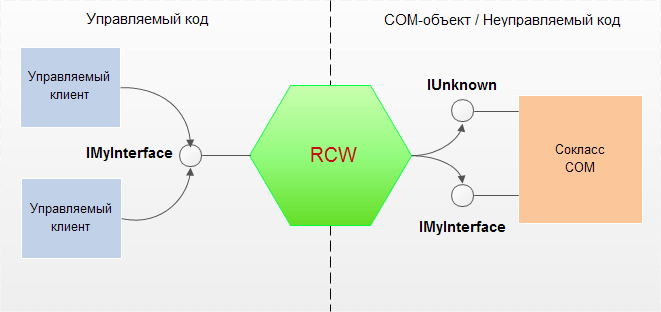
Создание COM-объекта ДАТА

**Работа с COM-объектами**

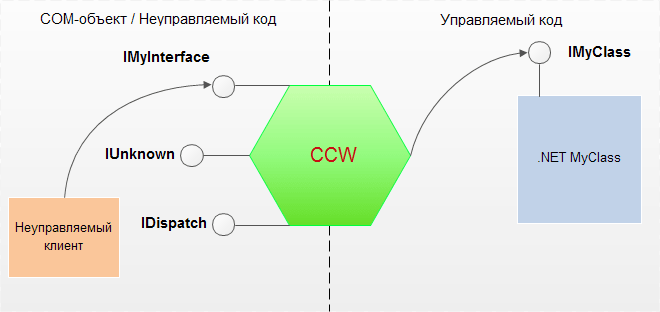
[C# и .NET Framework](https://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php) --- [Оптимизация приложений .NET Framework](https://professorweb.ru/my/csharp/optimization/level1/) --- Работа с COM-объектами

***Объектная модель программных компонентов (Component Object Model, COM)*** проектировалась с целью дать возможность создавать компоненты на любом языке/платформе, обладающем поддержкой этой модели, и использовать их в любом языке/платформе (другом), так же обладающем поддержкой COM. Платформа .NET не исключение и позволяет легко использовать сторонние COM-объекты и экспортировать типы .NET в виде COM-объектов.

Суть организации взаимодействий с COM-объектами та же, что и при использовании механизма P/Invoke: вы объявляете управляемое представление COM-объекта, а среда выполнения CLR создает объект-обертку, реализующий маршалинг. Существует две разновидности оберток: *обертка, вызываемая средой выполнения (Runtime Callable Wrapper, RCW)*, которая позволяет управляемому коду использовать COM-объекты:



и *обертка, вызываемая COM-объектами (COM Callable Wrapper, CCW)*, дающая возможность COM-объектам вызывать управляемый код:



Сторонние COM-объекты часто поставляются вместе с *основной сборкой взаимодействий (Primary Interop Assembly, PIA)*, содержащей определения, одобренные производителем, подписанной и устанавливаемой в **глобальный кеш сборок (Global Assembly Cache, GAC)**. В противном случае можно воспользоваться инструментом **tlbimp.exe**, являющийся частью Windows SDK, который автоматически генерирует сборку взаимодействий, опираясь на информацию, содержащуюся в библиотеке типов.

При взаимодействиях с COM-объектами повторно используется инфраструктура маршалинга параметров механизма P/Invoke, но с иными умолчаниями (например, по умолчанию строки преобразуются в тип BSTR), поэтому все советы, что были даны в предыдущей статье относительно механизма P/Invoke, также применимы и здесь.

Модель COM имеет собственные проблемы производительности, обусловленные характерными особенностями COM, такими как многопоточная модель подразделений и несогласованность между природой COM, основанной на подсчете ссылок, и моделью сборки мусора в .NET.

**Управление жизненным циклом**

Получая ссылку на COM-объект в .NET, вы фактически получаете ссылку на объект-обертку RCW. Обертка RCW всегда хранит единственную ссылку на COM-объект и для каждого COM-объекта создается только один экземпляр объекта-обертки RCW. Обертка RCW поддерживает собственный счетчик ссылок, не связанный со счетчиком ссылок COM-объекта. Значение этого счетчика ссылок обычно равно 1, но может быть больше, при участии в маршалинге большего числа интерфейсов или когда к одному и тому же интерфейсу обращается несколько потоков выполнения.

Как правило, когда удаляется последняя управляемая ссылка на RCW, в следующем же цикле сборки мусора в поколении, где находится обертка RCW, вызывается финализатор RCW, который уменьшает счетчик ссылок в COM-объекте (который имеет значение 1) вызовом метода Release() интерфейса *IUnknown* этого COM-объекта. COM-объект тут же уничтожает себя и освобождает занимаемую им память.

Поскольку сборщик мусора в .NET запускается в непредсказуемые моменты времени и не знает о блоках неуправляемой памяти, выделенных при создании оберток RCW для COM-объектов, он не может ускорить сборку мусора, вследствие чего объем занимаемой памяти может оказаться очень большим.

При необходимости можно вызвать метод *Marshal.ReleaseComObject()*, чтобы явно освободить объект. Каждый вызов уменьшает счетчик ссылок в RCW и когда он достигнет нуля, автоматически уменьшается счетчик ссылок в соответствующем COM-объекте (точно так же, как при вызове фииализатора RCW). После вызова метода Marshal.ReleaseComObject() нельзя использовать обертку RCW.

Если после вызова счетчик ссылок RCW может оказаться больше нуля, метод Marshal.ReleaseComObject() следует вызывать в цикле, пока он не вернет нулевое значение. Лучше всего вызывать Marshal.ReleaseComObject() внутри блока finally, чтобы гарантировать освобождение COM-объекта, даже если где-то между созданием его экземпляра и освобождением возникнет исключение.

**Маршалинг через границы подразделений**

Модель COM реализует собственные механизмы синхронизации потоков выполнения для поддержки вызовов между разными потоками, которые могут использоваться даже при работе с объектами, изначально не предназначенными для использования в многопоточной среде. Эти механизмы могут снижать производительность при неправильном их применении. Хотя эта проблема не имеет прямого отношения к взаимодействиям с COM-объектами из .NET, тем не менее, ее стоит обсудить, потому что с ней часто сталкиваются на практике, вероятно потому, что разработчики, привыкшие к типичным приемам синхронизации в .NET могут не знать, что конкретно происходит под покровом COM-объектов.

Модель COM связывает объекты и потоки выполнения с *подразделениями (apartments)*, служащими границами, через которые модель COM выполняет вызовы. Всего имеется несколько типов подразделений:

**Однопоточные подразделения (Single-Threaded Apartment, STA)**

В каждом подразделении имеется единственный поток выполнения, но может иметься любое количество объектов. В процессе может быть несколько подразделений STA.

**Многопоточные подразделения (Multi-Threaded Apartment, МТА)**

В каждом подразделении может иметься любое количество потоков выполнения и объектов, но в процессе может быть только одно подразделение МТА. Этот тип используется в .NET по умолчанию.

**Потоконезависимые подразделения (Neutral-Threaded Apartment, NTA)**

Содержат объекты, но не потоки. В процессе может быть только одно подразделение NTA.

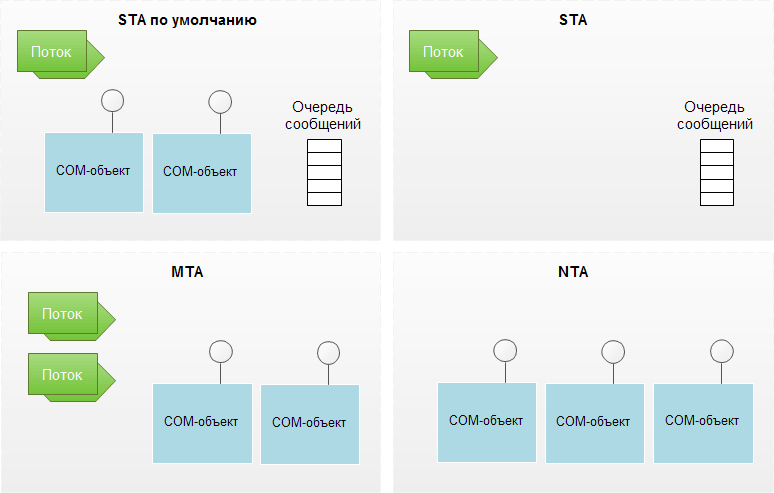
Связывание потока выполнения с подразделением происходит при вызове CoInitialize или CoInitializeEx для инициализации COM-объекта в этом потоке. Функция CoInitialize связывает поток с новым подразделением STA, а функция CoInitializeEx позволяет указать тип подразделения, STA или МТА.

В .NET вам не придется вызывать эти функции непосредственно, вместо этого достаточно добавить **атрибут STAThread** или **MTAThread** к точке входа в поток (методу Main). При желании можно также вызвать метод *Thread.SetApartmentState()* или установить значение в свойстве *Thread.ApartmentState* перед запуском потока выполнения. Если не указано иное, .NET инициализирует потоки (включая главный поток приложения) как принадлежащие подразделению МТА.

Связывание COM-объектов с подразделениями выполняется, исходя из параметра ThreadingModel в реестре, который может иметь следующие значения:

* Single - объект по умолчанию помещается в подразделение STA.
* Apartment - объект должен быть помещен в любое подразделение STA, и только потоку из этого подразделения будет позволено вызывать объект непосредственно. Другие экземпляры могут помещаться в другие подразделения STA.
* Free - объект помещается в подразделение МТА. Этот объект может вызываться непосредственно и одновременно из любого количества потоков в подразделении МТА. Объект должен обеспечивать поддержку использования в многопоточной среде.
* Both - объект помещается в подразделение создавшей его программы (STA или МТА). По сути, после создания он становится STA- или МТА-подобным объектом.
* Neutral - объект помещается в потоконезависимое подразделение и не требует маршалинга. Это - наиболее эффективный режим.

На рисунке ниже изображена схема взаимоотношений между подразделениями, потоками и объектами:



При попытке создать объект с моделью поддержки потоков, не совместимой с моделью потоков в текущем подразделении, вы получите указатель на интерфейс, который в действительности указывает на прокси-объект. Если потребуется передать интерфейс COM-объекта другому потоку, принадлежащему другому подразделению, указатель на интерфейс должен передаваться не напрямую, а через механизм маршалинга. Инфраструктура COM вернет соответствующий прокси-объект.

В процессе маршалинга вызов функции (включая параметры) преобразуется в сообщение, которое будет послано в очередь принимающего подразделения STA. Для объектов STA очередь реализуется как скрытое окно, оконная процедура которого принимает сообщения и передает COM-объекту с помощью заглушки (stub). При таком подходе COM-объекты в подразделении STA COM всегда вызываются в одном и том же потоке выполнения, благодаря чему обеспечивается безопасность при работе в многопоточном окружении.

Если вызывающее подразделение не совместимо с подразделением COM-объекта, происходит переключение потока и выполняется маршалинг параметра между потоками.

Чтобы избежать падения производительности из-за маршалинга между потоками, старайтесь обеспечить соответствие между подразделением COM-объекта и подразделением создающего его потока. Создавайте и используйте COM-объекты STA в потоках из подразделения STA, а COM объекты из подразделения МТА - в потоках МТА. COM-объекты, помеченные, как поддерживающие оба типа подразделений, могут свободно использоваться из любых потоков выполнения без лишних накладных расходов.

**Вызов объектов STA из ASP.NET**

По умолчанию среда ASP.NET выполняет страницы в потоках МТА. Если из этих страниц вызываются объекты, находящиеся в подразделениях STA, в дело вступает механизм маршалинга. Если основная масса используемых объектов принадлежит подразделениям STA, это приведет к деградации производительности. Эту проблему можно ликвидировать, пометив страницы атрибутом **AspCompat**, как показано ниже:

<%@Page Language="C#" AspCompat="true" %>

Обратите внимание, что конструкторы страниц все еще выполняются в потоке выполнения МТА, поэтому создание объектов STA следует выполнять в обработчиках событий Page\_Load и Page\_Init.

**Импортирование библиотек типов и Code Access Security**

Механизм Code Access Security выполняет те же проверки безопасности, что и P/Invoke. Вы можете добавлять ключ /unsafe при вызове утилиты tlbimp.exe, которая будет добавлять атрибут SuppressUnmanagedCodeSecurityAttribute к сгенерированным типам. Используйте эту возможность только в системах, пользующихся у вас безусловным доверием, так как она может порождать проблемы безопасности.

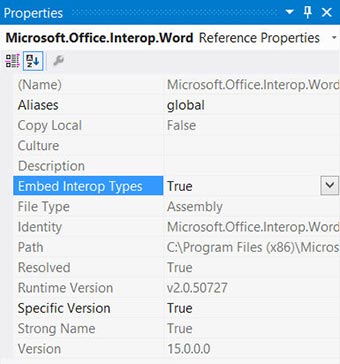
**NoPIA**

До выхода версии .NET Framework 4.0 приходилось вместе с приложением распространять сборки взаимодействий или основные сборки взаимодействий (Primary Interop Assemblies, PIA). Эти сборки обычно получались очень большими (даже в сравнении с кодом, использующим их) и как правило не входят в установочный комплект COM-компонентов; вместо этого их необходимо устанавливать отдельно, потому что сами они не требуются для работы самих COM-объектов. Другая причина, почему сборки PIA не включаются в установочные комплекты, состоит в том, что они устанавливаются в глобальный кеш сборок (GAC). Это вводит зависимость от .NET Framework в иначе полностью независимые приложения.

