEEDED(hr))
        m_plaintiff = pSpeaker;
      else
        Error(OLESTR("The Plaintiff does not support listening"),
          __uuidof(IListener), hr);
    break;
  }
  return hr;
}
```

Метод StopListening вызывает метод DispEventUnadvise, чтобы разорвать соединение.

```
STDMETHODIMP CEarPolitic::StopListening(SpeakerRole role) {
  HRESULT hr = S_OK;
  switch (role) {
  case Defendant:
    if (m_defendant)
      hr = DefendantEventImpl::DispEventUnadvise (m_defendant,
        &DIID__ISpeakerEvents);

    if (FAILED(hr))
```

```
      Error (OLESTR("Unexpected error trying to stop listening "
        "to the defendant"), __uuidof(IListener), hr);

    m_defendant = NULL;
    break;

  case Plaintiff:
    if (m_plaintiff)
      hr = PlaintiffEventImpl::DispEventUnadvise(m_plaintiff,
        &DIID__ISpeakerEvents);
    if (FAILED(hr))
      Error(OLESTR("Unexpected error trying to stop listening "
        "to the Plaintiff"), __uuidof(IListener), hr);

    m_plaintiff = NULL;
    break;

  default:
    hr = E_INVALIDARG;
    break;
  }

  return hr;
}
```

Обобщим все сказанное выше. Используйте классы `IDispEventImpl` и `Idisp EventSimpleImpl` для реализации приемников событий для интерфейса диспетчеризации. Вызывайте методы `DispEventAdvise` и `DispEventUnadvise` для установления и разрыва соединений.

Наследуйте свои калссы непосредственно от интерфейсов источников, если источники — простые интерфейсы COM. Вызывайте глобальные функции `AtlAdvise` и `AtlUnadvise` для установления и разрыва соединений. Если вам нужно реализовать одинаковые интерфейсы приемников в нескольких экземплярах, используйте для этого одну из стандартных методик (поэлементную композицию, раскрашивание методов, отдельные классы или промежуточные базовые классы), чтобы избежать конфликтов имен.

Как все это работает: детали реализации

Классы, используемые источником событий

Класс IConnectionPointContainerImpl

Для начала рассмотрим реализацию интерфейса `IConnectionPointContainer` в классе-шаблоне `IConnectionPointContainerImpl`.

Прежде всего класс должен предоставить `vtbl`, совместимую с интерфейсом `IConnectionPointContainer`. Эта `vtbl` должна содержать пять методов: три метода интерфейса `IUnknown` и два — интерфейса `IConnectionPointContainer`.

```
template <class T>
class ATL_NO_VTABLE IConnectionPointContainerImpl
  : public IconnectionPointContainer {
  typedef CComEnum<IEnumConnectionPoints,
    &__uuidof(IEnumConnectionPoints), IConnectionPoint*,
```

```
    _CopyInterface<IConnectionPoint> >
    CComEnumConnectionPoints;
public:
  STDMETHOD(EnumConnectionPoints)(
    IEnumConnectionPoints** ppEnum) {
    if (ppEnum == NULL) return E_POINTER;

    *ppEnum = NULL;
    CComEnumConnectionPoints* pEnum = NULL;
    ATLTRY(pEnum = new CComObject<CComEnumConnectionPoints>)
    if (pEnum == NULL) return E_OUTOFMEMORY;

    int nCPCount;
    const _ATL_CONNMAP_ENTRY* pEntry = T::GetConnMap(&nCPCount);

    // выделяем и инициализируем вектор указателей на
    // объекты точек соединения
    USES_ATL_SAFE_ALLOCA;
    if ((nCPCount < 0) || (nCPCount >
      (INT_MAX / sizeof(IConnectionPoint*))))
      return E_OUTOFMEMORY;
    size_t nBytes=0;
    HRESULT hr=S_OK;
    if( FAILED(hr=::ATL::AtlMultiply(&nBytes,
      sizeof(IConnectionPoint*),
      static_cast<size_t>(nCPCount)))) {
        return hr;
    }
    IConnectionPoint** ppCP =
      (IConnectionPoint**)_ATL_SAFE_ALLOCA(
      nBytes, _ATL_SAFE_ALLOCA_DEF_THRESHOLD);
    if (ppCP == NULL) {
      delete pEnum;
      return E_OUTOFMEMORY;
    }

    int i = 0;
    while (pEntry->dwOffset != (DWORD_PTR)-1) {
      if (pEntry->dwOffset == (DWORD_PTR)-2) {
        pEntry++;
        const _ATL_CONNMAP_ENTRY* (*pFunc)(int*) =
          (const _ATL_CONNMAP_ENTRY* (*)(int*))(
          pEntry->dwOffset);
        pEntry = pFunc(NULL);
        continue;
      }
      ppCP[i++] = (IConnectionPoint*)(
        (INT_PTR)this+pEntry->dwOffset);
      pEntry++;
    }

    // Копируем указатели: они будут вызывать AddRef для объекта
    HRESULT hRes = pEnum->Init((IConnectionPoint**)&ppCP[0],
      (IConnectionPoint**)&ppCP[nCPCount],
      reinterpret_cast<IConnectionPointContainer*>(this),
      AtlFlagCopy);
    if (FAILED(hRes)) {
      delete pEnum;
      return hRes;
```

```
    }
    hRes = pEnum->QueryInterface(
      __uuidof(IEnumConnectionPoints),
      (void**)ppEnum);
    if (FAILED(hRes)) delete pEnum;
    return hRes;
  }

