Создание надстроек для Autodesk Inventor

2015 г.

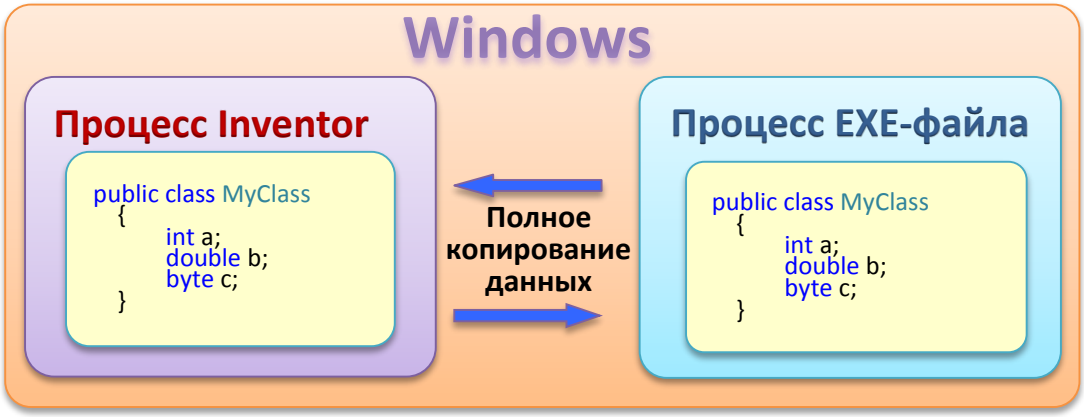
Оглавление

Почему DLL?

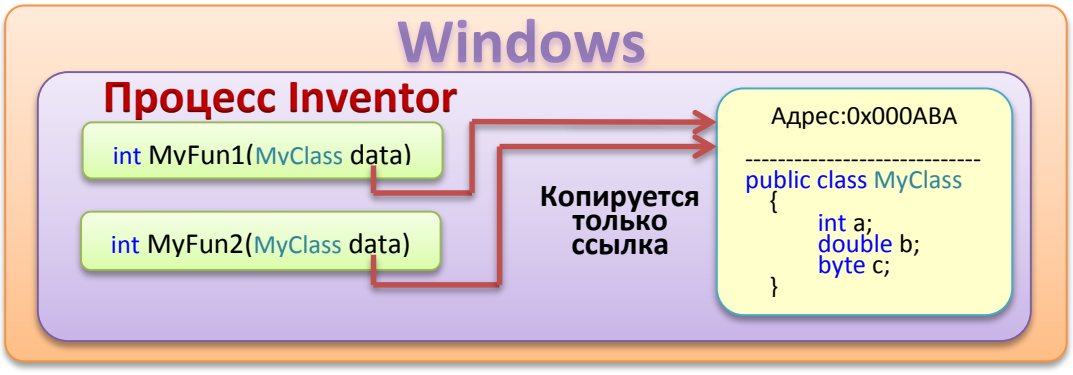
При решении прикладных задач, которые выходят за компетенцию макросов на VBA, приходится использовать профессиональные языки программирования. При использовании этих языков, Inventor позволяет выбрать два способа компиляции в исполняемый EXE-файл и DLL-файл (AddIn) динамически линкуемый при запуске Inventor.

На первый взгляд с EXE-файлом работать проще в плане отладки. Хотя на самом деле работа и отладка DLL-файла ненамного сложнее, чем с EXE-файлом. Просто отладка DLL-файла не достаточно освещена в хэлпе API Inventor. В свое время я потратил не один час, пытаясь запустить DLL от Inventor AddIn в отладочном режиме. Поэтому было решено осветить эту тему достаточно подробно, что бы по возможности не осталось белых пятен.

Перед тем как продолжить вкратце разберем, по какой причине Inventor AddIn работают гораздо быстрее, чем EXE-файлы, которые запущенны отдельным процессом. Процессы являются изолированными друг от друга адресными пространствами, поэтому нельзя передать данные из одного процесса в другой, используя только указатель (байтовый адрес в адресном пространстве). Поэтому Windows полностью копирует данные из одного адресного пространство в другое адресное пространство, какими бы объемными эти данные не были бы, в данном случае данные, процесса Inventor. В итоге получаются большие накладные расходы производительности компьютера. Для наглядности поясню на схеме:



В случае же с Inventor AddIn для передачи информации, не обязательно копировать весь объект, можно передать только указывается адрес, где этот объект находится в адресном пространстве. Т.е. передается только копия одного числа (указатель или ссылка на объект) который содержит номер адреса в памяти (смещение):



Не трудно догадаться, что сделать копию одного числа гораздо быстрее, чем производить копирование всего объекта.

По этой причине программный код из DLL-файлов всегда работает гораздо быстрее, чем программный код из соседнего процесса.

К тому же существует еще один не приятный момент, работы EXE-файлов применительно к Inventor. Как показала практика, объект ***IPictureDisp***, содержащий интерфейс картинки для иконки кнопки, невозможно передать из соседнего EXE-процесса. При попытке это сделать будет генерироваться ошибка и кнопку с иконкой создать будет невозможно. Создать кнопку можно будет только с текстовым названием. При работе VBA такой проблемы не возникает т.к. VBA работает в одном адресном пространстве с Inventor. Аналогично нет проблем при работе DLL-файла в одном адресном пространстве с Inventor.

Так же важное отличие AddIn от EXE-файла это возможность автоматической загрузки надстройки и автоматической настройки интерфейса (панели, кнопки и пр.) для своих инструментов, при загрузке Inventor. Т.е. AddIn полноценно интегрируется в Inventor. Все программы будут написаны на полноценных версиях Visual Studio (не Express). ***Однако проекты, созданные на Visual Studio, без проблем откроются и будут работать на Visual Studio Express, в том числе и все готовые примеры.*** В Visual Studio Express отсутствие некоторых возможностей, для запуска AddIn на отладку, можно будет легко компенсировать. Как это делается обязательно рассмотрим.

VB.NET, C# или C++/CLI?

Теперь выберем язык программирования. На чем же лучше всего остановится? Действительно, однозначного ответа на этот вопрос нет, и я как автор, не имею желания устраивать на эту тему «холи-вар». Однако некоторые особенности выбора языка программирования, я сейчас объясню, основываясь на собственном опыте освоения:

VB.NET достоинства:

- Хэлп Inventor API написан на VBA, родственном для VB.NET языке программирования.
- Читабельность грамматики написанного кода VB.NET немного выше, чем у остальных Си-подобных языков.

VB.NET недостатки:

- Удобство использования VB.NET исчезает при попытках использования различных «неуправляемых» библиотек, на которых работает Windows. Хотя встроенные возможности маршализации решают некоторые проблемы в этом плане, но, как часто бывает, то, что нужно в данный момент во Framework отсутствует и приходится делать вызов «неуправляемых» функций, что очень неудобно.
- VB.NET иногда оказывает «медвежьи услуги», особенно начинающим, т.к. обычно проверка приведения типов обычно отключена, а это может повлечь ошибки во в время выполнения программы.
- Грамматика VB.NET мешает быстрому освоению Си-подобных языков.

C# достоинства:

- C# это Си-подобный язык, и после его изучения легко переходить на другие Си-подобные языки типа Java, JavaScript и C++.
- C# строго типизированный, это означает, что потери точности вычислений будут возможны только с разрешения программиста.
- C# имеет немного больше возможности, чем VB.NET, в плане использования «неуправляемой» быстрой арифметики.