Начиная с версии .NET Framework 4.0, компиляторы C# и VB.NET могут проверить, какие COM-интерфейсы и методы используются в коде, и скопировать и встроить в вызывающую сборку только действительно необходимые определения, уменьшая размер кода и избавляя от необходимости распространять библиотеки PIA. В Microsoft эта особенность была названа NoPIA. Она действует как в отношении основных сборок взаимодействий, так и в отношении сборок взаимодействий в целом.

Сборки PIA обладают одной важной особенностью, которая называется *эквивалентностью типов*. Так как они имеют строгое именование и помещаются в глобальный кеш сборок, различные управляемые компоненты могут обмениваться обертками RCW и с точки зрения .NET они будут иметь эквивалентные типы. Напротив, сборки взаимодействий, сгенерированные с помощью tlbimp.exe, не обладают такой особенностью, так как каждый компонент в этом случае получит собственную, отдельную от других, сборку взаимодействий. С появлением поддержки особенности NoPIA отпала необходимость в строгом именовании сборок, и в Microsoft было предложено решение, позволяющее интерпретировать обертки RCW из других сборок, как принадлежащие тому же типу, если интерфейсы имеют одинаковый идентификатор GUID.

Чтобы включить поддержку NoPIA, выберите пункт Properties в контекстном меню Visual Studio после щелчка правой кнопкой мыши на сборке взаимодействий в разделе References, и установите параметр Embed Interop Types (Внедрять типы взаимодействий) в значение True:



**Исключения**

Большинство методов COM-интерфейсов сообщают об успехе или неудаче, возвращая значение типа HRESULT. Отрицательные значения HRESULT (с установленным старшим битом) сообщают об ошибке, а ноль (S\_OK) или положительные значения - об успехе. Кроме того, COM-объект может возвращать дополнительную информацию об ошибке при вызове функции SetErrorInfo, передавая объект IErrorInfo, созданный вызовом CreateErrorInfo.

При вызове COM-метода через механизм взаимодействий с COM, заглушка маршалера преобразует значение HRESULT в управляемое исключение, согласно самому значению HRESULT и данным, содержащимся в объекте IErrorInfo. Поскольку возбуждение исключения является достаточно дорогостоящей операцией, функции COM-объекта, которые часто терпят неудачу, могут отрицательно сказываться на производительности. Вы можете подавить автоматическое преобразование исключений, пометив методы атрибутом **PreserveSigAttribute**. При этом вам придется изменить управляемую сигнатуру, как возвращающую значение типа int, в результате чего параметр retval станет параметром out.

Начало формы

Конец формы

**C# для профессионалов. Том II**

**Робинсон Симон**

**Корнес Олли**

**Глинн Джей**

**Харвей Бартон**

**Макквин Крейг**

**Моемека Джерод**

**Нагель Кристиан**

**Скиннер Морган**

**Ватсон Карли**

**Глава 19**

**Взаимодействие с COM**

Компоненты COM и компоненты .NET не являются по своей природе совместимыми, так как они опираются на различные внутренние архитектуры. К счастью, однако, компания Microsoft предоставляет средства в SDK .NET для создания прокси COM для компонентов .NET и прокси .NET для компонентов COM. Используя эти прокси вместе с парой других технологий, организации могут использовать унаследованные компоненты COM в своих проектах .NET и могут также использовать компоненты .NET в своих приложениях, не являющихся приложениями .NET.

В этой главе будет показано, как достичь взаимодействия между COM и .NET, что поможет понять, почему COM и .NET будут играть различные, но жизненно важные роли в будущих приложениях Windows.

Сравнение COM и .NET

COM обозначает "Компонентная объектная модель" (Component Object Model). Для новичков платформы Windows понимание того, что делает COM, может быть трудным, а понимание того, как она действует, может показаться почти невозможным. Для новичков, не интересующихся созданием драйверов устройств или работой на компанию Microsoft, появление .NET и компонентов .NET позволит избежать сложностей программирования COM. Однако, так как COM была до сегодняшнего дня все же существенной частью программирования Windows, желательно быть знакомым с тем, как она действует и какие преимущества она предоставляет.

Повторное использование кода является одной из священных идей разработки программного обеспечения, и ранее разработчики добивались этого, поддерживая файлы стандартных функций. Когда разработчик хотел использовать стандартную функцию в одной из своих программ, он использовал инструкцию #include для соединения библиотечного файла и файла основной программы. После компиляции стандартные функции и программа, которая их вызывала, статически компоновались в один исполнимый файл.

Существует, по крайней мере, два недостатка у статической компоновки: она требует излишнего пространства памяти, размещая избыточные копии идентичных функций в нескольких исполнимых файлах, и, если обнаруживается ошибка в одной из стандартных функций, требуется перекомпиляция и повторное распространение всех исполнимых файлов, использующих эту функцию. Чтобы избежать таких проблем, разработчики нашли способ компилировать библиотеки функций в автономные двоичные файлы, которые могут динамически компоноваться различными исполнимыми файлами. С помощью этой схемы несколько исполнимых программ могут совместно использовать один и тот же двоичный файл динамически компонуемой библиотеки (Dynamic Link Library, DLL). Если требуется исправить ошибку в служебной функции, можно просто распространить заново DLL, в которой находится эта функция, без перекомпиляции или повторного распространения всех исполнимых файлов, которые ее используют.

Несмотря на свое удобство и эффективность, DLL имеют один серьезный недостаток: они зависят от конкретного языка программирования. Другими словами, DLL, компилированная из исходного кода С, может использоваться только из клиентской программы, которая также была написана на С. (Существуют исключения из этого правила, но такие исключения требуют от программиста специальных усилий).

COM была следующим шагом в эволюции повторного использования кода. С ее помощью программисты могли написать библиотеку классов на таком языке, как C++, откомпилировать библиотеку и использовать классы этой библиотеки из другого совместимого с COM языка программирования, такого как Delphi или Visual Basic. COM был технологией, лежащей в основе других технологий, таких как OLE (Связывание и встраивание объектов) и элементов управления ActiveX.

Службы COM+ — последняя версия технологии COM, первоначально известная как Сервер транзакций Microsoft, является частью операционной системы Windows 2000, которую компоненты COM могут использовать для общей, необходимой компонентам, функциональности: поддержки транзакций, обеспечения безопасности и события, что позволит сохранить программистам COM ценное время разработки. Более подробно об этом рассказывается в следующей главе.

Принципы работы COM

Чтобы понять, почему компоненты COM и компоненты .NET внутренне несовместимы, необходимо иметь общее понимание того, как работает COM. Далее следует существенно упрощенное объяснение. Более детальную информацию можно найти в книге "Professional COM Applications with ATL" издательства Wrox Press (ISBN 1861001703).

COM устанавливает стандарты для интерфейсов, с помощью которых клиентский код общается с массами компонентов. Так как клиентский код общается с классами компонентов только через эти стандартизованные интерфейсы, он может оставаться полностью несведущим о деталях, специфических для языка программирования, то есть о том, как реализованы эти классы компонентов. Например, используя интерфейсы COM в качестве промежуточных, клиент VB может ссылаться на компоненты COM, первоначально закодированные на C++.

Интерфейсы COM предоставляют другие возможности в дополнение к межъязыковой коммуникации. Интерфейс COM IUnknown, например, позволяет объекту COM подсчитать число клиентов, которые на него ссылаются, и автоматически выгрузить себя из памяти, когда этот счетчик уменьшится до нуля. Более того, реализуя интерфейсы, распознаваемые службами COM+, класс COM может воспользоваться написанной ранее функциональностью для безопасности, организации пула объектов, и сохранения объектов.

Каждый класс COM и каждый интерфейс, который он поддерживает, имеет 128-битный идентификатор, который гарантированно является уникальным во все времена и во всех местах. Эти глобально уникальные идентификаторы, или GUID, централизованно хранятся в реестре машины, и поддерживающие COM языки имеют средства для получения значения GUID компонента COM и использования его для вызова функциональности компонента.

Недостатки COM

Хотя COM предоставляет значительные преимущества, она имеет также недостатки. Первое: компоненты COM могут быть трудными для кодирования. В C++ разработка компонента COM включает реализацию стандартных интерфейсов COM и использование GUIDGEN.EXE для того, чтобы генерировать GUID для каждого класса и каждого интерфейса. (Хотя технологии, подобные VB и ATL Object Wizard упрощают процесс создания COM, они обеспечивают только подмножество свойств COM).

Второе: компоненты COM могут оказаться трудными для развертывания. Разработчики COM серверных компонентов предполагали обеспечить совместимость новых версий компонентов с более старыми, но не всегда это удавалось, поэтому установка нового приложения, которое ссылается на новую версию компонента COM, может внезапно дать отказ существующих приложений. Проблемы такого рода называются "адом DLL" и являются причиной большой головной боли и потерь времени.

Дополнительная информация о том, как .NET обращается с адом DLL, находится по адресу msdn.microsoft.com/library/techart/dplywithnet.htm.

Как работают компоненты .NET

Подход .NET к созданию компонентов использует многие возможности COM, игнорируя при этом ее недостатки. Компоненты должны иметь некоторый способ описания клиентам классов, которые они поддерживают. Вместо использования для этого GUID и реестра каждый файл компонента .NET инкапсулирует свое собственное описание во внутреннем сегменте, называемом манифестом (manifest).

Это означает, что размещение компонента выполняется просто: надо скопировать компонент .NET в папку исполнимого файла, который на него ссылается. Когда исполнимому файлу необходимо создать компонент, он просматривает в файле компонента в манифесте информацию, которая ему нужна. Различные версии одного и того же компонента могут существовать бок о бок на одной машине до тех пор, пока они хранятся в различных папках. (Существует центральная папка для хранения компонентов, которую можно сделать доступной для множества компонентов, она называется иногда "Global Assembly Cache" — Глобальный кэш сборок (см. главу 8).

Компоненты .NET также легко создавать в C# и VB.NET разработчик защищен от процесса создания манифеста. Разработчик создает просто проект библиотеки классов, заполняет его классами и позволяет компилятору выполнить грязную работу предоставления этих классов клиентам.

COM или .NET?

.NET не исключает COM. Многие организации, включая Microsoft, вложили значительные ресурсы в разработку COM. К тому же, поскольку компоненты .NET интерпретируются и используют среду времени выполнения .NET, они не так хорошо подходят, как компоненты COM, для сред выполнения, где скорость и эффективность являются основными требованиями.

Однако для разработки на уровне предприятия может оказаться, что компоненты .NET окажутся более практичными, чем компоненты COM. В организациях, которые действуют со скоростью Интернет, достаточно хорошее программное обеспечение, сделанное вовремя, предпочтительнее оптимизированного программного обеспечения, которое опоздало на шесть месяцев. Если программное обеспечение запускается с сервера Web, то можно гарантировать, что среда выполнения .NET правильно сконфигурирована, и таким образом получается выигрыш от сохраненного времени разработки и развертывания, что обеспечивает платформа .NET.

Речь не идет о том, что компоненты COM лучше, чем сборки .NET, или что сборки .NET лучше, чем компоненты COM. Сборки .NET, хотя и требуют больше поддержки во время выполнения, делают компонентную архитектуру более доступной для программистов.

Использование компонентов COM в .NET

В типичной организации, вероятно, невозможно (или, по крайней мере, нежелательно) выбросить компоненты COM, которые были в ней разработаны, так как компоненты .NET сделаны хорошо. Следовательно, можно ожидать ссылок на унаследованные компоненты COM из нового кода .NET, по крайней мере, в первых проектах .NET масштаба предприятия. В частности, если создается новое приложение .NET поверх существующей базы данных, вероятно, захочется использовать существующие объекты COM доступа к данным в слое доступа к данным создаваемого проекта. Если доступ к данным происходит через унаследованные компоненты, то .NET реализует бизнес-правила и доставит данные в формы ASP.NET или Windows интерфейса пользователя.

Взаимодействие COM использует классы "оболочки" и компоненты "прокси" — обычные соглашения в мире программирования в целом. Класс оболочки окружает класс, соответствующий другой архитектуре, предоставляя к нему знакомый интерфейс клиентам, которые не будут распознавать собственный интерфейс завернутого класса. Аналогично, клиент может использовать компонент прокси для доступа к компоненту, который соответствует другой архитектуре или является географически удаленным.

Диалоговое окно ссылок

Прежде чем погружаться в тонкости вопросов взаимодействия с унаследованными объектами COM, давайте кратко обсудим, как достичь такого взаимодействия. Наше объяснение строится с точки зрения IDE VS.NET, так как большинство читателей будут использовать его для написания своих программ C#, но альтернативные редакторы могут предоставить свои собственные эквивалентные методы.

Для этого понадобиться использовать диалоговое окно ссылок (References Dialog), которое доступно из пункта меню Add Reference… в меню Project IDE Visual Studio.NET. Это диалоговое окно имеет три вкладки, первая из которых просто перечисляет DLL, которые являются вспомогательными, но при этом очень важными для среды времени выполнения платформы .NET.