  STDMETHOD(FindConnectionPoint)(REFIID riid,
    IConnectionPoint** ppCP) {
    if (ppCP == NULL) return E_POINTER;
    *ppCP = NULL;
    HRESULT hRes = CONNECT_E_NOCONNECTION;
    const _ATL_CONNMAP_ENTRY* pEntry = T::GetConnMap(NULL);
    IID iid;
    while (pEntry->dwOffset != (DWORD_PTR)-1) {
      if (pEntry->dwOffset == (DWORD_PTR)-2) {
        pEntry++;
        const _ATL_CONNMAP_ENTRY* (*pFunc)(int*) =
          (const _ATL_CONNMAP_ENTRY* (*)(int*))(
            pEntry->dwOffset);
        pEntry = pFunc(NULL);
        continue;
      }
      IConnectionPoint* pCP =
        (IConnectionPoint*)((INT_PTR)this+pEntry->dwOffset);
      if (SUCCEEDED(pCP->GetConnectionInterface(&iid)) &&
        InlineIsEqualGUID(riid, iid)) {
        *ppCP = pCP;
        pCP->AddRef();
        hRes = S_OK;
        break;
      }
      pEntry++;
    }
    return hRes;
  }
};
```

Методы `IUnknown` просты: класс их не реализует. Реализацию этих методов мы получаем, определяя класс `CComObject`, параметризованный классом соединяемого объекта, например `CComObject<CConnectableObject>`.

Определение типа `CComEnumConnectionPoints` объявляет класс стандартного перечислителя COM, реализующий интерфейс `IEnumConnectionPoints`. Этот класс используется для перечисления интерфейсных указателей `IConnectionPoint`. Шаблонное расширение класса ATL `CComEnum` предоставляет реализацию. Реализация метода `EnumConnectionPoints` создает и возвращает экземпляр данного перечислителя.

Метод `EnumConnectionPoints` начинается со стандартных операций проверки, а затем создает в куче новый экземпляр перечислителя `CComEnumConnectionPoints`. Реализация перечислителей в ATL требует их инициализации после создания. Перечислители в ATL не слишком гибки: чтобы их инициализировать, нужно передавать им массивы элементов, которые должны перечисляться. В нашем случае перечислитель предоставляет указатели `IConnectionPoint`, поэтому массив, используемый для инициализации, должен состоять из указателей `IConnectionPoint`.

Карта соединений соединяемого объекта содержит информацию, необходимую для создания массива указателей IConnectionPoint. Каждая запись в этой карте содержит смещение соединяемого объекта от начала экземпляра IConnection PointContainerImpl (т.е. текущего значения указателя this) до начала экземпляра IConnectionPointImpl.

Метод EnumConnectionPoints выделяет память под инициализационный массив в стеке, используя макрос _ATL_SAFE_ALLOCA. Он просматривает все записи в карте соединений, вычисляет значение интерфейсного указателя IConnectionPoint для каждого объекта IConnectionPointImpl и сохраняет это значение в массиве. Обратите внимание, что для этих указателей не ведется подсчет ссылок, поскольку время жизни указателей в массиве, расположенном в стеке, ограничено временем выполнения метода.

Экземпляр перечислителя инициализируется вызовом метода перечислителя Init. Здесь важно использовать аргумент AtlFlagCopy. Это указывает перечислителю сделать копию элементов из инициализационного массива. Для интерфейсных указателей это значит вызвать AddRef при создании копии. Обратитесь к главе 8, "Коллекции и перечислители", за подробной информацией об инициализации перечислителей COM.

Указатель pEnum — это указатель на объект CComEnumConnectionPoints, хотя было бы лучше, если бы он был объявлен как указатель CComObject<CComEnum ConnectionPoints>, поскольку в действительности он является именно им. Но в любом случае метод EnumConnectionPoints должен вернуть указатель Ienum ConnectionPoints, а не сам pEnum, поэтому он запрашивает у перечислителя (через pEnum) соответствующий интерфейсный указатель и возвращает полученный указатель.

Метод FindConnectionPoint весьма прямолинеен. После обычных проверок он с помощью карты соединений вычисляет значения интерфейсных указателей IConnectionPoint для каждой точки соединения в соединяемом объекте. Используя интерфейсные указатели, метод запрашивает у каждой точки соединения IID поддерживаемого интерфейса и сравнивает это IID с тем, которое пытается найти. При обнаружении требуемого IID метод возвращает интерфейсный указатель, вызвав для него метод AddRef, с кодом состояния S_OK; если требуемый IID не найден, метод возвращает код состояния CONNECT_E_NOCONNECTION.

Довольно интересный момент в этом коде — проверка смещения на равенство -2 при просмотре карты соединений. Это значение появляется только при использовании ATL с атрибутами (она рассматривается в приложении Г, "Атрибуты в ATL"). ATL с атрибутами вставляет в класс вторую карту точек соединения. Проверка значения на равенство -2 позволяет определить, когда нужно переходить от стандартной карты точек соединения к карте с атрибутами. Для этого вызывается функция, указатель на которую хранится в соответствующей ячейке. Она позволяет перейти к начальному адресу новой карты, которую нужно просмотреть. Теоретически на основании этого фокуса можно реализовать сцепление карт точек соединения; однако макросы карты точек соединения не поддерживают сцепление, так что для его реализации придется повозиться.

Большая часть черновой работы достается реализации точек соединения, так что давайте взглянем на нее.

Класс IConnectionPointImpl

Класс-шаблон IConnectionPointImpl предоставляет реализацию интерфейса IConnectionPoint. Чтобы сделать это, класс должен предоставить vtbl, совместимую с интерфейсом IConnectionPoint. В этой vtbl должно быть восемь методов: три из интерфейса IUnknown и пять — из IConnectionPoint.

Первый существенный момент — то, что класс IConnectionPointImpl является производным от _ICPLocator.