C# недостатки:

- Аналогично VB.NET, удобство использования C# исчезает при попытках использования различных «неуправляемых» библиотек, на которых работает Windows.
- В хэлп Inventor API почти нет примеров на C#
- Читабельность грамматики кода немного ниже, чем у VB.NET

C++/CLI достоинства:

- Чистый C++ производительнее NET языков и может легко работать со всеми «неуправляемыми» библиотеками Windows, а также C++/CLI позволяет легко комбинировать «управляемый» и «неуправляемый» код вместе, даже, без маршализации.
- Чистый C++ дает представление, как, на самом деле, все работает внутри Windows.

C++/CLI недостатки:

- При чистом «нативном» C++ программа получается более громоздкая, в отличии от NET-языков
- На «управляемый» C++/CLI в Inventor API нет ни примеров, ни шаблонов.
- На чистый C++ нет примеров в хэлпе Inventor API, только есть примеры в Inventor SDK.

Раз уж речь в дальнейшем пойдет о создании AddIn, то подразумевается, что начинающий программист уже хочет выйти за рамки написания простых макросов и хочет перейти к написанию более серьезных программам. Т.к. языки программирования VB.NET и C# достаточно распространены и являются родственными, то примеры будем разбирать, используя оба этих языка. Но я все же рекомендовал бы свой остановить выбор на языке C#, который сохраняет в себе «простоту» NET языка и позволяет привыкнуть к синтаксису Си-подобных языков, что даст плюсы при дальнейшем развитии профессиональных навыков программиста.

Шаблоны и примеры для API Inventor

Для получения готовых шаблонов необходимо произвести установку SDK. Установочные файлы SDK обычно находятся в папке (для Inventor 2015):

C:\Users\Public\Documents\Autodesk\Inventor 2015\SDK

В этой папке можно увидеть два файла: ***developertools.msi*** и ***usertools.msi***.

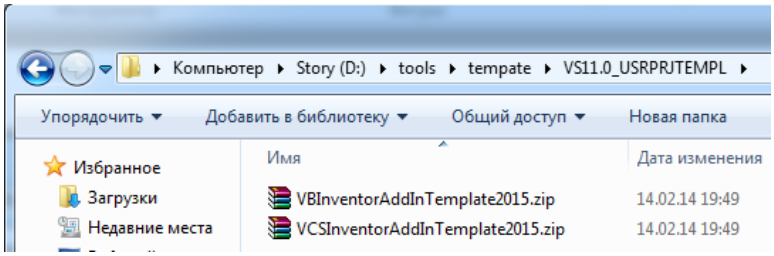
developertools.msi – устанавливает в Visual Studio шаблоны для создания AddIn, также содержит примеры на различных языках программирования, заголовочные файлы для «нативного» C++, схему объектной модели Inventor и пр.

usertools.msi – содержит дополнительные примеры для программирования.

Бывает ситуация, что SDK Inventor не адаптирован под версию Visual Studio и шаблоны для AddIn в Visual Studio не появляются. Что делать в этой ситуации? Можно в ручную распаковать MSI-файл и получить нужные шаблоны, для этого делаем следующие шаги:

- Копируем файл developertools.msi куда-нибудь в папку на диск D
- вызываем консоль с правами администратора в Windows и вызываем следующую команду (пути указываем без кавычек):
  msiexec /a путь к MSI-файлу /qb TARGETDIR=путь в папку извлечения

- после чего в извлеченном архиве находим архивы с соответствующими шаблонами



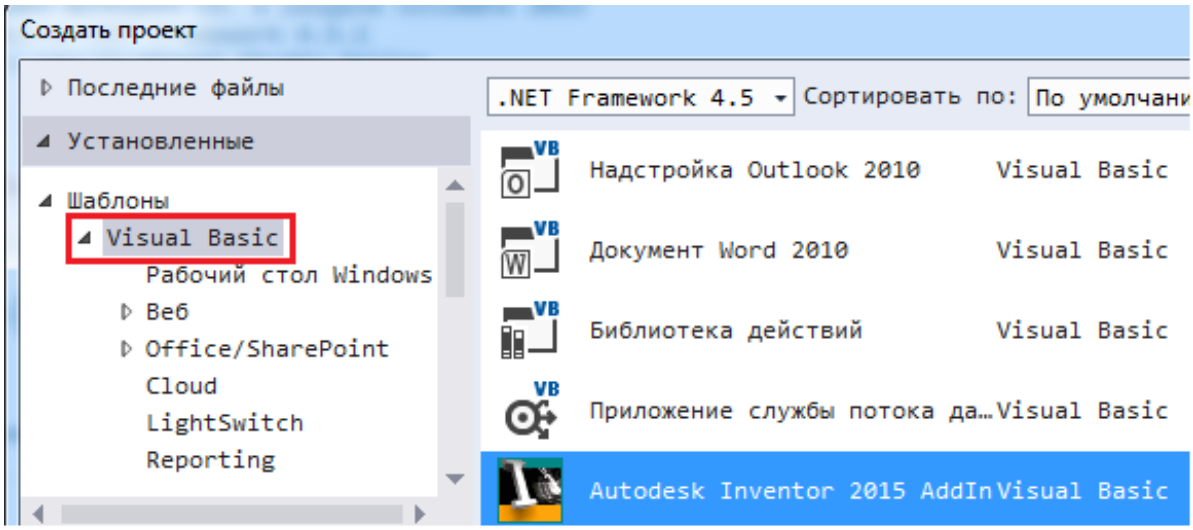
- Копируем файлы в соответствующие папки Visual Studio, в настройках по умолчанию путь имеет вид:

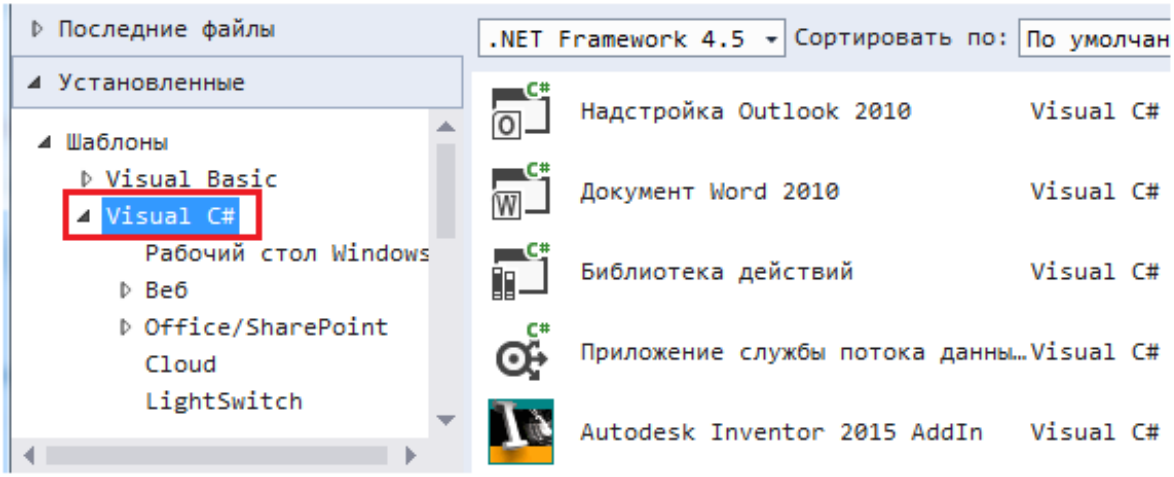
  C:\Users\{USER NAME}\Documents\Visual Studio 2013\Templates\ProjectTemplates\Visual C#

  и

  C:\Users\{USER NAME}\Documents\Visual Studio 2013\Templates\ProjectTemplates\Visual Basic