Самая правая вкладка, Projects, перечисляет все проекты .NET, на которые ссылаются в данном решении. На следующем экране, например, клиентское приложение C# для онлайновых заказов опирается на несколько проектов библиотек классов C#, которые реализуют бизнес-правила. (Конечно, они не обязаны быть проектами C#, чтобы на них ссылался клиент C#, они могут также легко быть проектами VB.NET. Фактически, один клиент .NET может одновременно ссылаться на проекты компонентов на нескольких различных языках платформы .NET).

Средняя вкладка, COM, является вкладкой для импортирования компонентов COM в проекте .NET.

В верхнем правом углу диалогового окна имеется кнопка Browse. При нажатии на эту кнопку появится другое диалоговое окно, которое позволит найти в файловой системе DLL COM, требующиеся проекту .NET:

Когда файл будет найден, добавляем его в список компонентов в панели формы COM:

После использования диалогового окна ссылок для нахождения DLL COM и добавления библиотеки к списку ссылок COM, можно использовать компонент COM в коде .NET. VS.NET создает пространство имен с таким же именем, как и у исходного компонента COM, и классы этого компонента COM помещаются в это пространство имен. Ссылки, создание экземпляров и вызов оболочки объекта COM производятся с таким же синтаксисом, как и у собственных объектов C#.

Посмотрим на следующий пример кода. В нем определен метод для добавления нового заказчика в базу данных. В качестве аргумента ввода этот метод получает ссылку на объект CustomerInfo, поля которого содержат имя определенного заказчика, код социального обеспечения, и т.д. Экземпляр класса доступа к данным CustomerTable создается из компонента COM и используется для вставки информации о заказчике в базу данных. В этом примере важно то, что код, связанный с объектом COM, является кодом обыкновенного объекта C#. Мы создаем экземпляр оболочки .NET и позволяем ему делегировать свою работу реальному объекту COM за сценой:

///

/// Этот код добавляет нового заказчика в базу данных,

/// обеспечивая при этом выполнение бизнес-правил.

///

public long AddNewCustomer(CustomerInfo objCustomerInfo) {

 long lngNewCustomerID;

 DataAccess.CustomerTable objCustomerTable;

 // Добавить запись в таблицу заказчиков.

 objCustomerTable = new DataAccess.CustomerTable();

 lngNewCustomerID =

  objCustomerTable.InsertRecord(

  objCustomerInfo.LastName, objCustomerInfо.FirstName,

  objCustomerInfo.MiddleName, objCustomerInfo.SocialSecurityNumber);

Конечно, если надоест вводить пространство имен компонента COM, то инструкция using в начале каждого файла позволит сослаться на классы в оболочке компонента COM с помощью сокращенных, относительных имен:

// Размещение следующей инструкции в начале файла ...

using DataAccess;

// позволяет ссылаться на классы DataAccess по их относительным именам

CustomerTable objCustomerTable;

objCustomerTable = new CustomerTable();

IDE VS.NET будет даже использовать Intellisense, чтобы помочь запомнить членов класса компонента:

Intellisense поможет также с членами данных и списком аргументов.

Оболочки времени выполнения

Чтобы добавить ссылку на DLL COM при использовании диалогового окна References, IDE VS.NET делает некоторую работу за сценой. В частности, создается компонент прокси .NET для DLL COM и копия DLL COM помещается в каталог проекта .NET.

Вспомните, что компоненты .NET описывают сами себя, в то время как компоненты COM хранят свои описания в реестре. Прокси, который генерирует IDE VS.NET, описывает DLL COM и служит для нее в качестве делегата, пересылая вызовы от клиента .NET через службы COM в DLL COM, которую он обертывает. Клиент .NET не знает, что он вызывает компонент COM; он общается только с прокси и получает данные, которые прокси пересылает обратно из DLL COM.

В терминологии .NET такой прокси называется оболочкой времени выполнения, или RCW. IDE создает DLL, которая имеет такое же имя, как и исходный компонент COM, но является в действительности .NET RCW, которая просто обертывает первоначальный компонент, предоставляя его клиентам .NET через интерфейс .NET, который они могут понять. Интересное дополнительное замечание: IDE VS.NET создает оболочку не только для каждой импортируемой DLL COM, но для каждой DLL COM, на которую ссылается импортированная DLL COM в своем открытом интерфейсе, поэтому в этой папке будет также находиться файл ADODB.dll, на который в DataAccess.dll существует ссылка.

В рассматриваемом примере DLL COM, DataAccess.dll предоставляет методы для вставки, извлечения, обновления и удаления записей в нескольких таблицах базы данных. Так как методы вставки возвращают множества записей ADO, клиентам компонента COM требуется ссылка на библиотеку типов ADO. При импорте DataAccess.dll среда разработки распознает эту необходимость и автоматически создает также RCW для компонента COM ADO.

TlbImp.exe

Опасно ли для RCW, созданной IDE VS.NET, иметь такое же имя, как и исходная DLL COM? В конце концов, если приложение переносится с одной машины на другую, будет очень легко спутать RCW с DLL COM, которую он заворачивает, и случайно перезаписать файл DLL COM. Или можно по ошибке попытаться зарегистрировать RCW в службах COM, удивляясь, почему программа регистрации (regsvr32.exe) не работает.

Чтобы избежать таких проблем, попробуйте воспользоваться напрямую утилитой TlbImp.exe. Поставляемая вместе с SDK .NET эта исполняемая программа специализируется на создании прокси .NET для DLL COM. Так как она вызывается из командной строки, можно задать TlbImp.exe с аргументом командной строки out, чтобы получающийся RCW имел имя, отличное от имени DLL COM.

TlbImp является сокращением от Type Library Importer (Импортер библиотеки типов). При выполнении этой программы с DLL COM она запрашивает библиотеку типов DLL COM и транслирует информацию оттуда в формат .NET, преобразуя стандартные типы данных COM в типы, распознаваемые .NET. После выполнения TlbImp.exe для DLL COM, файл вывода (RCW) просто помещается в папку исполняемого файла клиента, который будет его использовать. (Это дополнительный шаг, который необходим при явном использовании TlbImp.exe вместо диалогового окна References в IDE.) В приведенном выше примере TlbImp.exe выполняется с DLL COM, которая находится в той же папке, что и TlbImp, но TlbImp.exe работает, как любая другая утилита командной строки, т.е. можно определить файл в другой папке с помощью абсолютного или относительного пути доступа.

В заключение пара предупреждении о TlbImp.exe и RCW. Первое: не забывайте задавать аргумент out при использовании TlbImp.exe. Если не сделать этого, то программа TlbImp.exe будет заявлять, что она не может перезаписать исходный файл:

Второе: помните, что хотя RCW служит в качестве посредника между компонентом COM и клиентом .NET, который его вызывает, по-прежнему всю реальную работу делает компонент COM. То есть необходимо выполнить те же требования к развертыванию компонента COM, которые пришлось бы сделать, если бы он использовался напрямую. Это означает, что завернутый компонент COM тем не менее надо регистрировать в службах COM. Если попробовать сослаться на незарегистрированный компонент COM, то IDE VS.NET будет порождать ошибку.

Чтобы справиться с этой проблемой, понадобится, конечно, программа регистрации COM — regsvr32.exe. Ее можно вызвать из диалогового окна Windows Run, которое доступно из меню Start рабочего стола.

Поэтому не забывайте регистрировать компоненты COM.

Позднее связывание

с

компонентами COM

Прежде чем исполнимый файл клиента сможет вызвать методы и свойства объекта компонента, ему необходимо узнать адреса памяти этих методов и свойств. Существуют две различные технологии, которые клиентские программы могут использовать для определения этих адресов.

Программы с ранним связыванием узнают адреса на ранней стадии процесса компиляции/выполнения — во время компиляции. Когда программа (ранним связыванием компилируется, компилятор использует библиотеку типов компонента для включения адресов методов и свойств компонента в клиентскую исполнимую программу, чтобы к адресам можно было очень быстро и безошибочно обращаться. Технологии взаимодействия COM которые были рассмотрены до сих пор, используют раннее связывание.

Со своей стороны, программы с поздним связыванием узнают адреса свойств и методов на поздней стадии процесса компиляции/выполнения, в тот самый момент, когда эти свойства и методы вызываются. Код с поздним связыванием обычно обращается к клиентским объектам через базовые типы данных, такие как object, и использует среду времени выполнения для динамического определения адресов методов. Хотя код с поздним связыванием позволяет использовать некоторые сложные технологии программирования, такие как полиморфизм, он требует некоторых связанных расходов, которые мы вскоре увидим.

Но сначала проверим, как позднее связывание выполняется с помощью отражения в C# (Отражение, является способом, который используется кодом во время выполнения для определения информации об интерфейсах серверных классов; см. главу 5.)

При позднем связывании с объектом COM в программе C# не нужно создавать RCW для компонента COM. Вместо этого вызывается метод класса GetTypeFromProgID класса Type для создания экземпляра объекта, представляющего тип объекта COM. Класс Type является членом пространства имен System.Runtime.InteropServices и в коде ниже мы конфигурируем объект Type для того же компонента COM доступа к данным, который использовался в предыдущих примерах:

using System.Runtime.InteropServices;

Type objCustomerTableType;

objCustomerTableType = Type.GetTypeFromProgID("DataAccess.CustomerTable");

Когда имеется объект Type, инкапсулирующий информацию о типе объекта COM, он используется для создания экземпляра самого объекта COM. Это реализуется передачей объекта Type в метод класса CreateInstance класса Activator.CreateInstance создает экземпляр объекта COM и возвращает на него ссылку позднего связывания, которую можно сохранить в ссылке типа object.

object objCustomerTable;

objCustomerTable = Activator.CreateInstance(objCustomerTableType);

В этом месте код C# имеет ссылку позднего связывания на готовый экземпляр класса COM.

К сожалению, невозможно вызывать методы непосредственно на ссылке типа object. Чтобы можно было обратиться к объекту COM, необходимо использовать метод InvokeMember объекта Type, который был создан вначале. При вызове метода InvokeMember ему передается ссылка на объект COM вместе с именем вызываемого метода COM, а также массив типа object всех входящих аргументов метода.

ObjCustomerTableType.InvokeMember("Delete", BindingFlags.InvokeMethod, null, objCustomerTable, aryInputArgs);

Напомним еще раз последовательность действий:

1. Создать объект Type для типа объекта COM с помощью метода класса Type.GetTypeFromProgID().

2. Использовать этот объект Type для создания объекта COM с помощью Activator.CreateInstance().

3. Методы вызываются на объекте COM, вызывая метод InvokeMember на объекте Type и передавая в него ссылку object в качестве входящего аргумента. Ниже приведен пример кода, объединяющий все это в один блок:

using System.Runtime.InteropServices;

Type objCustomerTableType;

object objCustomerTable;

objCustomerTableType=Type.GetTypeFromProgID("DataAccess.CustomerTable");

objCustomerTable=Activator.CreateInstance(ObjCustomerTableType);

objCustomerTableType.InvokeMember("Delete", BindingFlags, InvokeMethod, null, objCustomerTable, aryInputArgs);

objCustomerTableType = Type.GetTypeFromProgID("DataAccess.CustomerTable");

Хотя средства позднего связывания C# позволяют избежать трудностей RCW, необходимо знать о некоторых, связанных с этим, недостатках.

Первый: позднее связывание может быть опасным. При использовании раннего связывания компилятор может запросить библиотеку типов компонента COM, чтобы убедиться, что все вызываемые на объектах COM методы в действительности существуют. При позднем связывании ничто не препятствует опечатке в вызове метода InvokeMember(), что может породить ошибку во время выполнения.

Второй: позднее связывание может быть медленным. Каждый раз при использовании InvokeMember() на объектной ссылке среда времени выполнения должна найти требуемый член в библиотеке функций класса COM. Это приводит к снижению производительности программы.

Третий: написание кода с поздним связыванием может оказаться трудоемким. Так как не требуется ссылаться на библиотеку типов компонента COM, IDE VS.NET не может использовать Intellisense, чтобы помочь с именами членов и списками аргументов, поэтому в коде могут появиться ошибки, которые будет трудно найти до времени выполнения.

Использование элементов управления ActiveX в .NET

Элемент управления ActiveX является частным типом компонента COM, который поддерживает специальное множество интерфейсов для обеспечения графического представления. Также, как можно импортировать стандартные компоненты COM для использования в проектах .NET, можно также импортировать элементы управления ActiveX. Это позволяет сделать утилита AxImp.exe.

AxImp.exe

Чтобы импортировать компонент ActiveX в среду .NET, утилита AxImp.exe вызывается из командной строки. Команда состоит из двух частей:

1. Имени AxImp.

2. Абсолютного или относительного пути доступа к файлу ActiveX (\*.осх), который должен быть импортирован.

Для примера рассмотрим снимок экрана, приведенный ниже. Здесь импортируется элемент управления ActiveX Win32 MAPI и определяется расположение файла .осх (C:\windows\system\msmapi32.осх):

Как можно видеть, программа AxImp.exe выводит два файла.