```
template <class T, const IID* piid,
  class CDV = CComDynamicUnkArray >
class ATL_NO_VTABLE IConnectionPointImpl
  : public _ICPLocator<piid>
{ ... }
```

Класс _ICPLocator

```
template <const IID* piid>
class ATL_NO_VTABLE _ICPLocator {
public:
    // этому методу требуется имя, отличное от QueryInterface
    STDMETHOD(_LocCPQueryInterface)(REFIID riid,
        void ** ppvObject) = 0;
    virtual ULONG STDMETHODCALLTYPE AddRef(void) = 0;
    virtual ULONG STDMETHODCALLTYPE Release(void) = 0;
};
```

Класс _ICPLocator содержит объявление виртуального метода _LocCPQuery Interface. Виртуальный метод занимает одну позицию в vtbl, поэтому данное объявление означает, что вызовы через первую позицию в vtbl, которая используется для обращений к QueryInterface, будут обрабатываться методом _LocCP QueryInterface. Этот метод объявлен как чисто виртуальный, поэтому его реализацию должны предоставлять производные классы. Это важно, поскольку каждая точка соединения должна предоставлять уникальную реализацию _LocCPQuery Interface.

Точка соединения должна быть сущностью COM, отдельной от контейнера точек соединения. Соответственно, точке соединения требуется своя реализация метода QueryInterface. Если бы мы назвали первый виртуальный метод в _ICPLocator QueryInterface, то согласно правилам множественного наследования C++ посчитал бы этот метод просто еще одной ссылкой на QueryInterface для соединяемого объекта. Обычно это именно то, что нам требуется. Например, представьте себе, что у нас есть класс, производный от трех интерфейсов. Все три интерфейса упоминают виртуальный метод QueryInterface, но нам нужна единая реализация, общая для всех базовых классов. Точно так же нам нужны общие реализации методов AddRef и Release. Но для точки соединения в базовом классе этого не нужно.

Идея заключается в том, что нам нужны две отдельные сущности COM (соединяемый объект и точка соединения) и соответственно две отдельные реализации QueryInterface. Но остальные элементы реализации IUnknown (AddRef и Release) мы можем объединить, поскольку вести отдельный подсчет ссылок для каждой точки соединения нам не нужно. ATL использует такой подход, чтобы не делегировать вызовы AddRef и Release из точек соединения в соединяемый объект.

Методы IConnectionPointImpl

Методы интерфейса IUnknown в IConnectionPointImpl сложнее, чем в IConnectionPointContainerImpl, поскольку точка соединения реализует собственный метод QueryInterface.

```
template <class T, const IID* piid,
  class CDV = CComDynamicUnkArray >
class ATL_NO_VTABLE IConnectionPointImpl
  : public _ICPLocator<piid> {
typedef CComEnum<IEnumConnections,
    &__uuidof(IEnumConnections), CONNECTDATA,
    _Copy<CONNECTDATA> > CComEnumConnections;
  typedef CDV _CDV;
public:
  ~IConnectionPointImpl();
  STDMETHOD(_LocCPQueryInterface)(REFIID riid, void ** ppvObject) {
#ifndef _ATL_OLEDB_CONFORMANCE_TESTS
  ATLASSERT(ppvObject != NULL);
#endif
  if (ppvObject == NULL)
    return E_POINTER;
  *ppvObject = NULL;

  if (InlineIsEqualGUID(riid, __uuidof(IConnectionPoint)) ||
    InlineIsEqualUnknown(riid)) {
    *ppvObject = this;
    AddRef();
#ifdef _ATL_DEBUG_INTERFACES
      _AtlDebugInterfacesModule.AddThunk((IUnknown**)ppvObject,
      _T("IConnectionPointImpl"), riid);
#endif // _ATL_DEBUG_INTERFACES
      return S_OK;
    }
    else
      return E_NOINTERFACE;
  }

  STDMETHOD(GetConnectionInterface)(IID* piid2) {
    if (piid2 == NULL)
      return E_POINTER;
    *piid2 = *piid;
    return S_OK;
  }
  STDMETHOD(GetConnectionPointContainer)(
    IConnectionPointContainer** ppCPC) {
    T* pT = static_cast<T*>(this);
    // Незачем проверять ppCPC на равенство NULL, поскольку
    // QI все равно сделает это за нас
    return pT->QueryInterface(
      __uuidof(IConnectionPointContainer), (void**)ppCPC);
  }

  STDMETHOD(Advise)(IUnknown* pUnkSink, DWORD* pdwCookie);
  STDMETHOD(Unadvise)(DWORD dwCookie);
  STDMETHOD(EnumConnections)(IEnumConnections** ppEnum);
  CDV m_vec;
};
```

Метод _LocCPQueryInterface

Сигнатура метода `_LocCPQueryInterface` такая же, как у метода `QueryInterface`, но он отвечает только на запросы `IID_Unknown` и `IID_IConnectionPoint`, предоставляя указатель на себя, для которого вызывает метод `AddRef`. Это делает каждый экземпляр базового класса объекта `IConnectionPointImpl` с помощью отдельной сущности COM.

Методы AddRef и Release

Как обычно, реализацию этих методов мы получаем, определяя класс `CComObject`, параметризованный классом соединяемого объекта, например `CComObject<CComConnectableObject>`.

Методы GetConnectionInterface и GetConnectionPointContainer

Метод `GetConnectionInterface` просто возвращает IID интерфейса источника для точки соединения, поэтому его реализация тривиальна. Метод `GetConnectionPointContainer` тоже прост, но требует приведения типов для запроса правильного интерфейсного указателя.

Проблема в том, что данное конкретное расширение `IConnectionPointImpl` не поддерживает интерфейс `IConnectionPointContainer`. Но структура классов-шаблонов требует задания класса соединяемого объекта в параметр `T` для реализации интерфейса `IConnectionPointContainer`.