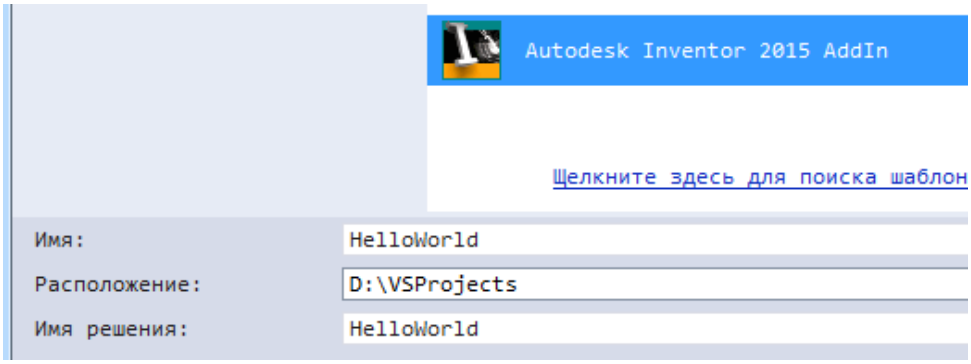
- Запускаем Visual Studio, шаблоны должны быть в доступе, что бы их увидеть кликните на узел соответствующего языка программирования





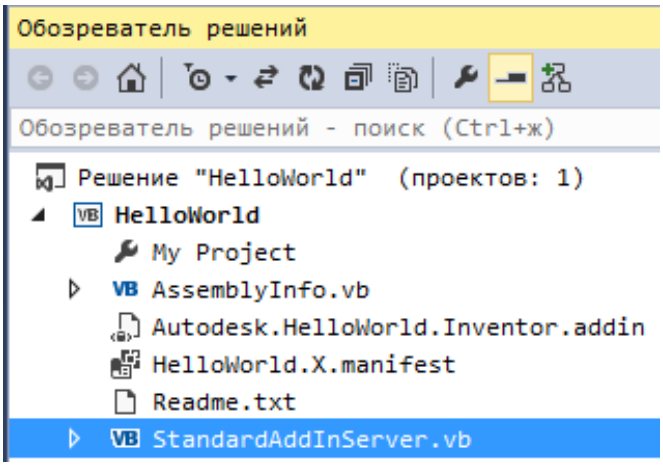
Пример «Здравствуй мир!» с использованием готового шаблона

Воспользуемся готовым шаблоном, чтобы создать пробный AddIn. Для этого запустим создание шаблона на VB.NET или C# и назовем проект «HelloWorld»:

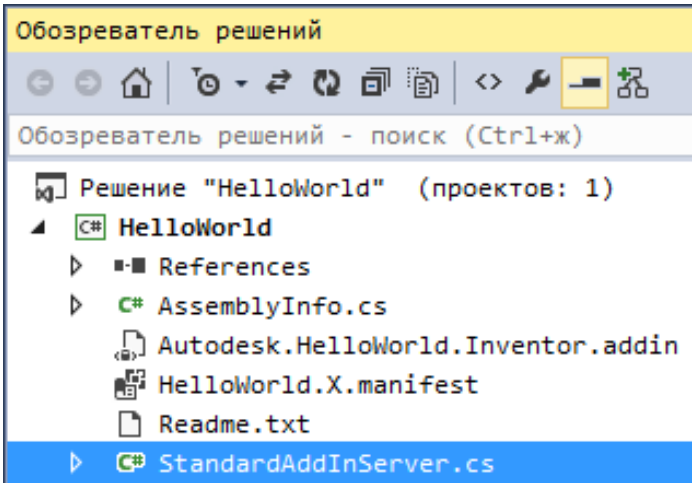


В ***«Обозревателе решений»*** Visual Studio откроем на редактирование файл, в котором содержится базовый программный код

для VB.NET это фал ***StandardAddInServer.vb***



для C# это файл ***StandardAddInServer.cs***



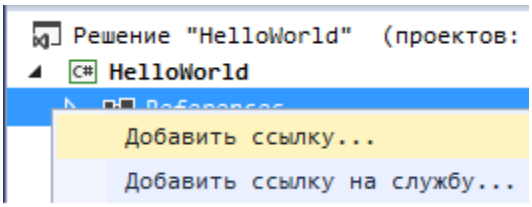
Ищем, в открывшемся программном коде процедуру с именем ***Activate*** и вставляем в неё запуск окна приветствия:

Для VB.NET:

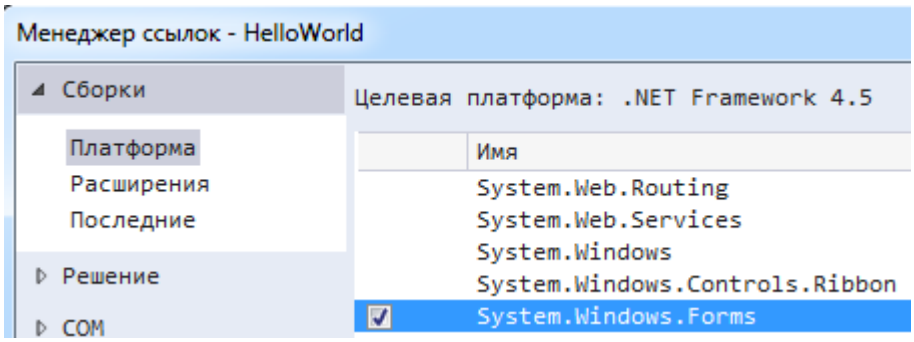
```
Public Sub Activate(ByVal addInSiteObject As Inventor.ApplicationAddInSite, _
     ByVal firstTime As Boolean) Implements Inventor.ApplicationAddInServer.Activate
            ' This method is called by Inventor when it loads the AddIn.
            ' The AddInSiteObject provides access to the Inventor Application object.
            ' The FirstTime flag indicates if the AddIn is loaded for the first time.
            ' Initialize AddIn members.
            m_inventorApplication = addInSiteObject.Application
            MsgBox("Здравствуй мир!")
            ' TODO:  Add ApplicationAddInServer.Activate implementation.
            ' e.g. event initialization, command creation etc.
End Sub
```

Для C#:

Для вывода стандартного окна на C#, необходимо сначала подключить ссылку на класс ***System.Windows.Forms***. Для этого в ***«Обозревателе решений»*** вызываем контекстное меню на строке ***References***:



В ***«Менеджере ссылок»*** находим и подключаем нужную ссылку:

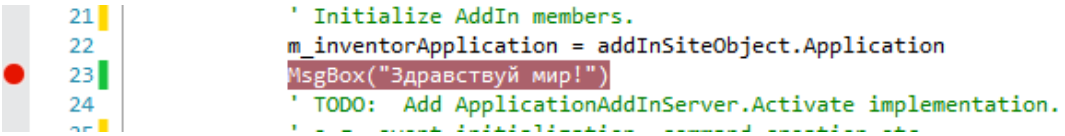


Теперь дописываем программный код:

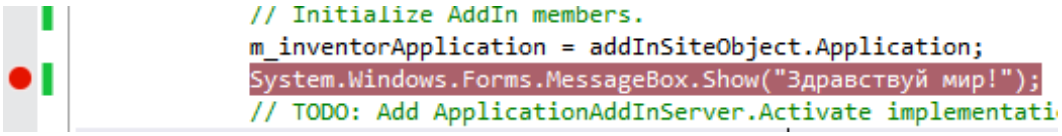
```
public void Activate(Inventor.ApplicationAddInSite addInSiteObject, bool firstTime)
        {
            // This method is called by Inventor when it loads the addin.
            // The AddInSiteObject provides access to the Inventor Application object.
            // The FirstTime flag indicates if the addin is loaded for the first time.
            // Initialize AddIn members.
            m_inventorApplication = addInSiteObject.Application;
            System.Windows.Forms.MessageBox.Show("Здравствуй мир!");
            // TODO: Add ApplicationAddInServer.Activate implementation.
            // e.g. event initialization, command creation etc.
        }
```