Первый выходной файл MSMAPI.dll является прокси для сборки. Он позволяет ссылаться на компонент ActiveX, как если бы это был неграфический объект, с помощью его методов и свойств таким же образом, каким используются методы и свойства неграфического класса.

Второй файл AxMSMAPI.dll является элементом управления Windows. Он позволяет использовать графический аспект импортированного элемента управления ActiveX как элемент управления Windows в проектах Forms Windows .NET.

При использовании AxImp желательно убедиться, что для этих двух выходных файлов предоставляются явные и уникальные имена, так как, по крайней мере, в бета-версии SDK .NET AxImp.exe имеет потенциальную возможность перезаписывать входные файлы без предупреждения.

Ссылка на сборку прокси ActiveX

Как сказано выше, вывод сборки прокси утилитой AxImp.exe позволяет ссылаться на компонент ActiveX программным путем без использования графических аспектов. Для этого необходимо просто добавить ссылку на прокси сборки с помощью вкладки Reference в IDE VS.NET после того, как AxImp.exe сделает свою работу:

После создания ссылки на сборку прокси можно использовать в коде программы компонент ActiveX. В примере ниже используется прокси MSMAPI32.осх для отправки почтового сообщения:

public static int Main() {

 MAPISession objSession=new MAPISession();

 MAPIMessages objMessage=new MAPIMessages();

 objSession.Password="password";

 objSession.UserName="me@mydomain.com";

 objSession.SignOn();

 objMessage.SessionID=objSession.SessionID;

 objMessage.Compose();

 objMessage.MsgSubject="MAPI Contol Test";

 objMessage.MsgNoteText="The message body.";

 objMessage.RecipAddress="you@yourdomain.com";

 objMessage.ResolveName();

 objMessage.Send(null);

 objSession.SignOff();

 return 0;

}

Размещение элемента управления ActiveX в WinForm

Также достаточно просто разместить элемент управления ActiveX в форме Windows. Чтобы сделать это, сначала запустим диалоговое окно Customize Toolbox из IDE VS.NET, щелкнув правой кнопкой мыши в панели инструментов IDE и выбрав соответствующий пункт в контекстном меню. Когда появится диалоговое окно, надо перейти на вкладку .NET Framework Components и найти созданный утилитой AxImp.exe файл элемента управления Windows. После закрытия диалогового окна Customize Toolbox импортированный элемент управления появится в панели управления IDE и можно будет добавить его в Windows Form таким же образом, каким добавляются любые другие элементы управления Windows.

Использование компонентов .NET в COM

Также, как используются компоненты COM и элементы управления ActiveX в коде .NET, можно применять компоненты .NET в стандартном коде Windows. Только несколько свойств сборок .NET недоступны через COM, включая параметризованные конструкторы, статические методы и константные поля. Кроме того, доступ к перегруженным методам .NET из COM требует дополнительной работы.

RegAsm.exe

Применение компонентов COM в коде .NET требует другой утилиты, которая существует в SDK .NET, являющейся аналогом программы импорта библиотеки типов, которую мы видели ранее. Название этой утилиты — RegAsm.exe.

Название утилиты RegAsm (Register Assembly) обозначает ее функцию, она отвечает за ввод информации о типе компонента .NET в системный реестр, чтобы службы COM могли к нему обратиться. После регистрации компонента .NET с помощью RegAsm, стандартные клиенты Windows могут сделать позднее соединение с классами компонента. Процесс регистрации компонента должен быть сделан только один раз. После регистрации все клиенты COM могут к нему обращаться.

В качестве примера рассмотрим следующий код. Он принадлежит классу в библиотеке классов .NET. Функция получает просто число в качестве аргумента ввода и возвращает факториал этого числа:

namespace Factorial {

 using System;

 public class Factorial {

  // Этот метод вычисляет факториал числа

  public int ComputeFactorial(int n) {

   int intFactorial=n;

   for (int i = 1; i<="" p="">

   intFactorial\*=i;

  }

  return intFactorial;

 }

 }

}

После компиляции класса примера в сборку .NET можно зарегистрировать эту сборку в службах COM с помощью RegAsm.exe:

Теперь, когда сборка зарегистрирована в службах COM с помощью RegAsm, можно выполнить позднее связывание со сборкой .NET через службы COM. С целью демонстрации создадим простой сценарий VB, который это делает. (Сценарий VB можно создать с помощью текстового редактора, такого как Notepad; введите просто следующий код и сохраните файл с расширением .vbs. Предполагая, что Windows Script Host установлен, файл при вызове будет выполняться как сценарий. Помните, что сценарий VB выполняет позднее связывание для компонентов COM.)

Option Explicit

Dim objFactorial

Dim lngResult

Dim lngInputValue

Set objFactorial=CreateObject("Factorial.Factorial")

lngInputValue=InputBox("Numbers?")

IngResult=objFactorial.ComputeFactorial(CLng(lngInputValue))

Call MsgBox(lngResult)

Однако, прежде чем это можно будет использовать, необходимо установить сборку в глобальный кэш. Для этого сначала создадим устойчивое имя сборки с помощью следующей команды:

sn -k Factorial.dll

Далее необходимо создать файл AssemblyInfo.cs со следующим содержимым:

using System.Reflection;

[assembly: AssemblyKeyFile("factorial.snk")]

Затем это надо откомпилировать с помощью следующей команды, чтобы превратить в модуль:

CSC /t:module /out: AssemblyInfo.dll AssemblyInfo.cs

После этого компилируется файл Factorial.cs, и полученная DLL устанавливается в глобальный кэш с помощью gacutil следующим образом:

csc /t:library /addmodule:assemblyinfо.dll Factorial.cs gacutil /i Factorial.dll

При выполнении сценарий VB воспользуется службами COM для создания объекта .NET, вызовет метод на этом объекте и выведет возвращаемое из объекта .NET значение в окне сообщения:

Интересная техника, не правда ли? Но она не решает ни одной из упомянутых выше проблем, связанных с поздним связыванием. К счастью, другой член набора инструментов SDK .NET может в этом помочь. Это утилита TlbExp.exe.

Однако, прежде чем попрощаться с TlbImp.exe, необходимо запомнить одну вещь: службы COM должны иметь возможность найти компонент сборки .NET, когда он вызывается. Это означает, что сборка должна располагаться в рабочей папке клиента или в глобальном коде сборки системы.

TlbExp.exe

TlbExp обозначает Type Library Exporter (экспортер библиотеки типов). При выполнении на файле сборки .NET TlbExp может запросить внутренний манифест этой сборки и вывести соответствующий файл библиотеки типов COM (\*.tlb). Когда TlbExp создаст файл библиотеки типов для компонента .NET, языки разработки не принадлежащие .NET, такие как VB6, смогут ссылаться на него, используя для того, чтобы получить эффективное раннее связывание с компонентами .NET:

Как можно было уже догадаться, утилиты TlbExp и RegAsm были созданы для работы в тесном взаимодействии. RegAsm используется для регистрации компонента .NET в службах COM, а затем TlbExp — для экспорта библиотеки типов COM клиенту и ссылки на нее из языков не принадлежащих .NET.

Службы вызова платформы

Мы рассказали о взаимодействии между компонентами COM и .NET. Теперь поговорим о другом виде — о взаимодействии между кодом .NET и так называемом неуправляемом коде. Его обеспечивает технология, называемая службами вызова платформы, или, кратко, PInvoke.

Неуправляемый код и ненадежный код

Первое, о чем необходимо знать, — термины неуправляемый и ненадежный не являются синонимами.

Ненадежный код C# является кодом, который встроен в блок с префиксом в виде ключевого слова unsafe. Код в таком блоке может использовать весь диапазон идиом C++, таких как указатели и массивы на основе стека. Он считается ненадежным, так как такие идиомы часто ассоциируются с ошибками, но такой код по-прежнему управляется средой времени выполнения .NET.

Со своей стороны, неуправляемый код не управляется средой времени выполнения .NET. Когда поток выполнения приложения .NET входит в сегмент неуправляемого кода, среда времени выполнения .NET больше не имеет контроля над действиями этого кода, и не может обеспечить для него сборку мусора или правила безопасности (по этой причине приложения, которые используют неуправляемый код, должны наделяться доверием со стороны системного администратора).

Службы вызова платформы позволяют коду .NET взаимодействовать не только с ненадежным кодом, но и с достоверно неуправляемым.

Доступ к неуправляемому коду

Хотя .NET может взаимодействовать с неуправляемым кодом в любой DLL, чаще всего он взаимодействует с кодом в DLL, составляющим базовую функциональность Windows API. Это включает user32.dll, gdi32.dll и kernel32.dll. Процесс представления функций из этих DLL для кода .NET должен быть знаком любому, кто использовал ключевое слово Declare для предоставления вызовов Win32 API для кода VB6:

[sysimport(dll="user32.dll")]

public static extern int MessageBoxA(int Modal, string Message, string Caption, int Options);

В приведенном примере с помощью оболочки .NET вызова осуществляется вызов Windows API, который выводит окно с сообщением. В атрибуте выше функции оболочки определяется DLL, которой функция в оболочке должна делегировать свою работу. Теперь клиент кода .NET может вызывать функцию оболочки для вызова функций API:

MessageBoxA(0, "PInvoke worked!", "PInvoke Example", 0);

Хотя здесь для функции оболочки задано такое же имя, как и для вызова Windows API, в который она отображается, можно задать для нее и другое имя, как делалось в примере выше. Изменим теперь имя "MessageBox" на имя, которое более точно определяет, как будет использоваться вызов API. Сделаем это, определяя дополнительное значение в атрибуте sysimport:

[sysimport (dll="ustr32.dll", name="MessageBoxA") ]

public static extern int ErrorMessage(int Modal, string Message, string Caption, int Options);

При переименовании вызова Windows API таким образом клиенты могут вызывать функцию с новым именем:

ErrorMessage(0, "PInvoke worked!", "PInvoke.Example", 0);

Недостатки PInvoke

Мы видели, что достаточно просто ссылаться и вызывать неуправляемую функцию из кода .NET. К сожалению, существуют потенциальные недостатки использования неуправляемого кода.

Хотя Microsoft сознательно откладывает вопрос взаимодействия платформ, многие люди подозревают, что оно уже на горизонте для платформы .NET. При взаимодействии платформ можно выполнить программу .NET на любой платформе — от Macintosh до Unix при условии, что платформ? обеспечена средой времени выполнения .NET. Однако при использовании PInvoke, код .NET соединяется с операционной системой Windows.

Рассматривая использование PInvoke, прежде всего необходимо проверить, что требуемая функциональность не представлена базовым классом .NET. Если среда времени выполнения .NET будет когда-либо перенесена на другую платформу, базовые классы .NET также будут перенесены, и код получит шанс правильно выполниться на новой платформе, возможно, с небольшими изменениями.

Заключение

Как показывает эта глава, COM и .NET являются различными технологиями, которые способны работать совместно, если применяются соответствующие технологии. Используя утилиты взаимодействия, такие как TlbImp.exe, RegAsm.exe и TlbExp.exe, разработчики могут применять унаследованные компоненты COM в качестве строительных блоков для новых приложений .NET.

По сравнению с компонентами COM сборки легче создавать, развертывать и поддерживать. Маловероятно, что .NET когда-либо полностью заменит разработчикам COM, для которых скорость выполнения является основной задачей. Однако, скорее всего, разработчики приложений Web и программ, которые внутренне используются организациями, сочтут сборки .NET избавлением от ада DLL.

Реализация COM-коллекций средствами C#

##### Автор: [Сергей Иванов](http://rsdn.org/account/info/17246) JSC Dorogobuzh Источник: [RSDN Magazine #3-2005](http://rsdn.ru/mag/main.htm)

###### Опубликовано: 07.10.2005 Исправлено: 10.12.2016 Версия текста: 1.0

|  |  |
| --- | --- |
| [**Предисловие**](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#E5) [**Что такое COM-коллекция**](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EUB) [**Реализация коллекций**](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EDE)  [Ограничения](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EHE) [Реализация COM-интерфейса IEnumVARIANT](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#E2E) [Общая схема реализации коллекций](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EOHAC) [Нетипизированная коллекция](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EQIAC) [Типизированная коллекция](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EOKAC) [Коллекция строк](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EZLAC) [Value-type коллекции](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EINAC) [Коллекция с несколькими типами индексации](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#ESOAC)  [**Заключение**](https://rsdn.org/article/dotnet/ComCollectionsNet.xml#EMBAE) |  |

## Предисловие

Несмотря на явные преимущества управляемого кода .Net бывают ситуации, когда при построении программных систем приходится заниматься разработкой COM-объектов. Так, например, для организации пользовательских интерфейсов в системах уровня предприятия нередко используются средства Microsoft Office, функциональные возможности которых расширяются за счет макросов (программ), реализуемых с помощью VBA (Visual Basic for Applications). Возможности же VBA в свою очередь расширяются именно за счет использования COM-объектов.

|  |
| --- |
| **ПРИМЕЧАНИЕ**  Здесь имеются в виду версии Office, которые напрямую не поддерживают .Net, т.е. до версии Microsoft Office 2003 SP1. |

Другая ситуация, в которой приходится использовать COM не по желанию, а по необходимости – разработка дополнительной функциональности для существующих программ, которые расширяются за счет COM. Примеры подобных программ – Microsoft Outlook, Microsoft Internet Explorer и др.