```
T* pT = static_cast<T*>(this);
return pT->QueryInterface(IID_IConnectionPointContainer,
  (void**)ppCPC);
```

Приведение типа переходит от подобъекта точки соединения вниз по иерархии классов к производному классу соединяемого объекта и вызывает метод этого класса `QueryInterface`, чтобы получить требуемый интерфейсный указатель `IConnectionPointContainer`.

Методы Advise, Unadvise и EnumConnections

Все эти методы работают со списком активных соединений. Метод `Advise` добавляет в список новые пункты, `Unadvise` — удаляет их, а `EnumConnections` возвращает объект, представляющий собой перечислитель списка.

Тип этого списка определяется параметром шаблона `CDV`. По умолчанию это тип `CComDynamicUnkArray`, предоставляющий для реализации списка массив динамически изменяющегося размера. Как я уже упоминал ранее, ATL предоставляет реализацию списка фиксированного размера и специальную реализацию для списка из одного элемента. Однако относительно несложно предоставить нестандартную реализацию списка, поскольку методы `Advise`, `Unadvise` и `EnumConnections` обращаются к этому списку через хорошо известные методы: `Add`, `Remove`, `begin`, `end`, `GetCookie` и `GetUnknown`.

Определение типа `CComEnumConnections` определяет класс для стандартного перечислителя COM, реализующего интерфейс `IEnumConnections`. Этот класс перечислителя используется для перечисления структур `CONNECTDATA`, содержащих указатель интерфейса преемника в клиенте и связанное с ним cookie-значение для регистрации. Расширение шаблона ATL `CComEnum` предоставляет реализацию. Реализация метода `EnumConnections` создает и возвращает экземпляр этого перечислителя.

Метод Advise

```
template <class T, const IID* piid, class CDV>
STDMETHODIMP IConnectionPointImpl<T, piid, CDV>::Advise(
    IUnknown* pUnkSink,
    DWORD* pdwCookie) {
    T* pT = static_cast<T*>(this);
    IUnknown* p;
    HRESULT hRes = S_OK;
    if (pdwCookie != NULL)
        *pdwCookie = 0;
    if (pUnkSink == NULL || pdwCookie == NULL)
        return E_POINTER;
    IID iid;
    GetConnectionInterface(&iid);
    hRes = pUnkSink->QueryInterface(iid, (void**)&p);
    if (SUCCEEDED(hRes)) {
        pT->Lock();
        *pdwCookie = m_vec.Add(p);
        hRes = (*pdwCookie != NULL) ? S_OK :
            CONNECT_E_ADVISELIMIT;
        pT->Unlock();
        if (hRes != S_OK)
            p->Release();
    }
    else if (hRes == E_NOINTERFACE)
        hRes = CONNECT_E_CANNOTCONNECT;
    if (FAILED(hRes))
        *pdwCookie = 0;
    return hRes;
}
```

Метод `Advise` получает IID интерфейса приемника для данной точки соединения и запрашивает интерфейсный указатель `IUnknown`, предоставляемый клиентом для интерфейса приемника. Это гарантирует, что клиент передает интерфейсный указатель объекта, который действительно реализует интерфейс приемника. Если передан некорректный указатель, метод вернет значение `CONNECT_E_CANNOTCONNECT`. Вам не захочется поддерживать соединение с чем-то, не способным принимать обратные вызовы. Кроме того, убедившись, что интерфейсный указатель корректен, точка соединения избавится от необходимости запрашивать его при каждом обратном вызове.

Если запрос выполняется успешно, точка соединения должна добавить соединение в список. Поэтому она получает блокировку на весь соединяемый объект, добавляет соединение в список, если в списке есть место, а затем снимает блокировку.

Метод Unadvise

```
template <class T, const IID* piid, class CDV>
STDMETHODIMP IConnectionPointImpl<T, piid, CDV>::Unadvise(
    DWORD dwCookie) {
    T* pT = static_cast<T*>(this);
    pT->Lock();
    IUnknown* p = m_vec.GetUnknown(dwCookie);
    HRESULT hRes = m_vec.Remove(dwCookie) ? S_OK :
        CONNECT_E_NOCONNECTION;
    pT->Unlock();
    if (hRes == S_OK && p != NULL)
```

```
        p->Release();
    return hRes;
}
```

Этот метод относительно прост. Он блокирует соединяемый объект, запрашивает у класса списка указатель IUnknown, соответствующий cookie-значению, удаляет соединение, которому соответствует это cookie-значение, снимает блокировку с соединяемого объекта и освобождает интерфейсный указатель приемника.

Метод EnumConnections

Метод EnumConnections начинается с нескольких проверок, а затем создает в куче новый экземпляр класса перечислителя CComObject<CComEnumConnections>. Как и раньше, реализация перечислителя в ATL требует, чтобы перечислитель после создания инициализировался массивом — в данном случае непрерывным массивом структур CONNECTDATA.