Создаем точку останова на строке, где будет выводиться окно:

VB.NET:

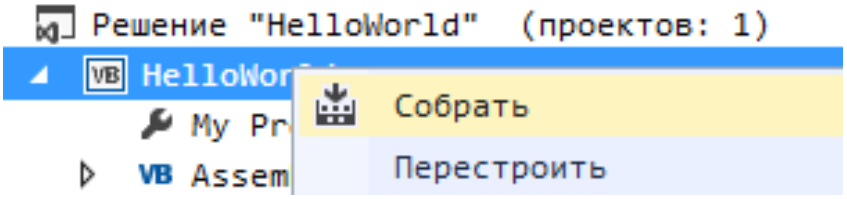


C#:

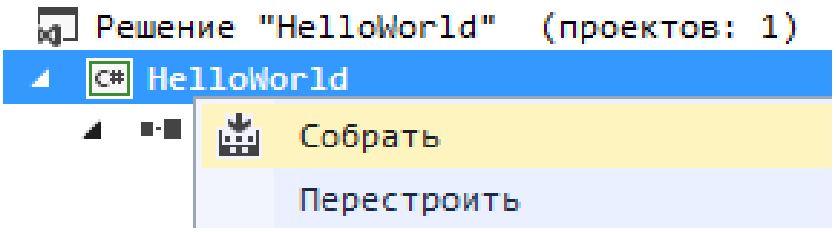


Далее компилируем программу, через контекстное меню в ***«Обозревателе решений»***:

VB.NET:



C#:



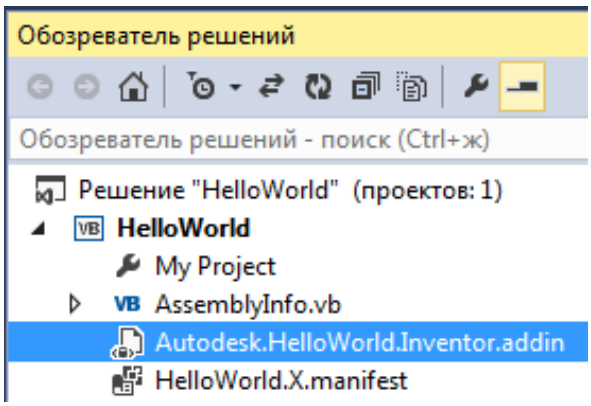
Отладка Visual Studio и Visual Studio Express через *«Присоединение к процессу»*

AddIn на отладку можно подключить двумя способами.

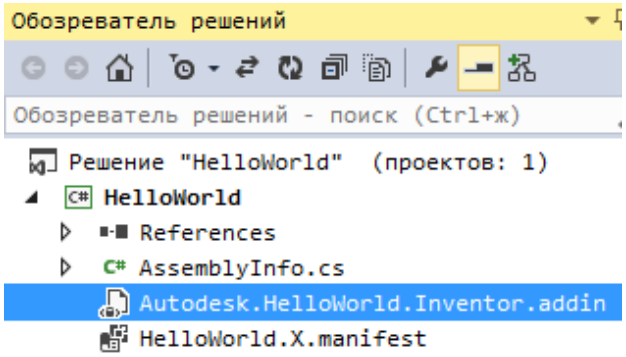
Первый способ универсальный для Visual Studio и для Visual Studio Express. Суть этого способа чтобы подключится к запущенному заранее процессу Inventor.

Для этого откроем файл с расширением ***.addin***:

VB.NET:



C#:



В этом файле найдем тег ***<LoadOnStartUp>*** и заменим в нем значение на ***0***:

```
<LoadOnStartUp>0</LoadOnStartUp>
```

Т.е. инициировать запуск отладки AddIn будем в ручную через пользовательский интерфейс самого Inventor.

Тег ***<Assembly>*** должен будет содержать полный путь к откомпилированному DLL-файлу. Этот путь у вас скорее всего будет отличаться от моего:

```
<Assembly>D:\VSProjects\HelloWorld(CS)\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>
```

Сохраняем ***.addin***-файл и копируем его любую из двух папок для (первая папка не зависит от установленной версии Inventor, но версия Inventor должна быть не старше 2012):

либо

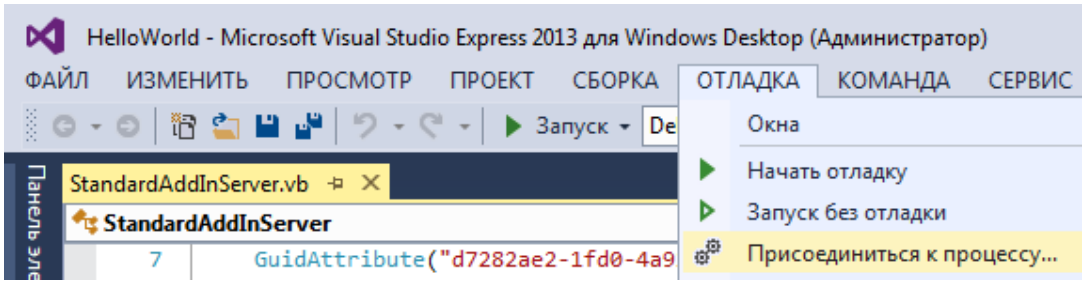
C:\Users\Все пользователи\Autodesk\Inventor Addins

либо

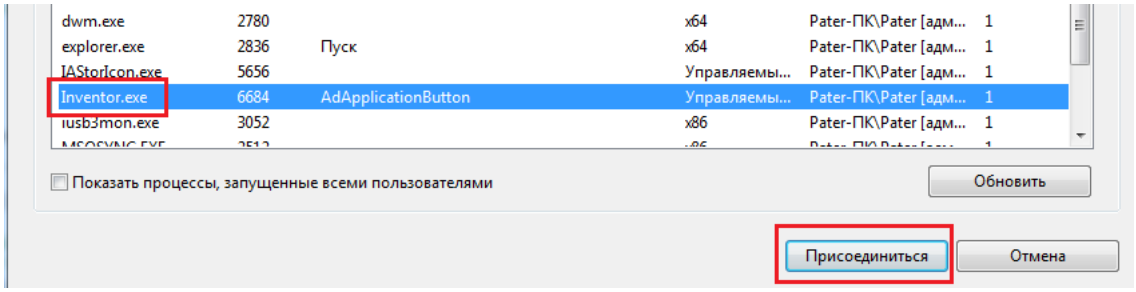
C:\Users\Все пользователи\Autodesk\Inventor 2015\Addins

Запускаем непосредственно сам Inventor:

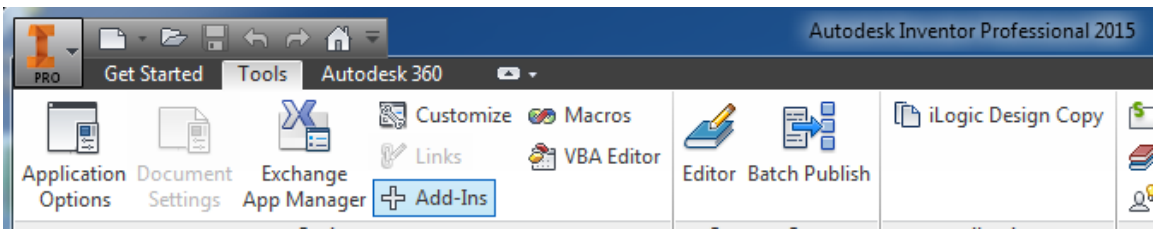
Теперь в Visual Studio или Visual Studio Express запускаем отладку через команду ***«Присоединится к процессу»***:



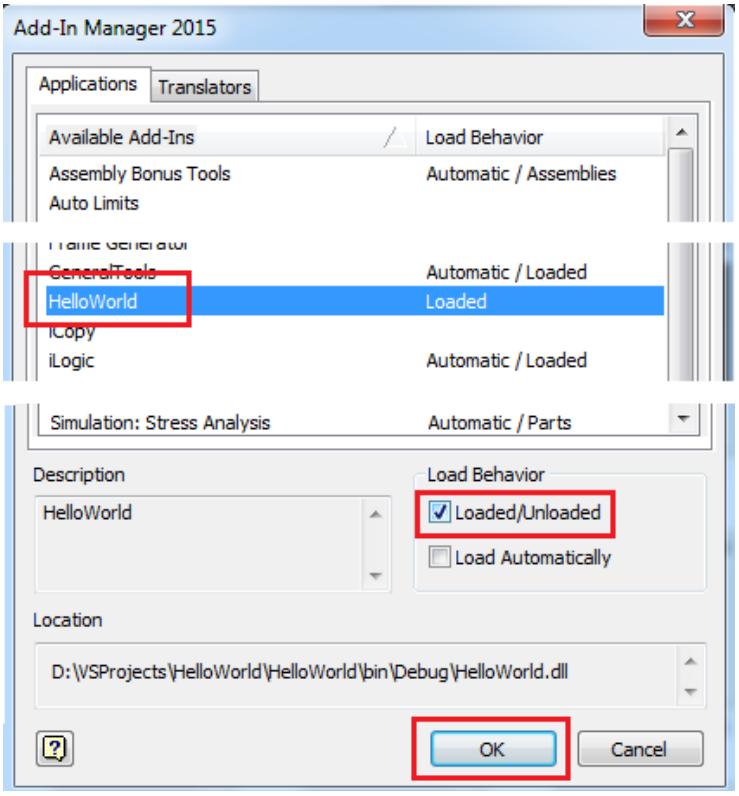
Появится следующее окно в котором будет нужно выбрать запущенный процесс Inventor:



К нужному процессу присоединились, но пока ничего не происходит. Чтобы Inventor вызвал процедуру (функцию) ***Activate***, где у нас находится точка останова, необходимо в ручную, в ***«Менеджере надстроек»***, произвести загрузку данного AddIn. Вызовем ***«Менеджер надстроек»*** в загруженном Inventor:



Появится окно со списком AddIn, выбираем нужный нам и устанавливаем опцию загрузки:



После нажатия кнопки «Ок» Visual Studio должна среагировать и остановится в точке останова, т.к. Inventor вызвал процедуру ***Activate***. При этом появится возможность отслеживать состояние имеющихся программных объектов:

VB.NET:

```
22          m_inventorApplication = addInSiteObject.Application
23          MsgBox("Здравствуй мир!")
24          ' TODO:  Add ApplicationAddInServer.Activate implementation.
25          ' e.g. event initialization, command creation etc.
```

Локальные

Имя	Значение
Me	{HelloWorld.HelloWorld.StandardAddInServer}
addInSiteObject	{Inventor.ApplicationAddInSite}
firstTime	True

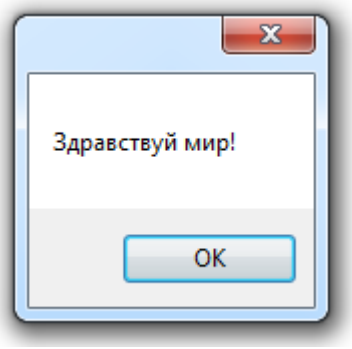
C#:

```
// Initialize AddIn members.
m_inventorApplication = addInSiteObject.Application;
System.Windows.Forms.MessageBox.Show("Здравствуй мир!");
// TODO: Add ApplicationAddInServer.Activate implementation.
// e.g. event initialization, command creation etc.
```

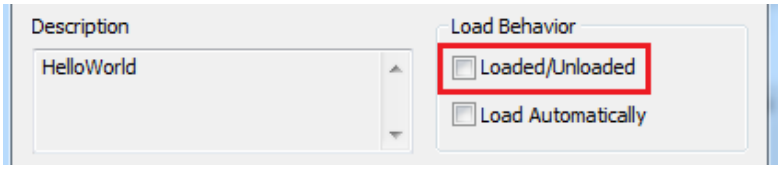
Локальные

Имя	Значение
this	{HelloWorld.StandardAddInServer}
addInSiteObject	COM-объект
firstTime	true

Дальнейшее продолжение работы программы приведет к выводу окна с сообщением:



Соответственно, если будет произведена выгрузка AddIn через «Менеджер надстроек»:

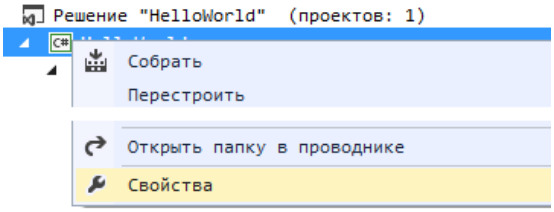


то Inventor инициализирует вызов процедуры ***Deactivate***.

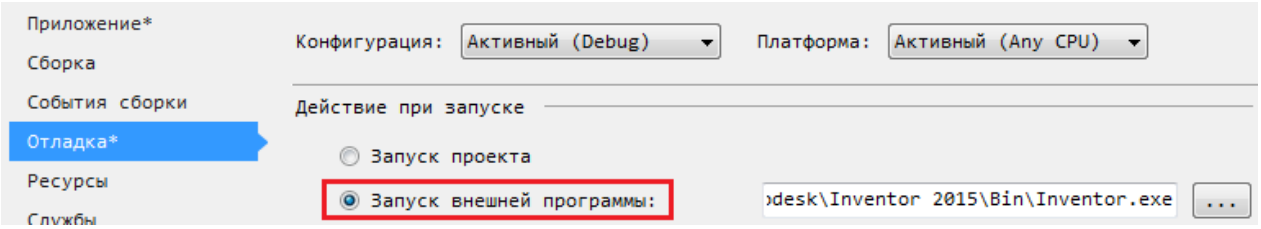
При необходимости изменения программного кода перед компиляцией проекта ***Inventor придется выгрузить***. Это связано с тем, что при загрузки Inventor в память, Inventor забирает права на редактирование DLL-файла от AddIn.