Возможны и другие ситуации, когда без COM не обойтись – например, реализация с помощью .Net существующих COM-протоколов. Частный пример – разработка OPC-серверов для систем сбора информации (OPC – OLE for Process Control).

Таким образом, хотя разработка COM-объектов и не является главной возможностью .Net, владение этой техникой может быть весьма полезным.

В настоящей статье не будет рассматриваться C++ with managed extensions – разработка COM-объектов с помощью этого языка программирования не очень существенно отличается от традиционной разработки с помощью C++. Ограничимся лишь C#, который позволяет разрабатывать COM-объекты гораздо проще, чем C++ или Delphi (субъективное мнение автора, основанное на личном практическом опыте).

Более того, не будут здесь рассматриваться ни общие вопросы COM, ни конкретные механизмы и технологии C# для разработки COM – все это достаточно подробно документировано в MSDN. Будет рассмотрен абсолютно частный вопрос, который, к сожалению, не нашел отражения ни в MSDN, ни в других источниках информации, доступных автору, включая поиск в Сети – как средствами C# реализовать COM-коллекции.

## Что такое COM-коллекция

Руководство по программированию на Visual Basic для Microsoft Excel 5.0 определяет термин *Automation collection* следующим образом:

*«A group of objects. An object's position in the collection can change whenever a change occurs in the collection. Therefore, the position of any specific object in the collection is unpredictable. This unpredictability distinguishes a collection from an array.»*

Более детальное определение COM-коллекции включает следующие моменты:

1. Доступ к коллекции, а также ее элементам, осуществляется через OLE-интерфейс. Коллекция представляет собой COM-объект.
2. Элементами коллекции могут быть как простые данные (число, строка и пр.), так и COM-объекты. Элементы коллекции, как правило, однородны, т.е. представляют собой данные одного и того же типа. Но однородность данных не является обязательной – в общем случае элементы коллекции могут быть типа OLE-variant и содержать данные разного типа.
3. Интерфейс коллекции должен содержать свойство Count, через которое можно получить количество элементов в коллекции.
4. Интерфейс коллекции должен содержать метод Item, с помощью которого можно получить конкретный элемент коллекции (в некоторых руководствах, например, <http://www.rsdn.ru/article/com/comcoll.xml>, вместо метода рассматривается свойство с параметром). Указание элемента осуществляется через параметр метода, который соответствует некоторому логическому индексу элемента. При целочисленной индексации рекомендуется использовать нумерацию элементов коллекции с 1. Если метод Item возвращает значение типа OLE-variant, коллекция считается нетипизированной. В остальных случаях коллекция считается типизированной.
5. Интерфейс коллекции должен содержать свойство \_NewEnum, которое возвращает нумератор коллекции. Нумератор представляет собой COM-объект, реализующий интерфейс IEnumVARIANT, и обеспечивает для внешних языков программирования типа Visual Basic, VBA, VBScript и пр. выполнение циклов for each.
6. Интерфейс коллекции может содержать другие необходимые свойства и методы, а также события. В частности, для изменяемых коллекций рекомендуется включать методы Add (добавление элементов) и Remove (удаление элементов).

С точки зрения внешнего кода (например, VBA) использовать COM-коллекции можно двумя способами.

Первый способ заключается в использовании обычного цикла for, например:

|  |
| --- |
| For I = 1 To collection.Count  Set element = collection.Item(I)  ' некоторые действия с элементом коллекции Next I |

или с учетом доступа к значению по умолчанию:

|  |
| --- |
| For I = 1 To collection.Count  Set element = collection(I)  ' некоторые действия с элементом коллекции Next I |

Второй способ заключается в использовании цикла типа for each (поддерживается не всеми языками программирования):

|  |
| --- |
| For  Each element In collection  ' некоторые действия с элементом коллекции Next |

## Реализация коллекций

### Ограничения

Для упрощения кода примем следующие ограничения для разрабатываемых коллекций:

1. Будет реализован минимальный набор функциональности коллекций, т.е. коллекции будут неизменяемыми, дополнительные методы и свойства использоваться не будут.
2. Коллекции будут использоваться только в Visual Basic, VBA, VBScript.
3. Все коллекции будут использовать простые массивы в качестве внутреннего хранилища данных.
4. Для всех коллекций будет использоваться единая реализация нумератора (реализация COM-интерфейса IEnumVARIANT).

Примем также единую нумерацию элементов всех коллекций, начиная с 1.

### Реализация COM-интерфейса IEnumVARIANT

#### Общие положения

COM-интерфейс IEnumVARIANT определен в библиотеке stdole32.dll, которая находится в каталоге <Windows>\system32. Указанная библиотека сопровождается файлом stdole32.tlb, что позволяет импортировать интерфейс IEnumVARIANT в .Net с помощью утилиты tlbimp.exe из состава .Net Framework SDK. Результат импорта – сборка, аналогичная сборке stdole.dll из поставки Microsoft .NET Primary Interop Assemblies (PIA).

К сожалению, импортированный таким образом интерфейс оказывается неработоспособным при обратной передаче в COM.

PIA-сборку можно построить и вручную, но это занятие хлопотное и к тому же неблагодарное, т.к. пространство имен System.Runtime.InteropServices уже включает определение интерфейса UCOMIEnumVARIANT, соответствующего IEnumVARIANT.

Назначение и рекомендации по реализации всех методов интерфейса IEnumVARIANT хорошо документированы в MSDN. Отметим лишь, что этот интерфейс в разном программном окружении может использоваться с некоторыми вариациями. Так, Visual Basic, VBA, VBScript используют следующую схему работы с интерфейсом (псевдокод C#):

|  |
| --- |
| IEnumVARIANT enumerator = collection.\_NewEnum();  while (true)  {  object element;  int pceltFetched;  int hr = enumerator.Next(1, out element, out pceltFetched);  if (hr == 0)  // некоторые действия с elementelsebreak;  } |

Таким образом, если потенциальное использование интерфейса ограничить только указанными языками программирования, можно существенно упростить реализацию.

С целью же универсального использования различных .Net-коллекций (Array, ArrayList, etc.) можно использовать интерфейс System.Collections.IEnumerator, получаемый через интерфейс IEnumerable реальной коллекции.

С учетом изложенного общее определение класса, реализующего COM-интерфейс IEnumVARIANT, будет следующим (константы S\_OK и S\_FALSE будут использованы позже в реализации методов):

|  |
| --- |
| public  class \_EnumVariant : UCOMIEnumVARIANT  {  privateconstint S\_OK = 0;  privateconstint S\_FALSE = 1;  public \_EnumVariant(IEnumerator enumerator)  {  this.enumerator = enumerator;  }  publicvoid Clone(int ppenum) { ... }  publicint Next(int celt, int rgvar, int pceltFetched) { ... }  publicint Reset(){ ... }  publicint Skip(int celt) { ... }  IEnumerator enumerator = null;  } |

Отметим, что в классе \_EnumVariant не предусматривается конструктор без параметров. Это позволяет исключить создание экземпляров нумератора во внешнем коде. Кроме того, имя класса начинается с подчеркивания, что в соответствии с рекомендациями Microsoft позволяет сделать его «невидимым» для Visual Basic, VBA, VBScript и пр.

Далее рассмотрим реализацию отдельных методов.

#### Метод Clone

С учетом того, что разрабатываемый нумератор будет использоваться только в контексте Visual Basic, VBA, VBScript (см. Общие положения), метод можно не реализовывать:

|  |
| --- |
| public  void Clone(int ppenum) { thrownew NotSupportedException(); } |

Если же в другом контексте этот метод необходим, то реализация класса изменится принципиально, т.к. клонирование через IEnumerator невозможно.

Кроме того, в соответствии с описанием интерфейса IEnumVARIANT через параметр ppenum придется возвращать указатель на новый нумератор, а сигнатура C#-метода на первый взгляд для этого не очень подходит. Впрочем, желаемого можно добиться и в такой ситуации (соответствующие решения см. в реализации метода Next).

#### Метод Reset

В разрабатываемом контексте (см. Общие положения) метод Reset не является необходимым, но легко реализуется через IEnumerator:

|  |
| --- |
| public  int Reset()  {  enumerator.Reset();  return S\_OK;  } |

Единственный нюанс реализации – возврат значения S\_OK. Впрочем, если бы мы описывали подобный интерфейс вручную, то получили бы аналогичную ситуацию при использовании следующей конструкции:

|  |
| --- |
| [PreserveSig]  int Reset(); |

#### Метод Skip

Реализация метода через IEnumerator также не представляет труда:

|  |
| --- |
| public  int Skip(int celt)  {  for ( ; celt > 0; celt--)  if (!enumerator.MoveNext())  return S\_FALSE;  return S\_OK;  } |

Отметим только, что в реализации этого метода предполагается значение параметра celt, большее 0. Это соответствует описанию метода IEnumVARIANT::Skip в MSDN.

#### Метод Next

Данный метод является по сути единственным значимым и достаточно нестандартным с точки зрения реализации. Поэтому разберем его подробнее.

Сравним сигнатуры методов IEnumVARIANT.Next (OLE) и UCOMIEnumVARIANT.Next (C#):

|  |
| --- |
| [OLE]  HRESULT Next(  unsignedlong celt,  VARIANT FAR\* rgVar,  unsignedlong FAR\* pCeltFetched  );  [C#]  publicint Next(int celt, int rgvar, int pceltFetched); |

Возвращаемое значение формируется аналогично методу Reset.

Параметр celt тоже не представляет проблем – в обоих случаях это обычный входной параметр.

С параметром pCeltFetched ситуация несколько сложнее. В соответствии с описанием метода IEnumVARIANT.Next через этот параметр передается указатель на переменную, в которую записывается количество считанных элементов или null, если внешнее окружение не использует эту информацию. Такая конструкция в C# не может быть определена ни через ref-, ни через out-параметр, поэтому тип этого параметра и определен как int (хотя, наверное, удобнее было бы использовать IntPtr).

Тем не менее, можно использовать класс Marshal, чтобы напрямую записать результат в неуправляемую память:

|  |
| --- |
| IntPtr pceltFetchedPtr = new IntPtr(pceltFetched);  if (pceltFetchedPtr != IntPtr.Zero)  Marshal.WriteInt32(pceltFetchedPtr, 0); |

Параметр rgVar создает еще большую проблему, т.к. через него внешний код передает массив (неинициализированных) значений типа OLE-variant емкостью celt, а реализация метода должна заполнить этот массив значениями считанных элементов коллекции.

Такую функциональность можно реализовать с помощью ручного маршалинга посредством класса Marshal и небольшого трюкачества с unsafe-кодом:

|  |
| --- |
| int\* p = (int\*)rgvar;  for (int i = 0; i < celt; i++)  {  if (enumerator.MoveNext())  Marshal.GetNativeVariantForObject(  enumerator.Current,  new IntPtr((void\*)p));  p += 4;  } |

Трюк заключается в том, что для перехода к очередному элементу выходного массива нужно прибавить 4 к указателю на int,что и обеспечит смещение в 16 байт (размер OLE-variant).

Наверное, трюкачество с указателями – не самый лучший подход для управляемых языков. Впрочем, подобную технику можно встретить и в «промышленных» разработках, и даже в исходных текстах самого .Net Framework. А иначе зачем было вводить в язык возможности unsafe-кода?

Если же такие трюки кому-то не нравятся, можно описать структуру, соответствующую tagVARIANT из C++, и выполнить операции с указателем на эту структуру:

|  |
| --- |
| struct VARIANT  {  publicshort vt;  publicshort reserved1;  publicshort reserved2;  publicshort reserved3;  publicint data1;  publicint data2;  }  //...  VARIANT\* p = (VARIANT\*)rgvar;  for (int i = 0; i < celt; i++, p++)  if (enumerator.MoveNext())  Marshal.GetNativeVariantForObject(  enumerator.Current,  new IntPtr((void\*)p)); |

Впрочем, вводить структуру только для того, чтобы выполнять корректное перемещение указателя – это уже в некотором смысле излишество. Поэтому оставим вариант с «трюком», и тогда полный текст метода Next будет выглядеть следующим образом:

|  |
| --- |
| public  int Next(int celt, int rgvar, int pceltFetched)  {  IntPtr pceltFetchedPtr = new IntPtr(pceltFetched);  if (pceltFetchedPtr != IntPtr.Zero)  Marshal.WriteInt32(pceltFetchedPtr, 0);  if (celt <= 0)  return S\_FALSE;  else  {  unsafe  {  int\* p = (int\*)rgvar;  for (int i = 0; i < celt; i++)  {  if (enumerator.MoveNext())  Marshal.GetNativeVariantForObject(  enumerator.Current,  new IntPtr((void\*)p));  elsereturn S\_FALSE;  p += 4; // смещаем указатель на 16 байт вперед  }  }  if (pceltFetchedPtr != IntPtr.Zero)  Marshal.WriteInt32(pceltFetchedPtr, celt);  return S\_OK;  }  } |

С учетом же того, что Visual Basic, VBA, VBScript всегда считывают по одному элементу коллекции, исходный код метода можно существенно упростить:

|  |
| --- |
| public  int Next(int celt, int rgvar, int pceltFetched)  {  if (celt == 1 && enumerator.MoveNext())  {  Marshal.GetNativeVariantForObject(  enumerator.Current,  new IntPtr(rgvar));  return S\_OK;  }  elsereturn S\_FALSE;  } |

### Общая схема реализации коллекций

При реализации коллекций необходимо использовать рекомендации Microsoft по разработке COM-объектов в .Net, сформулированные в документации .Net Framework SDK. В частности, целесообразно использовать такие атрибуты, как GuidAttribute, ClassInterfaceAttribute, ComVisibleAttribute и пр.