```
template <class T, const IID* piid, class CDV>
STDMETHODIMP IConnectionPointImpl<T, piid, CDV>::EnumConnections(
    IEnumConnections** ppEnum) {
    if (ppEnum == NULL)
        return E_POINTER;
    *ppEnum = NULL;
    CComObject<CComEnumConnections>* pEnum = NULL;
    ATLTRY(pEnum = new CComObject<CComEnumConnections>)
    if (pEnum == NULL)
        return E_OUTOFMEMORY;
    T* pT = static_cast<T*>(this);
    pT->Lock();
    CONNECTDATA* pcd = NULL;
    ATLTRY(pcd = new CONNECTDATA[m_vec.end()-m_vec.begin()])
    if (pcd == NULL) {
        delete pEnum;
        pT->Unlock();
        return E_OUTOFMEMORY;
    }
    CONNECTDATA* pend = pcd;
    // Копируем корректные структуры CONNECTDATA
    for (IUnknown** pp = m_vec.begin();pp<m_vec.end();pp++) {
        if (*pp != NULL)
        {
            (*pp)->AddRef();
            pend->pUnk = *pp;
            pend->dwCookie = m_vec.GetCookie(pp);
            pend++;
        }
    }
    // не копируем данные, просто принимаем право владения ими
    pEnum->Init(pcd, pend, NULL, AtlFlagTakeOwnership);
    pT->Unlock();
    HRESULT hRes = pEnum->_InternalQueryInterface(
        __uuidof(IEnumConnections), (void**)ppEnum);
    if (FAILED(hRes))
        delete pEnum;
    return hRes;
}
```

В списке соединений хранятся указатели IUnknown. Структура CONNECTDATA содержит интерфейсный указатель и связанное с ним cookie-значение. Метод Enum Connections выделяет память под инициализационный массив из кучи. Он перебирает элементы списка соединений, копируя интерфейсные указатели и cookie-значения в динамически выделяемый массив структур CONNECTDATA. Важно заметить, что для любых интерфейсных указателей, не равных NULL, вызывается метод AddRef. Эта копия интерфейсных указателей существует дольше времени выполнения метода EnumConnections.

Экземпляр перечисления инициализируется вызовом метода Init. Здесь важно использовать аргумент AtlFlagTakeOwnership. Этот аргумент указывает перечислителю непосредственно использовать заданный массив, а не создавать его копию. Это также значит, что перечислитель отвечает за корректное освобождение элементов массива и самого массива.

Метод EnumConnections использует _InternalQueryInterface для возвращения интерфейсного указателя IEnumConnections объекта-перечислителя, который в данный момент является единственной внешней ссылкой на перечислитель[2].

Классы, используемые приемником событий

Прежде всего нужно уяснить себе общее положение дел с приемниками. Ваш объект может реализовывать несколько разных интерфейсов приемников событий или реализовывать один и тот же интерфейс несколько раз. Все интерфейсы приемников событий требуют практически идентичной функциональности: IUnknown, IDispatch, Invoke и способности искать DISPID в таблице приемников и делегировать событие соответствующему обработчику. Но каждой реализации также требуется определенная специфическая функциональность, а точнее, каждая реализация должна быть самостоятельной сущностью в COM.

В ATL определен класс _IDispEvent, реализующий стандартную функциональность и позволяющий каждому производному классу быть отдельной сущностью в COM за счет использования раскрашивания интерфейсов и параметров шаблона. Это значит, что ATL реализует все специализированные реализации приемников событий с помощью одного класса C++ — _IDispEvent.

Класс _IDispEvent

Давайте рассмотрим класс _IDispEvent. Первый интересный момент в нем — то, что он предназначен для использования в качестве абстрактного базового класса. Первые три виртуальных метода объявлены согласно стандартным правилам вызова COM и являются чисто виртуальными. Первый метод — это _LocDEQueryInterface, а следующие два — AddRef и Release. В результате vtbl класса _IDispEvent обеспечивает поддержку интерфейса IUnknown. Класс _IDispEvent просто не может быть производным от IUnknown, поскольку он должен предоставлять специализированную версию QueryInterface. Производный класс должен предоставлять методы _LocDEQueryInterface, AddRef и Release.

---

[2] Метод EnumConnections использует _InternalQueryInterface вместо IUnknown::QueryInterface, поскольку первая функция вызывается напрямую, а вторая (менее эффективно) как виртуальный метод.

```
class ATL_NO_VTABLE _IDispEvent {
...
public:
    // этому методу требуется имя, отличное от QueryInterface
    STDMETHOD(_LocDEQueryInterface)(REFIID riid,
        void ** ppvObject) = 0;
    virtual ULONG STDMETHODCALLTYPE AddRef(void) = 0;
    virtual ULONG STDMETHODCALLTYPE Release(void) = 0;
...
};
```

В классе есть пять переменных-членов, только одна из которых используется всегда — это переменная m_dwEventCookie. Все переменные-члены инициализируются конструктором.

```
_IDispEvent() :
  m_libid(GUID_NULL),
  m_iid(IID_NULL),
  m_wMajorVerNum(0),
  m_wMinorVerNum(0),
  m_dwEventCookie(0xFEFEFEFE)
{ }
```

В переменной m_dwEventCookie хранится регистрационное значение точки соединения, возвращаемое из метода IConnectionPoint::Advise объекта-источника. Это значение хранится до тех пор, пока оно не понадобится для разрыва соединения. Класс подразумевает, что никакой источник событий не будет использовать значение 0xFEFEFEFE, поскольку оно обозначает отсутствие соединения.[3]

В переменных m_libid, m_iid, m_wMajorVerNum и m_wMinorVerNum хранятся соответственно GUID библиотеки типов, IID интерфейса источника и номера версии и подверсии.

```
GUID m_libid;
IID m_iid;
unsigned short m_wMajorVerNum;
unsigned short m_wMinorVerNum;
DWORD m_dwEventCookie;
```

Класс _IDispEvent предоставляет методы DispEventAdvise и DispEvent Unadvise, которые соответственно устанавливают и разрывают соединение между интерфейсом источника piid объекта источника pUnk и объектом-приемником _DispEvent.

```
HRESULT DispEventAdvise(IUnknown* pUnk, const IID* piid) {
    ATLENSURE(m_dwEventCookie == 0xFEFEFEFE);
    return AtlAdvise(pUnk, (IUnknown*)this, *piid, &m_dwEventCookie);
```

[3] Заметьте, что такой подход накладывает ограничение на реализацию списка соединений (т.е. класс-параметр шаблона CDV) — в этом списке никогда не должно использоваться cookie-значение 0xFEFEFEFE.

Разумеется, класс _IDispEvent мог бы использовать специальный флаг "соединение установлено", но этот флаг увеличил бы размер объектов класса. Однако похоже, что разработчики класса _IDispEvent не ставили себе цели минимизировать размер объектов, поскольку в этом классе есть четыре переменных-члена, используемых далеко не всегда. Обычно в ATL переменные, которые не всегда используются, выделяются в отдельный производный класс.