Отладка Visual Studio и Visual Studio Express через *«Запуск»*

Что бы каждый раз при запуске отладки через ***«Присоединение к процессу»***, не совершать много кликов мышкой, Visual Studio имеет настройку для автоматизации этих рутинных операций. Для этого нужно подключить Inventor в качестве программы для отладки AddIn. Для этого открываем свойства проекта через контекстное меню в ***«Обозревателе решений»***:



Ищем в меню ***«Отладка»*** текстовое поле ***«Запуск внешней программы»***, в котором пропишем к запускному файлу ***Inventor.exe***:



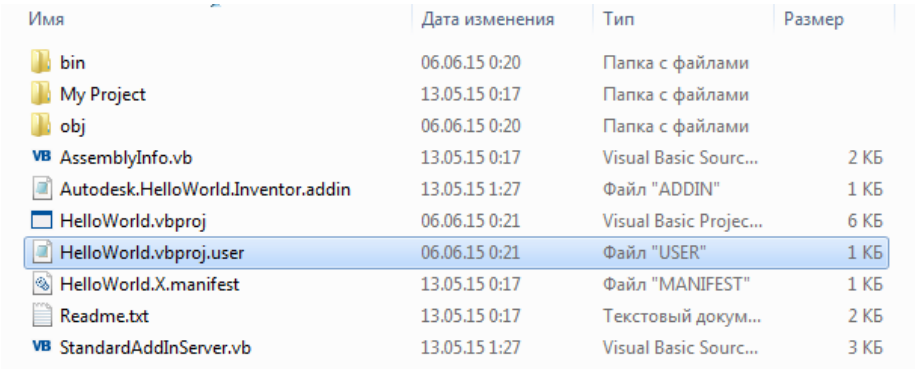
Visual Studio Express тоже имеет возможность автоматического запуска Inventor при отладке AddIn, как и полноценная версия Visual Studio. Внимательно сравним папки с программным проектами от полноценного Visual Studio и Visual Studio Express. Легко заметить, что заметить, то в программном проекте Visual Studio Express отсутствует xml-файл в формате:

Имя проекта.vbproj.user

В нашем случае отсутствует файл с именем:

HelloWorld.vbproj.user

В этом файле, как раз, и должны находиться настройки по запуску внешней программы для отладки, в нашем случае Inventor. Достаточно создать текстовый файл в папке с программным проектом и переименовать его в соответствии с именем проекта:



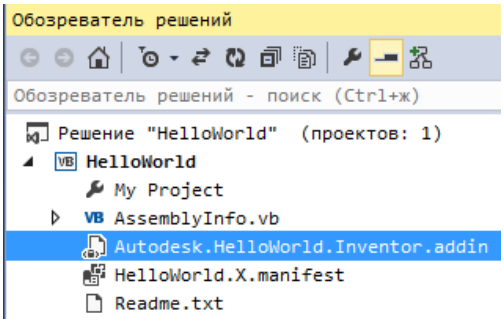
Далее открываем этот файл на редактирование и вписываем в него данные в формате xml:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="12.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup Condition="'$(Configuration)|$(Platform)' == 'Debug|AnyCPU'">
    <StartAction>Program</StartAction>
    <StartProgram>D:\Autodesk\Inventor 2015\Bin\Inventor.exe</StartProgram>
  </PropertyGroup>
  <PropertyGroup Condition="'$(Configuration)|$(Platform)' == 'Debug|x64'">
    <StartProgram>D:\Autodesk\Inventor 2015\Bin\Inventor.exe</StartProgram>
    <StartAction>Program</StartAction>
  </PropertyGroup>
</Project>
```

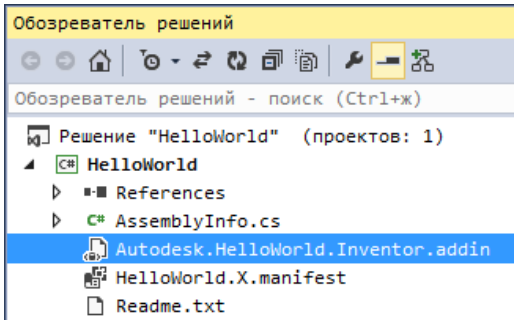
Думаю, что данный текст особенно в комментариях не нуждается. Сохраняем файл и ***перегружаем программный проект***. Теперь проект AddIn созданный на Visual Studio Express будет подключаться к отладке так же, как и проект созданный на полноценной Visual Studio.

Далее в ***«Обозревателе решений»*** открываем на редактирование файл с расширением ***.addin***:

VB.NET:



C#:



В этом файле имеется тэг ***<Assembly>***, в этот тэг прописываем абсолютный путь к откомпилированному Dll-файлу. Абсолютный путь зависит от того где на жестком диске расположен проект, в моем случае это выглядит так, у вас скорее всего будет другой абсолютный путь:

```
<Assembly>D:\VSProjects\HelloWorld(CS)\HelloWorld\bin\Debug\HelloWorld.dll</Assembly>
```

Если вы экспериментировали с отладкой через ***«Присоединится к процессу»***, то нужно проверить значение тега ***<LoadOnStartUp>***, что бы он был равен ***1***:

```
<LoadOnStartUp>1</LoadOnStartUp>
```

Сохраняем ***.addin***-файл и копируем его любую из двух папок (первая папка не зависит от установленной версии Inventor, но версия Inventor должна быть не старше 2012):

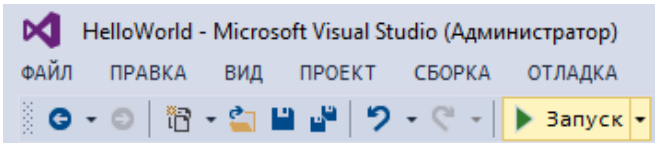
либо

C:\Users\Все пользователи\Autodesk\Inventor Addins

либо

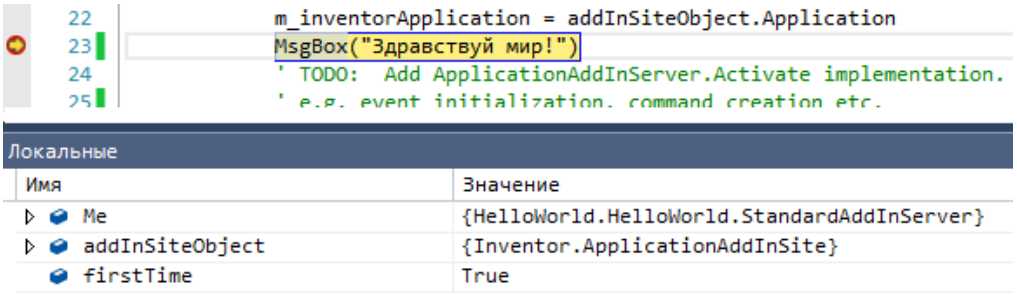
C:\Users\Все пользователи\Autodesk\Inventor 2015\Addins

Запускаем в Visual Studio проект на выполнение:

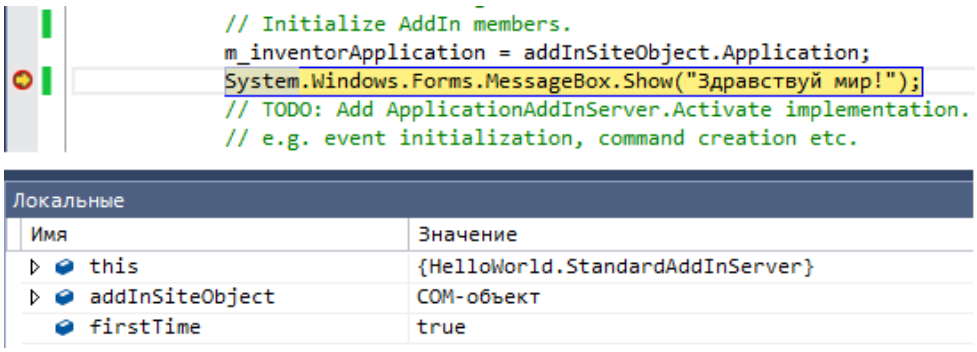


После запуска проекта на отладку должен начать загружаться Inventor, после чего программа остановится в назначенной точке. При этом появится возможность отслеживать состояние имеющихся программных объектов:

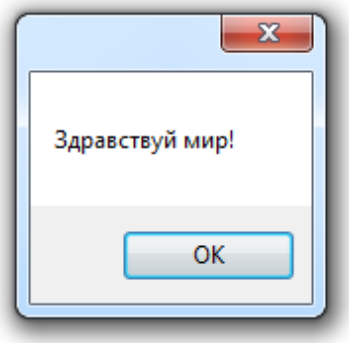
VB.NET:



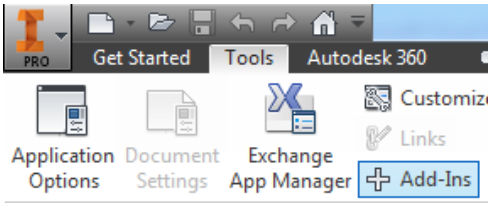
C#:



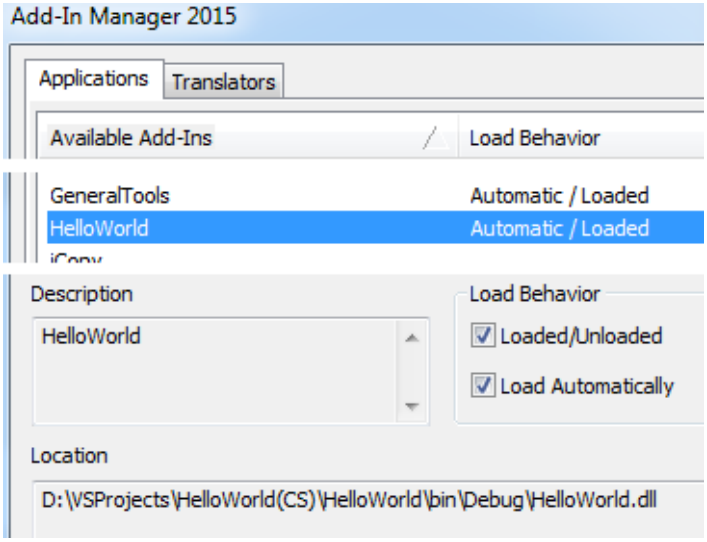
Дальнейшее продолжение работы программы приведет к выводу окна с сообщением:



После закрытия этого окна Inventor полностью загрузится, откроем ***«Менеджер надстроек»***:



В нем можно будет увидеть созданный нами AddIn.



Если остановить отладку в Visual Studio произойдет автоматическая выгрузка Inventor:

Если все было сделано по описанным выше шагам корректно, но окно сообщения «Здравствуй мир» не появилось, то единственной причиной, этого может быть не соответствие выбранной версии Framework для компиляции проекта. Поддерживаемую Inventor версию Framework можно увидеть в файле ***Inventor.exe.config***.

Файл ***Inventor.exe.config*** находится в папке ***«Bin»*** в месте, где установлен сам Inventor. Ниже приведен фрагмент из конфигурационного файла для Inventor 2015:

```
<startup useLegacyV2RuntimeActivationPolicy="true">
     <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/>
</startup>
```

как видно, Inventor 2015 поддерживает версии Framework 4.5 и более ранние версии.

На этом пробное тестирование подошло к концу. Далее мы перейдем к подробному рассмотрению многих параметров проекта для создания AddIn. Для это мы будем создавать AddIn не из готового шаблона, а с базового проекта для создания DLL-файла, который предлагает Visual Studio на языках VB.NET и C#. Попутно будет приведено подробное описание многих параметров для AddIn находящихся в описании API Inventor. И ближе к концу будет рассмотрено создание AddIn на языке C++/CLI, создание смешенного решения на C# и C++, а так же создание AddIn на чистом «нативном» C++.

Немного о технологии подключения к Inventor.

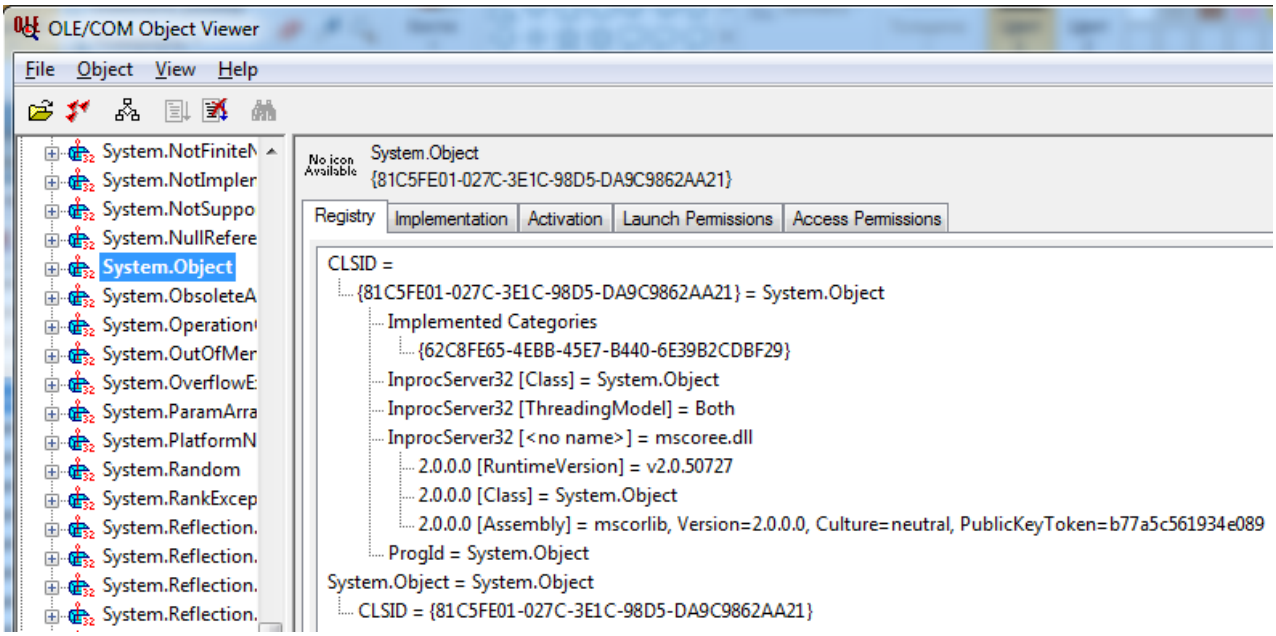
Изначально, Microsoft, разработала COM-технологию (Component Object Model), для создания программных компонентов независимых от языка программирования. COM-технология позволяет, как из кирпичиков, создавать и модифицировать приложение. Но здесь не нужно путать DLL и COM. Если привести простую аналогию, то DLL-это контейнер-грузовик, а COM-это содержимое грузовика, т.е. COM-это набор правил создания DLL. Технология COM существует с 1993 года. Со временем технология COM модифицировалась до технологий ***IDispatch*** (на ***IDispatch*** основана работа VBA, VB6 и собственно API Inventor), а также в другие технологии OLE и ActiveX.

Надо отметить, что базовая часть COM-технология основанная на интерфейсе ***IUnknown*** работает на «бинарном» программном коде, поэтому имеет высокую производительность. Напротив, модифицированная COM-технология на интерфейсе ***IDispatch*** может не обладать большой производительностью из-за реализации «позднего связывания», «интеллектуального приведения типов» и пр. что приводит к упрощению программирования на VBA и VB.NET.

За это время на технологии COM основе было написано бесчисленное множество компонентов. Сам Inventor так же достаточно напичкан COM-компонентами. Так же и сам Framework (NET) в основе своей содержит COM-технологию. Что бы убедится в этом достаточно открыть утилиту ***OleView.exe***, которая находится, в моем случае, папке:

C:\Program Files (x86)\Windows Kits\8.0\bin\x64\oleview.exe