Кроме того, необходимо использовать явное определение интерфейса коллекции, подобное следующему:

|  |
| --- |
| public  interface IxxxCollection  {  [DispId(1)]  int Count { get; }  [DispId(0)]  object Item([In] int index);  [DispId(-4)]  object \_NewEnum  {  [return: MarshalAs(UnmanagedType.IUnknown)]  get;  }  } |

Значения идентификаторов DispId членов Item и \_NewEnum важны.

Для метода Item идентификатор DispId должен иметь зарезервированное значение 0 (соответствует DISPID\_VALUE), что позволяет внешнему окружению использовать этот метод неявно (пример VBA):

|  |
| --- |
| x = collection.Items(0)  x = collection(0) |

Для свойства \_NewEnum идентификатор DispId должен иметь зарезервированное значение -4 (соответствует DISPID\_NEWENUM), что позволяет внешнему окружению использовать это свойство для получения нумератора. Кроме того, подчеркивание в начале названия свойства позволяет сделать это свойство «невидимым» для сред типа VB (в соответствии с рекомендациями Microsoft).

### Нетипизированная коллекция

В качестве объектов нетипизированной коллекции используем COM-объекты Loan из примера, поставляемого с .Net Framework SDK (см. проект <SDK>\Samples\Technologies\Interop\Applications\LoanApps\COMtoNET\loanlib).

Исходный код:

|  |
| --- |
| public  interface ILoanCollection  {  [DispId(1)]  int Count { get; }  [DispId(0)]  object Item([In] int index);  [DispId(-4)]  object \_NewEnum  {  [return: MarshalAs(UnmanagedType.IUnknown)]  get;  }  }  publicclass LoanCollection : ILoanCollection  {  public LoanCollection()  {  items = new Loan[2];  items[0] = new Loan();  items[0].OpeningBalance = 100;  items[1] = new Loan();  items[1].OpeningBalance = 200;  }  publicint Count  {  get { return items.Length; }  }  publicobject Item(int index)  {  try { return items[index - 1]; }  catch { returnnull; }  }  publicobject \_NewEnum  {  get { returnnew \_EnumVariant(items.GetEnumerator()); }  }  Loan[] items;  } |

### Типизированная коллекция

Типизированная коллекция реализуется аналогично нетипизированной (см. выше), за исключением того, что метод Item будет возвращать типизированное значение интерфейса ILoan:

|  |
| --- |
| public  interface ITypedLoanCollection  {  //...  [DispId(0)]  ILoan Item([In] int index);  //...  }  publicclass TypedLoanCollection : ITypedLoanCollection  {  //...public ILoan Item(int index)  {  try { return items[index - 1]; }  catch { returnnull; }  }  //...  Loan[] items;  } |

Фактически метод Item может возвращать и объект Loan. По крайней мере, VBA одинаково корректно работает в обоих случаях.

Кроме того, в соответствии с рекомендациями Microsoft по разработке коллекций в случае, если в метод передано значение индекса несуществующего элемента, метод Item должен возвращать значение null.

### Коллекция строк

Исходный код коллекции строк аналогичен коду типизированной коллекции:

|  |
| --- |
| public  interface IStringTestCollection  {  //...  [DispId(0)]  string Item([In] int index);  //...  }  publicclass StringTestCollection : IStringTestCollection  {  //...publicstring Item(int index)  {  try { return items[index - 1]; }  catch { returnnull; }  }  //...string[] items;  } |

Отметим, что в методе Item возможен возврат null, т.к. строки допускают значение null.

Иная ситуация с коллекциями, элементы которых имеют value-тип, например, целые числа.

### Value-type коллекции

Под термином «value-type коллекции» здесь понимаются коллекции, элементы которых имеют простой тип, совместимый с COM, – int, double, bool, DateTime и др. (о соответствии .Net-типов COM-типам в документации .Net Framework SDK написано достаточно подробно). К этой же категории можно отнести и коллекции, элементы которых являются структурами (структуры должны быть совместимы с COM).

Отличие value-type коллекций от коллекций объектов и строк заключается в том, что метод Item не может возвращать значение null. В случае же передачи в качестве параметра метода некорректного индекса можно сгенерировать соответствующее COM-исключение. Подобную технику, кстати, можно использовать и для коллекций объектов и строк.

Рассмотрим в качестве примера реализацию целочисленной коллекции:

|  |
| --- |
| public  interface IIntTestCollection  {  //...  [DispId(0)]  int Item([In] int index);  //...  }  publicclass IntTestCollection : IIntTestCollection  {  //...publicint Item(int index)  {  return items[index - 1];  }  //...int[] items;  } |

Отметим, что в методе Item в случае некорректного индекса сгенерируется .Net-исключение IndexOutOfRangeException, которое автоматически будет преобразовано в соответствующую COM-ошибку (80131508 – Index was outside the bounds of the array).

### Коллекция с несколькими типами индексации

До сих пор рассматривались коллекции с целочисленной индексацией элементов. В общем же случае индексом коллекции может выступать произвольное значение. Так, например, коллекция Documents COM-объекта Word.Application обеспечивает доступ к своим элементам двумя способами:

|  |
| --- |
| Documents(1).Activate  Documents("Report.doc").Activate |

Т.е. в качестве индексов элементов могут выступать как целые числа, так и строковые значения.

Построим подобную коллекцию:

|  |
| --- |
| public  interface IVariantIndexTestCollection  {  //...  [DispId(0)]  int Item([In] object index);  //...  }  publicclass VariantIndexTestCollection : IVariantIndexTestCollection  {  //...publicint Item(object index)  {  string s = index asstring;  if (s != null)  {  if (s == "один")  return items[0];  elseif (s == "два")  return items[1];  elsethrownew IndexOutOfRangeException();  }  else  {  try  {  return items[Convert.ToInt32(index) - 1];  }  catch (IndexOutOfRangeException e)  {  throw e;  }  catch  {  thrownew ArgumentException();  }  }  }  //...int[] items;  } |

Здесь необходимо обратить внимание на то, что параметр index метода Item имеет тип object, который соответствует типу variant в COM. И, конечно же, из-за необходимости проверки реального типа параметра реализация метода несколько усложняется.

И еще один момент – использование класса Convert позволяет унифицировать обработку разных целочисленных типов (byte, sbyte, short, ushort, int, uint).

## Заключение

Рассмотренные в настоящей статье коллекции, естественно, максимально упрощены. Тем не менее, можно надеяться, что описанные техники и приведенный в качестве примеров код помогут сделать разработку реальных COM-коллекций достаточно простой.

Полные исходные тексты прилагаются.

Исходные тексты C# реализованы как дополнение к проекту <SDK>\Samples\Technologies\Interop\Applications\LoanApps\COMtoNET\loanlib. Для сборки и регистрации тестовой библиотеки выполните следующее:

* перейдите в режим командной строки Windows и сделайте текущим указанный выше каталог;
* запустите утилиту nmake с параметром clean;
* сохраните копию файла makefile из каталога loanlib;
* скопируйте в каталог loanlib новый файл makefile и файлы \*.cs из прилагаемого архива;
* запустите утилиту nmake all.

Исходные тексты примеров использования коллекций находятся в файле Samples.bas. Тестирование выполнялось в Excel 2003 (файл Test.xls), и файл Samples.bas был экспортирован из VBA. Поэтому исходный код процедур необходимо скопировать в ту среду исполнения, в которой будут выполняться прилагаемые примеры. Кроме того, если в среде исполнения требуется указывать ссылки на внешние библиотеки COM-объектов, необходимо подключить библиотеку LoanLib.dll.

В завершение хотелось бы заметить, что в статье довольно мало исходного кода. И это, наверное, хорошо, когда такие сложные задачи, как создание коллекций, решаются так просто. И все это возможно благодаря тому, что .Net обеспечивает очень мощные и в то же время простые в использовании технологии для работы с COM.

*Эта статья опубликована в журнале RSDN Magazine #3-2005. Информацию о журнале можно*

### Работа с Excel из C#.

Столкнулся с необходимостью поработать с файлами Excel из C#. Задачи простые - считать данные из книги Excel, создать новую книгу Excel, внести в нее данные. Предполагается, что Excel на компьютере имеется, но какой версии, не известно. Файлы по размерам маленькие, скорость работы не критична.  
  
Существует несколько вариантов работы с Excel из C#: автоматизация Excel, подключение через OleDB/ODBC, дополнительные библиотеки ([Aspose Excel](http://www.aspose.com/categories/file-format-components/aspose.cells-for-.net-and-java/default.aspx)), работа через [XML](http://www.carlosag.net/Tools/ExcelXmlWriter/Default.aspx), через [Open XML](http://www.developerfusion.com/article/6170/read-and-write-open-xml-files-ms-office-2007/) и т.п.  
  
Наиболее простой вариант - воспользоваться автоматизацией Excel. Да, скорость работы - не блеск. Зато удобно использовать, код пишется быстро, объемы кода не велики. Из .NET автоматизация подключается парой кликов мыши - добавил в References сборку Microsoft.Office.Interop.Excel - и работай с привычными com-объектами Application,Workbook, Worksheet и т.п.  
  
Проблема одна - сборки "Microsoft.Office.Interop.Excel" для каждой версии Excel разные. У меня установлен office 2003 и, соответственно, interop-сборка версии 11. А что делать, если мне нужно разработать приложение, которое может работать с Excel 97? Или с **любыми** версиями Excel? Для того, чтобы считать или записать пару значений в ячейки листа Excel сложного api не требуется - любой Excel сгодится. А мне приходится привязываться к конкретной версии.  
  
Можно воспользоваться [поздним связыванием](http://www.gotdotnet.ru/LearnDotNet/Misc/156928.aspx). Но если его использовать "в лоб", то ни о никаком удобстве работы речи уже не идет - код станет сложным, а вызовы методов - косвенными, нетипизированными. Ошибок будет - вагон.  
  
Выход нашелся. На СodeProject обнаружил замечательную статью ["SafeCOMWrapper - Managed Disposable Strongly Typed safe wrapper to late bound COM"](http://www.codeproject.com/KB/COM/safecomwrapper.aspx). В ней изложена элегантная методика использования позднего связывания, устраняющая все проблемы: связывание становится поздним (привязки к конкретной версии Excel нет), для объектов автоматизации автоматически реализуется шаблон IDisposable (отработанные объекты уничтожаются автоматически), все вызовы методов явные.  
  
Идея реализации, вкратце, следующая. Для каждого COM-объекта автоматизации вы прописываете отдельный интерфейс. В интерфейс включаете все методы и свойства, которые вам необходимо использовать у этого COM-объекта. Обратите внимание - только те, которые *необходимо использовать*, а вовсе не все, реализуемые COM-объектом.   
  
Каждый COM-объект автоматизации "заворачивается" в класс COMWrapper, унаследованный от [RealProxy](http://msdn.microsoft.com/ru-ru/library/system.runtime.remoting.proxies.realproxy.aspx). RealProxy - это стандартный класс, позволяющий организовать перехват вызовов методов. При создании COM-объекта вы создаете экземпляр COMWrapper и указываете требуемый вам интерфейс. Среда *динамически* генерирует прокси-объект, реализующий этот интерфейс. С этим прокси-объектом вы в дальнейшем и работаете как с объектом автоматизации. Вызов любого метода прокси-объетка перехватывается и транслируется в вызов метода Invoke класса RealProxy, перекрытый в классе COMWrapper. Здесь его можно обработать как душе угодно. В реализации по умолчанию, вызовы свойств транслируются в вызовы соответствующих методов get\_ и set\_, создаваемых .NET, возвращаемые объекты автоматизации автоматически заворачиваются в COMWrapper и т.п.  
  
В оригинально статье приведен пример использования данной методики для Microsoft.Outlook. Я, на базе приведенных исходных кодов, адаптировал методику для работы с Microsoft.Excel. Естественно, реализовав интерфейсы только для тех объектов автоматизации, которые потребовались мне для работы. Дополнить интерфейсы недостающими методами или добавить интерфейсы для других объектов автоматизации не составляет труда. Достаточно посмотреть через Object Browser метаданные сборки "Microsoft.Office.Interop.Excel" и скопировать оттуда необходимые объявления методов и интерфейсов (с минимальной адаптацией).  
  