```
}

HRESULT DispEventUnadvise(IUnknown* pUnk, const IID* piid) {
    HRESULT hr = AtlUnadvise(pUnk, *piid, m_dwEventCookie);
    m_dwEventCookie = 0xFEFEFEFE;
    return hr;
}
```

Можно реализовать несколько приемников событий в одном объекте ATL COM. В самом общем случае это значит, что вам потребуется отдельный приемник событий для каждого источника событий (идентификатора источника). Кроме того, вам потребуется отдельный приемник событий для каждого отдельного соединения с источником событий (интерфейса источника).

Класс _IDispEventLocator

Реализация нескольких приемников событий требует непрямого многократного наследования приемника от `_IDispEvent`. Но нам нужно выполнить наследование так, чтобы можно было найти требуемый экземпляр базового класса `_IDispEvent` по идентификатору объекта-источника и идентификатору интерфейса источника в этом объекте.

В ATL для этого используется класс-шаблон `_IDispEventLocator`. Каждое обращение к шаблону `_IDispEventLocator` генерирует отдельный, доступный для обращений экземпляр приемника событий `_IDispEvent`.[4]

```
template <UINT nID, const IID* piid>
class ATL_NO_VTABLE _IDispEventLocator : public _IDispEvent {
public:
};
```

Класс IDispEventSimpleImpl

Этот класс реализует интерфейс `IDispatch`. Он наследует класс `_IDispEvent Locator<nID, pdiid>`, чтобы получить `vtbl`, поддерживающую интерфейс `IUnknown`, переменные-члены и методы `Advise` и `Unadvise` для работы с точками соединений, предоставляемые базовым классом `_IDispEvent`.

```
template <UINT nID, class T, const IID* pdiid>
class ATL_NO_VTABLE IDispEventSimpleImpl :
    public _IDispEventLocator<nID, pdiid> {
// Сокращен для компактности
    STDMETHOD(_LocDEQueryInterface)(REFIID riid,
        void ** ppvObject);
    virtual ULONG STDMETHODCALLTYPE AddRef();
    virtual ULONG STDMETHODCALLTYPE Release();
    STDMETHOD(GetTypeInfoCount)(UINT* pctinfo);
    STDMETHOD(GetTypeInfo)(UINT itinfo, LCID lcid,
```

[4] Кейт Браун (Keith Brown) считает, что было бы лучше называть эти классы не `Locator`, а `COMIdentity`, например `_IDispEventComIdentity` и `IConnectionPointCOMIdentity`, поскольку они должны только предоставлять отдельные экземпляры базовых классов для каждой используемой сущности COM. Да, возникает необходимость поиска требуемого объекта базового класса, но единственная цель переименования метода `QueryInterface` — создание отдельных сущностей.

```
        ITypeInfo** pptinfo);
    STDMETHOD(GetIDsOfNames)(REFIID riid, LPOLESTR* rgszNames,
        UINT cNames, LCID lcid, DISPID* rgdispid);
    STDMETHOD(Invoke)(DISPID dispidMember, REFIID riid,
        LCID lcid, WORD wFlags, DISPPARAMS* pdispparams,
    VARIANT* pvarResult, EXCEPINFO* pexcepinfo,
        UINT* puArgErr);
};
```

Заметьте, что класс `IDispEventSimpleImpl` предоставляет реализацию методов `_LocDEQueryInterface`, `AddRef` и `Release`, унаследованных от базового класса `_IDispEvent`. Кроме того, заметьте, что следующие четыре виртуальных метода — это стандартные методы интерфейса `IDispatch`. Поэтому класс `IDisp EventSimpleImpl` предоставляет `vtbl`, поддерживающую `IDispatch`. Он не может просто наследовать `IDispatch`, чтобы получить такую `vtbl`, поскольку должен предоставлять специализированную версию `QueryInterface`.

Класс `IDispEventSimpleImpl` реализует метод `_LocDEQueryInterface`, чтобы каждый приемник событий был отдельной от производного класса сущностью COM. Объект-приемник событий должен положительно отвечать на запросы к ID интерфейса диспетчеризации источников, интерфейсу `IUnknown`, интерфейсу `IDispatch` и GUID, содержащемуся в переменной `m_iid`.

```
STDMETHOD(_LocDEQueryInterface)(REFIID riid,
    void ** ppvObject) {
    ATLASSERT(ppvObject != NULL);
    if (ppvObject == NULL)
        return E_POINTER;
    *ppvObject = NULL;

    if (InlineIsEqualGUID(riid, IID_NULL))
        return E_NOINTERFACE;

    if (InlineIsEqualGUID(riid, *pdiid) ||
        InlineIsEqualUnknown(riid) ||
        InlineIsEqualGUID(riid, __uuidof(IDispatch)) ||
        InlineIsEqualGUID(riid, m_iid)) {

        *ppvObject = this;
        AddRef();
#ifdef _ATL_DEBUG_INTERFACES
        _AtlDebugInterfacesModule.AddThunk(
            (IUnknown**)ppvObject, _T("IDispEventImpl"),
            riid);
#endif // _ATL_DEBUG_INTERFACES
            return S_OK;
        }
        else
            return E_NOINTERFACE;
    }
```

Класс `IDispEventSimpleImpl` также предоставляет простую реализацию методов `AddRef` и `Release`. Это позволяет использовать данный класс непосредственно как объект COM.

```
template <UINT nID, class T, const IID* pdiid>
class ATL_NO_VTABLE IDispEventSimpleImpl : ... {
...
    virtual ULONG STDMETHODCALLTYPE AddRef() { return 1; }
    virtual ULONG STDMETHODCALLTYPE Release() { return 1; }
...
};
```

Однако при использовании этого класса в более сложных объектах COM используются методы AddRef и Release производных классов. Другими словами, при вызове AddRef для приемника событий типичного объекта ATL COM выполняется метод CComObject::AddRef (или метод другого класса, последнего в иерархии наследования). Будьте внимательны. Учитывайте появляющуюся при этом цикличность в подсчете ссылок. Клиент хранит ссылку на источник событий, который, в свою очередь, хранит ссылку на приемник событий, а на самом деле — на самого клиента.

Класс IDispEventSimpleImpl реализует методы GetTypeInfoCount, GetTypeInfo и GetIDsOfNames, просто возвращая из них значение E_NOTIMPL. Интерфейс диспетчеризации событий используется только для поддержки методов интерфейса IUnknown и метода Invoke.

```
STDMETHOD(GetTypeInfoCount)(UINT*)
{ATLTRACENOTIMPL(_T("IDispEventSimpleImpl::GetTypeInfoCount"));}

STDMETHOD(GetTypeInfo)(UINT, LCID, ITypeInfo**)
{ATLTRACENOTIMPL(_T("IDispEventSimpleImpl::GetTypeInfo"));}

STDMETHOD(GetIDsOfNames)(REFIID, LPOLESTR*, UINT, LCID, DISPID*)
{ATLTRACENOTIMPL(_T("IDispEventSimpleImpl::GetIDsOfNames"));}

STDMETHOD(Invoke)(DISPID dispidMember, REFIID, LCID lcid, WORD
/*wFlags*/,
                  DISPPARAMS* pdispparams, VARIANT* pvarResult,
                  EXCEPINFO* /*pexcepinfo*/, UINT* /*puArgErr*/);
```

Метод Invoke ищет в таблице приемников событий производного класса обработчик, подходящий для текущего события. Он находит требуемую таблицу приемников, вызывая _GetSinkMap — статический метод, определенный в наследующем классе с помощью макроса BEGIN_SINK_MAP (мы рассмотрим этот макрос ниже в данном разделе). Подходящему обработчику событий соответствует запись с теми же ID источника событий (nID) и IID интерфейса источника событий (pdiid), что и в вызове шаблона, и с тем же DISPID, что и у аргумента Invoke.

Если соответствующая запись в таблице приемников указывает структуру _ATL_FUNC_INFO (т.е. запись была определена с помощью макроса SINK_ENTRY_INFO), метод Invoke использует эту структуру для вызова обработчика. В противном случае Invoke вызывает виртуальную функцию GetFuncInfoFromID для получения требуемой структуры. Если функция GetFuncInfoFromID не дает требуемой структуры, Invoke молча возвращает значение S_OK. Это необходимо, поскольку обработчик события должен отвечать значением S_OK на события, которые он не распознает.

При использовании макроса SINK_ENTRY_EX с классом IDispEventSimpleImpl нужно переопределять метод GetFuncInfoFromID. Реализация этого метода по умолчанию просто генерирует ошибку.

```
virtual HRESULT GetFuncInfoFromId(const IID&, DISPID, LCID,
  _ATL_FUNC_INFO&) {
  ATLTRACE(_T("TODO: Classes using IDispEventSimpleImpl should "
    "override this method\n"));
  ATLASSERT(0);
  ATLTRACENOTIMPL(_T("IDispEventSimpleImpl::GetFuncInfoFromId"));
}
```

Это значит, что если вы воспользуетесь классом IDispEventSimpleImpl, зададите обработчик события с помощью макроса SINK_ENTRY_EX и забудете переопределить метод GetFuncInfoFromId или переопределите его неправильно, то все будет замечательно компилироваться, но ваш обработчик событий *никогда не будет вызываться*.

Класс IDispEventSimpleImpl предоставляет несколько вспомогательных перегруженных методов для установки и разрыва соединений с источниками событий. Два приведенных ниже метода устанавливают и разрывают соединение между текущим приемником и указанным интерфейсом событий (piid) указанного источника событий (pUnk).

```
// Вспомогательные методы для работы с IUnknown*
    HRESULT DispEventAdvise(IUnknown* pUnk, const IID* piid) {
        ATLENSURE(m_dwEventCookie == 0xFEFEFEFE);
        return AtlAdvise(pUnk, (IUnknown*)this, *piid,
&m_dwEventCookie);
    }

    HRESULT DispEventUnadvise(IUnknown* pUnk, const IID* piid) {
        HRESULT hr = AtlUnadvise(pUnk, *piid, m_dwEventCookie);
        m_dwEventCookie = 0xFEFEFEFE;
        return hr;
    }
```

Следующие два метода устанавливают и разрывают соединение между текущим приемником событий и указанным источником, используя интерфейс диспетчеризации приемника.

```
HRESULT DispEventAdvise(IUnknown* pUnk) {
  return _IDispEvent::DispEventAdvise(pUnk, pdiid);
}

HRESULT DispEventUnadvise(IUnknown* pUnk) {
  return _IDispEvent::DispEventUnadvise(pUnk, pdiid);
}
```

Таблица приемников: структуры, макросы и метод _GetSinkMap

Таблица приемников — это массив структур _ATL_EVENT_ENTRY. В этих структурах есть такие поля.

- nControlID — идентификатор источника событий; ID элемента управления для внутренних элементов управления.

- piid — IID интерфейса диспетчеризации источника.
- nOffset — смещение реализации приемника событий от производного класса.
- dispid — ID диспетчеризации обратных вызовов для событий.
- pft — указатель на метод-обработчик событий, который нужно вызвать.[5]
- pInfo — структура _ATL_FUNC_INFO, используемая для вызовов обработчика.

```
template <class T>
struct _ATL_EVENT_ENTRY {
    UINT nControlID; // ID, обозначающий экземпляр
    const IID* piid; // IID интерфейса dispinterface
    int nOffset;     // смещение dispinterface от указателя this
    DISPID dispid;   // DISPID метода/свойства
    void (__stdcall T::*pfn)(); // метод, который нужно вызвать
    _ATL_FUNC_INFO* pInfo; // указатель на структуру
                           // с информацией об обработчике
};
```

Макрос BEGIN_SINK_MAP при использовании определяет в вашем классе статический метод _GetSinkMap. Этот метод возвращает адрес массива структур _ATL_EVENT_ENTRY.

```
#define BEGIN_SINK_MAP(_class)\
    typedef _class _GetSinkMapFinder;\
    static const ATL::_ATL_EVENT_ENTRY<_class>* _GetSinkMap() {\
        PTM_WARNING_DISABLE \
        typedef _class _atl_event_classtype;\
        static const ATL::_ATL_EVENT_ENTRY<_class> map[] = {
```

Каждый макрос SINK_ENTRY_INFO добавляет в этот массив одну структуру _ATL_EVENT_ENTRY.