Пример кода для работы с Excel:

|  |  |  |
| --- | --- | --- |
| 01 | using (Application app = ExcelApplication.Create()) { | |
| 02 | String s = app.Version; |

|  |  |
| --- | --- |
| 03 | app.Visible = true; // make excel visible |
| 04 | using (Workbook wb = app.Workbooks.Add(Type.Missing)) { | |

|  |  |
| --- | --- |
| 05 | wb.Title = "new workbook"; |
| 06 | using (Worksheets worksheets = wb.Sheets) { | |

|  |  |
| --- | --- |
| 07 | using (Worksheet ws = worksheets[1]) { |
| 08 | //try to assign some values to some cells | |

|  |  |  |
| --- | --- | --- |
| 09 | using (Range cells = ws.Cells) { | |
| 10 | for (int i = 1; i < 10; ++i) { |

|  |  |  |
| --- | --- | --- |
| 11 | using (Range r = cells[i, i]) { | |
| 12 | r.Value = i \* i; |

|  |  |
| --- | --- |
| 13 | } |
| 14 | } |

|  |  |
| --- | --- |
| 15 | } |
| 16 | } |

|  |  |
| --- | --- |
| 17 | } |
| 18 | wb.Saved = true; | |

|  |  |  |
| --- | --- | --- |
| 19 | wb.Close(); | |
| 20 | } |

|  |  |  |
| --- | --- | --- |
| 21 | app.Quit(); | |
| 22 | } |

Оригинальные исходные коды лежат на codeproject вместе со [статьей](http://www.codeproject.com/KB/COM/safecomwrapper.aspx). Исходные коды, адаптированные для Microsoft Excel, можно взять [~~здесь~~](http://www.rammus.ru/downloads/20090520ManagedExcel).

* [Download source codes of Managed Excel project](http://code.google.com/p/dvsrc/downloads/detail?name=ManagedExcel.7z&can=2&q=)
* [View source codes of Managed Excel project](http://code.google.com/p/dvsrc/source/browse/trunk/CSharp/ManagedExcel)
* [Short description in English](http://code.google.com/p/dvsrc/wiki/CSharpManagedExcel)

**Update.** Практика использования выявила небольшую проблему - забыть вызвать Dispose у оберток объектов Excel очень легко. После чего Excel висит в памяти до тех пор, пока не закроешь приложение (или до следующей сборки мусора). Пример:

|  |  |  |
| --- | --- | --- |
| 1 | using (Workbook wb = app.Workbooks.Add(Type.Missing)) { | |
| 2 | using (Worksheet ws = wb.Sheets[1]) |

|  |  |  |
| --- | --- | --- |
| 3 | /\*Здесь потерян Dispose у объекта Worksheets, который создается при | |
| 4 | вызове wb.Sheets\*/ |

|  |  |
| --- | --- |
| 5 | { .... }} |

вместо

|  |  |  |
| --- | --- | --- |
| 1 | using (Workbook wb = app.Workbooks.Add(Type.Missing)) { | |
| 2 | using (Worksheets worksheets = wb.Sheets) { |

|  |  |  |
| --- | --- | --- |
| 3 | using (Worksheet ws = worksheets[1]) { | |
| 4 | .... }} |

В статье на codeproject в исходниках автор включает файлы:  
ComEventProvider.cs  
ComEventsAttribute.cs  
ComEventSink.cs  
ComProgIdAttribute.cs  
  
Вопрос: это сгенерированные файлы (типа AssemblyInfo.cs) или написаны вручную. Если вручную, то основываясь на чем. Тем более, что в твоей версии, как я понял, они не используются

Пытаюсь ипользовать этот подход в своей программе. Все работает, кром того что если считывать свойства объекта в цикле, допустим пробегать по ячейкам листа, то появляется исключение. Вот код:  
  
Workbook oBook;  
Sheets xlsheets = oBook.sheets;  
Worksheet curWS = xlsheets[2];  
  
for (int i = 0; i < Count; i++)  
{  
Range curRange = curWS.Cells[4 + i, counter];  
  
// при i > 80 вылетает исключение. i примерное, не точное число, иногда до 116 дотягивает, у меня в логике нет проблем)   
  
curRange.Value2 = (object)sendValue;  
curRange = null;  
}  
  
Исключение: COM object that has been separated from its underlying RCW can not be used  
  
Чтобы точно убедиться, что ошибка не в логике, нужно сделать отдельное маленькое тестовое приложение. С одной функцией - считывание свойств объектов в цикле. Попробуй. Если ошибка воспроизведется - выкладывай текст функции, я постараюсь помочь с поиском ошибки.  
  
P.s. Имхо все же лучше писать так:  
using (Range curRange ... ) {  
...  
curRange.Value2 = (object)sendValue;  
/\*сurRange = null; \*/  
}  
  
Если просто присваивать curRange = null, то вызов Dipsose у curRange будет откладываться до сборки мусора на "неизвестно когда".

Спасибо большое за совет. Действительно using помог. Хотя не очень понимаю, как это влиять может. Видимо тонокости неподвластные пониманию :)  
  
Но вот вопрос, если у меня есть класс, а в нем два метода которые вызываются в разное время, в одной создается например Sheet, а в другой через какое-то время закрывается или производятся манипуляции с яйками. То как в этом случе using использовать? Его же на два метода не растянешь.  
  
У меня просто есть wrap класс позволяющий манипулироватьс экзелем. Так чтобы пользователи класса не знали что там внутри вообще происхдит. Типа метод "Создать лист", или "создать лист и згрузить в него некую структур данных" и т.д. Я хочу сказать о том что за пределы класа объявление книг и листов не получится вынести.  
  
Такие дела. Спасибо большое еще раз.

Код:  
  
using (A a = new A()) {  
...  
}  
  
по большому счету, эквивалентен  
  
A a = new A(); //создание объекта Excel  
try {  
...  
} finally {  
a.Dispose(); //уничтожение объекта Excel  
}  
  
Так что можно обойтись без using - в одном методе вызывать A a = new A(), а в другом a.Dispose().   
  
Все классы объектов Excel, используемые в описываемом подходе, реализуют интерфейс [IDisposable](http://msdn.microsoft.com/ru-ru/library/system.idisposable.dispose.aspx). Для уничтожения экземпляров таких объектов нужно явно вызывать у них метод Dispose. Если этого не сделать, то Dispose будет вызван автоматически, при сборке мусора. Но когда это произойдет - не известно. А до той поры все объекты будут накапливаться в памяти неуничтоженными. И если их максимально возможное количество ограничено - полезут ошибки.  
  
P.s. на rsdn есть [хорошая статья на тему IDisposable и using](http://www.rsdn.ru/article/dotnet/GCnet.xml). После ее прочтения вся эта суета с Dispose станет понятнее :)

Тут еще вопрос сопутствующий, если я вызываю  
a.Dispose(); но мне по какой-то причине нужно чтобы объект превратился в null. Могу ли я записать так:  
  
object a = new a();  
...  
  
a.Dispose();  
a.Null;  
  
Прочитал статью которую вы рекомендовали. Некторые вещи стали теперь понятнее. Спасибо за сслыку. Не могли бы вы посоветовать программу или способ поискать утечки памяти. Спасибо.

Конечно, объект можно "занулить" стандартным способом:  
a.Dispose();  
a = null;   
Если ссылка на объект "a" у нас была единственной, то после a = null объект станет неиспользуемым и вся занимаемая им (управляемая) память будет освобождена при сборке мусора.  
  
Могу посоветовать посмотреть в сторону утилит   
[Redgate ANTS Memory Profiler](http://www.red-gate.com/products/dotnet-development/ants-memory-profiler/) и [ProfileSharp](http://csharp-source.net/open-source/profilers/profilesharp). Первая коммерческая, вторая бесплатная.

Просто чисто логически не очень понятно. Я так понял что Dispose и должен был сразу дaвать сигнал на чистку памяти этого объекта, и когда мы пишем a=null, то не противоречит ли это друг другу. Т.е. как можно поставить флаг (=null) чтобы GC очистил занмаимую им память, если мы уже вызвали a.Dispose() для той же самой цели, но и еще более аккуратно, судя из того чтоя понял, при этом очистка будет выполнена как раз в момент вызова метода Dispose, а не когда вздумается, как это было бы с чистым a=null.

Вызовы a.Dipose() и a = null служат для разных целей.  
  
Вызов Dispose нужен для очистки неуправляемых ресурсов. При этом сам объект "a" остается живым и здоровым, мы все так же имеем на него ссылку в коде и можем вызывать его методы. Но все неуправляемые ресурсы в объекте "a" после вызова Dipose уничтожены.  
  
Выражение a = null говорит о том, что ссылка на a нам больше не нужна.. а значит, объект надо пометить как неиспользуемый. Все управляемые ресурсы, занятые объектом, будут уничтожены при сборке мусора.  
  
Пример. Пусть у нас есть класс A, содержащий как управляемый, так неуправляемый объекты:  
  
class A {  
//управляемый объект  
public String m\_S;   
//неуправляемый ресурс  
private UnmanagedObject m\_U;   
public A() {  
//создаем объект в неуправляемой памяти  
allocate\_unmanaged(m\_U);   
}  
  
void Dipose() {  
//уничтожаем объект в неуправляемой памяти  
deallocate\_unmanaged(m\_U);   
}  
}  
  
Создаем экземпляр:  
A a = new A();  
  
Уничтожаем объект:  
a.Dispose();   
//m\_U уничтожен  
  
a = null;   
//a и m\_S помечены как более неиспользуемые; память, занимаемая ими, будет освобождена при сборке мусора.  
  
Только класс A знает, как создавать неуправляемый объект. И только он знает, как его уничтожать. C# самостоятельно сделать это не может, его надо научить.. Поэтому придумали IDisposable.   
  
Интерфейс IDisposable - это соглашение. Соглашение о том, что в классе A будет реализована функция Dispose, которая корректно уничтожает все неуправляемые ресурсы, выделенные в этом классе. В нашем примере функция Dipose должна содержать вызов deallocate\_unmanaged, обратный вызову allocate\_unmanaged.  
  
Однако реализация IDisposable несколько нетривиальна (см. статью на RSDN). Нетривиальность связана с вопросом: кто и когда должен вызывать Dispose?  
  
По идее, ее должен вызвать программист, когда уничтожает объект. Но он может забыть это сделать и утечки останутся. Поэтому в класс добавляют финализатор и вызывают Dispose в финализаторе. Финализаторы автоматически вызываются во время сборки мусора. Так что если программист забыл вызвать Dispose, то Dispose будет вызвана во время сборки мусора.   
  
Но здесь другая проблема. Объекты с финализаторами обрабатываются сборщиком мусора не так эффективно, как объекты без финализаторов. Добавляются накладные расходы.. В том случае, если программист культурный и Dipose вызывать не забывает (например, использует using), таких накладных расходов нужно суметь избежать.   
  
В результате, корректная реализация IDisposable включает две функции Dispose, финализатор и вызов GC.SuppressFinalize в финализаторе.  
  
Надеюсь, стало понятнее. Вот [здесь](http://stackoverflow.com/questions/538060/proper-use-of-the-idisposable-interface) все подробно изложено in english.

Только я немного запутался в определениях. Автоматизация Excel это вот это: http://ru.wikipedia.org/wiki/Microsoft\_OLE\_Automation ?

Есть задача: открыть книгу, подгрузить в неё vba-скрипт из внешнего файла и запустить из него определённую функцию на выполнение. Сейчас пытаюсь добавить в интерфейс Application доступ к скриптам. Где можно почитать или может кто уже расширял этот код для работы со скриптами?

Нашёл, что загрузку vba-скрипта из файла можно провести так: ActiveWorkBook.VBProject.VBComponents.Import(lcFile), Осталось сообразить как определить VBProject, сидящий аж в Microsoft.Vbe.Interop.VBProject...

Мне нужно работать с чартами в Excel. Так что я стал добавлять нужные мне сущности в качестве оберток из Object Explorer. Добавил в интерфейс Worksheet метод ChartObjects ChartObjects(object Index); А в ChartObjects добавил object Item(object Index); Сам интерфейс ChartObject тоже добавил.   
  
Теперь хочу получить ссылку на какой-нибудь чарт. Далее вызываю все как положено:  
  
ManagedExcel.ChartObject chartObject = (ManagedExcel.ChartObject)charts.Item(ChartName);  
  
Тут и сваливается исключение: Unable to cast COM object of type 'System.\_\_ComObject' to interface type 'ManagedExcel.ChartObject'...(Exception from HRESULT: 0x80004002 (E\_NOINTERFACE)  
  
ChartName - это строковая переменная, имя чартра. Если заменить на int'овый индекс, то опять исключение. Все остальные интерфейсы подцепились без проблем. Может кто сталкивался с этим?

Трудно что-либо сказать, не видя код целиком (например, не ясно как инициализируется charts). Но вот что заметил. В MSDN метод Worksheet.ChartObjects описывается так:  
  
public virtual Object ChartObjects (  
[OptionalAttribute] Object Index  
)  
  
Return Value: an object that represents either a single embedded chart (a Microsoft.Office.Interop.Excel.ChartObject) or a collection of all the embedded charts (a Microsoft.Office.Interop.Excel.ChartObjects) on the worksheet.  
  
http://msdn.microsoft.com/en-us/library/microsoft.office.tools.excel.worksheet.chartobjects(v=vs.80).aspx  
  
Т.е. Worksheet.ChartObjects может возвращать как ChartObjects, так и CharObject. Я бы попробовал определить в Worksheet функцию Object ChartObjects(object Index); и вызвать ее так:  
  
ManagedExcel.ChartObject chartObject = (ManagedExcel.ChartObject)worksheet.ChartObjects(ChartName);

Привет!  
Да, моя вина, забыл указать, как инициализирую переменную charts:  
  
ManagedExcel.ChartObjects charts = (ManagedExcel.ChartObjects)worksheet.ChartObjects(Type.Missing);  
  
Ваш совет о приведении сразу к ManagedExcel.ChartObject не помог =( Ошибка все та же.  
  
Зато обнаружил следующее. Даже если worksheet типа ManagedExcel.Worksheet (т.е. объявлен и описан нами), то метод ChartObjects(object index) не хочет возвращать тип ManagedExcel.ChartObject, зато прекрасно кастится к обычному Excel.ChartObject из набора типов VSTO (Microsoft.Office.Interop.Excel)

Есть вопрос: подскажите, где отыскать описание всех интефейсов - например Range для диапазона ячеек; в MSDN найти не получается, видимо искать не умею :((

Самая полная справка конечно в MSDN  
http://msdn.microsoft.com/en-us/library/microsoft.office.interop.excel.range\_members.aspx  
  
Можно еще из Excel вызвать редактор Visual Basic и воспользоваться его справкой. Порой ее оказывается достаточно.

public interface Range : Common {  
Range this[Object rowIndex, Object columnIndex] { get; set; }  
int Row { get; }  
int Column { get; }  
Range Columns { get; }  
Range Rows { get; }  
String Text { get; }  
  
object Value { get; set; }  
object Value2 { get; set; }  
  
В качестве параметров вроде должен получать Object (диапазон ячеек), но передать удаётся только Int (одна ячейка). Пробовал и по простому ("A1","B3") и ячейки передавать, в первом случае ошибка всегда, во втором отрабатывает, если только ячейки содержат числовые значения, но опять получаю одну ячейку.  
P.S. Прошу простить невежество, но раньше использовал Automation

Добавлять классы и методы научился, к примеру:  
1. в EcxelWrappers.cs добавляем в Range такие строки:  
Interior Interior { get; }  
Font Font { get; }  
  
2. описываем их чуть ниже:  
public interface Interior : Common  
{  
Interior Interior { get; }  
Object Color { get; set; }  
}  
  
public interface Font : Common  
{  
Font Font { get; }  
Object Bold { get; set; }  
}  
  
3. далее обрабатываем ячейки в виде цикла:  
using (Worksheet ws = worksheets[1]) {  
//try to assign some values to some cells  
using (Range cells = ws.Cells) {  
for (int i = 1; i < 10; ++i) {  
using (Range r = cells[1, i]) {  
r.Value = i;  
r.Interior.Color = Color.Red;  
r.Font.Bold = true;  
}  
}  
}  
}  
  
Но это чертовски не удобно, по сравнению с привычным методом в VBA. Пробовал описать в таком варианте:  
Range this[Object Cell1, Object Cell2] { get; }   
увы не получилось =(

Здравствуйте Виктор. Скажите, подход, который Вы описываете в данной статье имеет сейчас актуальную ценность? Или сейчас есть более современный подход для работы с Excel из .NET? Просто у нас в компании имеется проект в котором также используем позднее связывание для работы с Excel, где потом следим за выгрузкой не используемых ресурсов, но как практика показала не удается все правильно выгружать и COM объекты накапливаются в памяти, что приводит к ошибкам и в дальнейшем к падению виндовой службы. Вот хочу у Вас спросить совета, что использовать для работы с Excel сейчас без привязки к оперделенной версии Excel и без слежения за неиспользуемыми объектами COM и дальнейшей очистки?

Раньше, приложения, разрабатываемые для работы с офисом, вынуждены были ссылаться на interop-сборки (например "Microsoft.Office.Interop.Excel") и эти сборки распространялись вместе с приложением. Поскольку сборки для разных версий Excel разные, возникали проблемы.   
  
В Net 4.0 появились "встраивание метаданных взаимодействия" (embed interop types) и "эквивалентность типов" (type equivalence).   
  
"Embed interop type" позволяет включать в приложение метаданные из интероп-сборки, причем только те, что реально используются приложением. Таскать интероп-сборку вместе с приложением больше не требуется.   
  
Если опция включена, то вместо   
  
var range = (Excel.Range) excel.Cells [1, 1];  
range.Font.FontStyle = "Bold";  
  
можно сразу писать такой код  
  
var range2 = excel.Cells [1, 1];  
range.Font.FontStyle = "Bold";  
  
Важный момент: range2 при этом имеет тип dynamic, а не object. Т.е. фактический тип range2 определяется во время выполнения. IntelliSense для range2 работать не будет, но это легко исправить -   
сразу (динамически) привести к нужному типу  
  
Excel.Range range2 = excel.Cells [1, 1];  
range.Font.FontStyle = "Bold";  
  
(пример взят из отличной книжки Albahari J., Albahari B. C# 5.0 in a Nutshell, в ней есть небольшой раздел на эту тему)  
  
Эквивалентность типов означает, что в период выполнения два разных типа могут считаться эквивалентными и взаимозаменяемыми. В частности, при подключени interop-сборок компилятор считает COM-типы эквивалентными при условии, что у них один и тот же GUID. Соответственно, приложение сможет корректно работать с той версией интероп-сборки, которая есть на машине пользователя - при условии, что в ней присутствуют реализации всех необходимых приложению COM-типов.  
  
Как это все использовать на практике подробно написано здесь  
http://www.claudiobernasconi.ch/2014/02/13/painless-office-interop-using-visual-c-sharp/  
  
Спасибо Вам за подробное описание и направление на новый подход. Буду пробовать и исследовать на практике, т.к., мы думали использовать среду DLR, но про подход эквивалентности типов и встраивание метеданных взаимодействия я не слышал. Спасибо за полезную информацию и ссылки. По поводу того как буду пробовать данный подход Вам отпишусь про выгрузку COM-объектов. Уж очень надо решить проблему с полной выгрузкой всех объектов, чтобы предотвратить накопление и соответственно падение службы.   
  
Просто этот проект у нас в компании старенький висит на .NET Framework 2.0 еще и использует позднее связывание с описанием интерфейсов для MS Excel без привязки к определенной версии Microsoft.Interop.Excel. Есть виндовая служба которая создет пул из трех процессов экземпляра Excel, каждый процесс подымается в своем домене. В дальнейшем данные процессы Excel используеются для генерации отчетов с сайта и через виндовое пиложение. Служба следит за тем корректно ли создаются отчеты и если произошел какой-то сбой, а он происходит обычно с Excel процессами, то служба выгружает данный домен с опеределенным процессом и подымает новый, но все равно где копится видно COM-объекты и не все подчищается. Но может конечно еще дело и в том что в Excel используется надстройка с помощью которой идет связь с базой данных PI через PI SDK откуда вытягиваются данные в шаблоны Excel. Может еще в этом проблема. Так как у нас в службе настроено логирование и проблему все равно не удается опеределить, т.к в логах ошибок не возникает, служба работает работает и вдруг бах остановилась. В виндовых логах кроме как вот этих ошибок ничего конкретного нет:  
  
Ошибка вязанная с PI от которого поступают данные  
  
The SDK Yield thread did not stop when signaled during the last Release of the SDKSessionMgr and was terminated.  
  
и вторая непосредственно нашей службы:  
Имя сбойного приложения: PIRSServerHost.exe, версия: 3.0.21.36502, отметка времени: 0x55965170  
Имя сбойного модуля: ntdll.dll, версия: 6.1.7601.17725, отметка времени 0x4ec49b8f  
Код исключения: 0xc0000005  
Смещение ошибки: 0x00038dc9  
Идентификатор сбойного процесса: 0x66428  
Время запуска сбойного приложения: 0x01d0de0ac41e843f  
Путь сбойного приложения: C:\Program Files (x86)\Indusoft\Сервер отчетов\PIRSServerHost.exe  
Путь сбойного модуля: C:\Windows\SysWOW64\ntdll.dll  
Код отчета: 444c80e6-4a0b-11e5-9ec9-005056ba014f  
  
Но ничего конкретного определить из данных ошибок не удается.   
Так что буду пробовать Ваш совет.

явно использовать dynamic нет необходимости. Можно написать так  
  
dynamic range = excel.Cells [1, 1];  
  
но гораздо удобнее сразу  
  
Excel.Range range = excel.Cells [1, 1];  
  
При этом "внутри" dynamic используется неявно, как-то так:  
dynamic j = excel.Cells [1, 1];  
Excel.Range range = (Excel.Range)j;

И поправьте меня, если я не так понял, что то из ссылки на статью. Правильно я полагаю, что если мое приложение имеет необходимость поддержки Office 2003 - Office 2013 у клиента, то сборке приложения я должен хранить ссылку на офис 2003, чтобы поддерживать все остальные более высокие версии, так? И если у клиента вдруг будет на машине будет стоять Office 97, то мое приложение будет работать с ошибками или вообще не будет работать, так.

Имхо вся эта кухня работает следующим образом. Рассмотрим пример. Пусть у нас есть три офиса: 2003, 2010, 2013. В них реализованы фичи:  
  
2003 - F1  
2010 - F1, F2  
2013 - F1, F2, F3  
  
Вы собираете приложение на основе интероп сборки для офиса 2010. Вопрос: будет ли работать приложение с 2003 и 2013?  
  
В приложение вкомпилены метаданные для интеропсборки от десятого офиса (за счет "Embed interop type = true"). Поэтому приложение с любым установленным офисом работает как с десятым. Таким образом фича F3 приложению автоматически недоступна - для нее нет метаданных.  
  
Фича F1 будет работать в любом офисе начиная с 2003.  
Фича F2 будет работать только в том случае, если на машине пользователя установлен десятый офис или выше. Если же там 2003 офис, то при попытке использовать фичу F2 в рантайме вылетит ошибка.  
  
Так что если требуется нужна поддержка 2003-офиса, можно использовать интеропсборку от любого офиса - но при этом работать только с теми функциями, которые гарантированно есть в 2003 офисе.  
  
Я написал коротенькое тестовое приложение, чтобы проверить подход. Вот оно  
https://www.dropbox.com/s/aosprjypqk46v2o/TestExcel.7z?dl=0  
  
Приложение тривиальное. При нажатии на кнопку загружает из указанного файла все строки из UsedRange и показывает их в TextBox.  
  
Я собирал его на машине, на которой установлен старый Office 2003. В references добавил ссылку на Microsoft.Office.Interop.Excel версии 14 (т.е. от офиса 2010).  
  
С офисом 2003 работает без проблем. С десяткой тоже. На машине без офиса запускается - но при нажатии на кнопку загрузки данных - падает. Все как и должно быть.  
  
Проверил жрет ли память. Проблема с памятью никуда не делась. Чтобы эксели не зависали в списке процессов после того, как работа с ним закончена, следует: 1) не допускать неявного использования COM-объектов 2) явно релизить com-объекты после того, как они стали не нужны.  
  
Т.е. нельзя написать  
  
app.Workbooks.Open(workbookFullName)  
  
вместо этого следует написать так  
  
Workbooks workbooks = app.Workbooks;  
Workbook book = workbooks.Open(workbookFullName);  
  
и затем явно зарелизить объекты  
  
Marshal.FinalReleaseComObject(book);  
Marshal.FinalReleaseComObject(workbooks);  
  
а в конце и объект приложения  
app.Quit();  
Marshal.FinalReleaseComObject(app);  
  
Рекомендую посмотреть описание Marshal.FinalReleaseComObject в msdn, там подробнее написано.

Здраствуйте! Подскажите пожалуйста, а как можно поступить в такой вот интересной ситуации:  
Если на ПК установлен Microsoft Office то данный подход будет работать... Но! Есть такой нюанс: пробовал сохранить в Ексель-файл на ПК, где Microsoft Office не установлен, либо стоит альтернативный офис типа Либры или OpenOffice... И такая конструкция уже не срабатывает!(( Выдает ошибки, мол COM не зарегистрирован в системе и всё такое...   
Отсюда у меня вопрос: как можно реализовать сохранение/открытие в/из файлов Ексель на форму программы, без вызова СОМ? Ну или с вызовом, если без этого никак, но как тогда для случаев, когда конкретно майкрософтовский Excel не установлен на ПК, или вообще никакой не установлен, а из моей программы нужно просто выполнить сохранение данных в екселевский файл, как в таком случае реализовать сохранение/чтение в/из вайла эксель-форматов?

Можно попробовать задействовать одну из библиотек, способных считывать/создавать Excel-файлы без установленного Excel.  
  
Например, NPOI и [EPPlus](http://epplus.codeplex.com). Первая поддерживает форматы xls и xlsx, вторая - только xlsx.  
  
Если требуется только считывать данные, то есть еще интересная библиотека [ExcelDataReader](https://github.com/ExcelDataReader/ExcelDataReader).

[Работа с Excel из C#.](http://derevyanko.blogspot.com/2009/05/excel-c.html)