```
#define SINK_ENTRY_INFO(id, iid, dispid, fn, info) {id, &iid,
  (int)(INT_PTR)(static_cast<ATL::_IDispEventLocator<
    id, &iid>*>((_atl_event_classtype*)8))-8,
  dispid, (void (__stdcall _atl_event_classtype::*)())fn, info},
```

В этом макросе несколько необычны два аспекта. Показанное ниже выражение вычисляет смещение базового класса _IDispEventLocator<id, &iid> относительно производного класса (т.е. класса, в котором содержится таблица приемников). Это позволяет найти приемник события, к которому относится запись в таблице приемников.

```
(int)(INT_PTR)(static_cast<ATL::_IDispEventLocator<
  id, &iid>*>((_atl_event_classtype*)8))-8,
```

Последующее приведение типов сохраняет адрес функции-обработчика события как указатель на метод.

```
(void (__stdcall _atl_event_classtype::*)()) fn
```

[5] Возможно, вам интересно, почему обработчики событий должны использовать правила вызова __stdcall? Потому что эти правила задаются в объявлении данного указателя.

Макрос SINK_ENTRY_EX повторяет SINK_ENTRY_INFO, но вместо указателя на структуру с информацией о функции использует NULL-указатель. Макрос SINK_ENTRY такой же, как SINK_ENTRY_INFO, но использует значение IID_NULL как идентификатор интерфейса диспетчеризации и NULL-указатель вместо указателя на структуру с информацией о функции.

```
#define SINK_ENTRY_EX(id, iid, dispid, fn) \
  SINK_ENTRY_INFO(id, iid, dispid, fn, NULL)
#define SINK_ENTRY(id, dispid, fn) \
  SINK_ENTRY_EX(id, IID_NULL, dispid, fn)
```

Макрос END_SINK_MAP заканчивает массив и завершает реализацию функции _GetSinkMap.

```
#define END_SINK_MAP() {0, NULL, 0, 0, NULL, NULL} }; \
  return map;\
```

Класс IDispEventImpl

И наконец, мы добрались до класса IDispEventImpl. Это класс, используемый генерирующими код мастерами. Он является производным от IDispEventSimpleImpl и соответственно содержит всю ранее рассмотренную функциональность. Дополнительные параметры шаблона позволяют задавать библиотеку типов, описывающую интерфейс диспетчеризации источника для приемника событий.

```
template <UINT nID, class T, const IID* pdiid = &IID_NULL,
  const GUID* plibid = &GUID_NULL,
  WORD wMajor = 0, WORD wMinor = 0,
  class tihclass = CComTypeInfoHolder>
class ATL_NO_VTABLE IDispEventImpl :
  public IDispEventSimpleImpl<nID, T, pdiid>
{ ... }
```

Основная особенность класса IDispEventImpl — использование в нем информации о типах для реализации функциональности, которой не хватает в базовом классе. Класс реализует методы GetTypeInfoCount, GetTypeInfo и GetIDsOfNames, используя библиотеку типов через объект CComTypeInfoHolder.

```
STDMETHOD(GetTypeInfoCount)(UINT* pctinfo) {
  *pctinfo = 1; return S_OK; }
STDMETHOD(GetTypeInfo)(UINT itinfo, LCID lcid, ITypeInfo** pptinfo)
{ return _tih.GetTypeInfo(itinfo, lcid, pptinfo); }
STDMETHOD(GetIDsOfNames)(REFIID riid, LPOLESTR* rgszNames,
  UINT cNames, LCID lcid, DISPID* rgdispid) {
    return _tih.GetIDsOfNames(riid, rgszNames, cNames,
      lcid, rgdispid);
}
```

Кроме того, этот класс переопределяет метод GetFuncInfoFromId и инициализирует структуру _ATL_FUNC_INFO информацией, полученной из библиотеки типов.

```
HRESULT GetFuncInfoFromId(const IID& iid,
    DISPID dispidMember, LCID lcid,
    ATL_FUNC_INFO& info) {
    CComPtr<ITypeInfo> spTypeInfo;
```

```
    if (InlineIsEqualGUID(*_tih.m_plibid, GUID_NULL))
    {
        m_InnerLibid = m_libid;
        m_InnerIid = m_iid;
        _tih.m_plibid = &m_InnerLibid;
        _tih.m_pguid = &m_InnerIid;
        _tih.m_wMajor = m_wMajorVerNum;
        _tih.m_wMinor = m_wMinorVerNum;
    }
    HRESULT hr = _tih.GetTI(lcid, &spTypeInfo);
    if (FAILED(hr))
        return hr;
    return AtlGetFuncInfoFromId(spTypeInfo, iid,
        dispidMember, lcid, info);
}
```

Резюме

Протокол точек соединения определяет механизм для клиента, заинтересованного в получении уведомлений о событиях, который позволяет передать интерфейсный указатель приемника событий источнику этих событий. При этом клиент и источник событий не обязательно должны быть написаны на одном языке. Объекты, расположенные на Web-страницах или в более общем случае объекты, используемые скриптовыми языками, должны использовать протокол точек соединения, чтобы сообщать скриптовым движкам о происходящих событиях. Кроме того, элементы управления ActiveX генерируют события с помощью протокола точек соединения. Хотя этот протокол и подходит для использования внутри апартамента, он слишком громоздок и неэффективен для организации взаимодействия между несколькими апартаментами.

ATL предоставляет классы `IDispEvent` и `IDispEventSimple` для клиентских объектов, которые должны получать уведомления о событиях. Кроме того, ATL предоставляет мастер Implement Connection Point Wizard, позволяющий легко генерировать классы, которые работают с точками соединения и содержат методы для сообщения о событиях всем подсоединенным клиентам.