Запускать утилиту обязательно нужно от имени ***Администратора***. После запуска данной утилиты легко убедится, что весь Framework это COM.



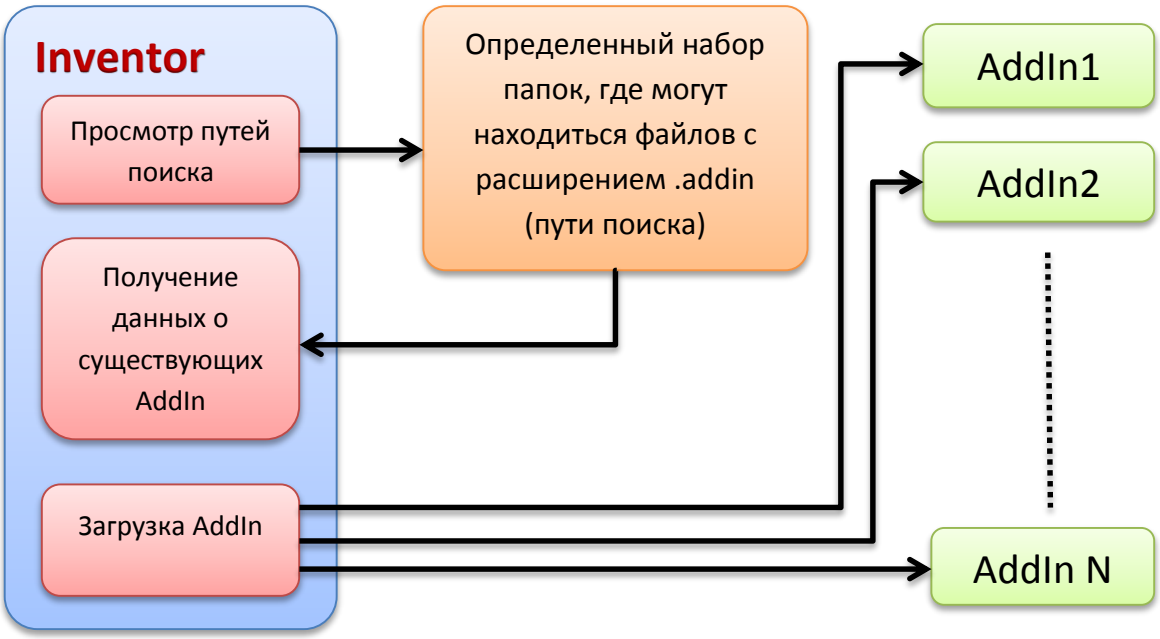
Но у технологии COM быстро выявился один значительный недостаток: регистрация в реестре Windows. Это вызывает известные проблемы:

- Увеличиваются накладные расходы по обработке ключей реестра при большом количестве записей в нем
- Нельзя зарегистрировать больше одной DLL с одинаковым уникальным 128-ми битным ключом CLSID (потенциально вызывает конфликты, если разные программы требуют разные версии DLL)
- Работа в классическом стиле COM достаточно сложная и кропотливая
- Достаточно емкие правила описания ключей в реестре.
- Требуются права администратора для работы с реестром (хотя это не всегда является недостатком)

Поэтому Microsoft через некоторое время выпустил замену COM-технологии, известную как NET. Полной заменой COM-технологии NET-технология, конечно же, не стала, т.к. за более высокоуровневое программирование всегда приходится «платить» производительностью, некоторыми ограничениями по функциональность и «гибкостью» программы.

Изначально, написание AddIn для Inventor так же производилось на базе COM-технологии. Но с появлением NET-технологии, со временем, разработчики Inventor упростили способ подключения AddIn к Inventor. Новый способ подключения называется ***«Registry free»***, т.е. «без регистрации». ***«Registry free»*** значительно упрощает подключение AddIn. Т.к. с появлением NET-регистрируемых в реестре dll-файлов, в сочетании с разной разрядностью Windows, сделало подключение AddIn несколько запутанной процедурой.

Общий принцип подключения по технологии ***«Registry free»*** представлен на схеме:

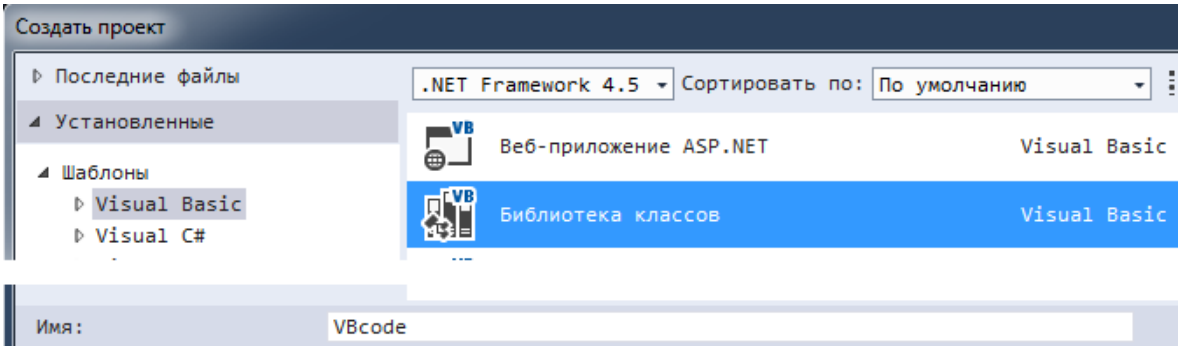


Далее рассмотрим пошаговое создание AddIn «в ручную», для того что бы пройтись по всем незнакомым местам. Это даст более полное понимание, для чего нужен каждый объект. Конечно же, в будущем вы будете легко пользоваться готовыми шаблонами, уже без «белых пятен» в понимании устройства сгенерированного шаблоном программного кода.

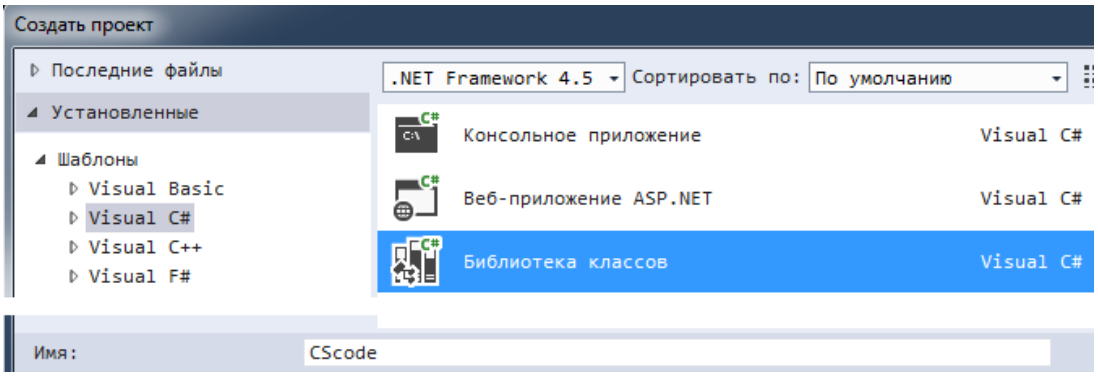
Интерфейс подключения к Inventor.

Как и в случае с EXE-файлом для Inventor, перед началом работы необходимо подключить библиотеку API Inventor. Для этого нужно вызвать ***«Менеджер ссылок»***. Создадим приложение «Библиотеки классов» на VB.NET или C#.

VB.NET с названием проекта VBcode:

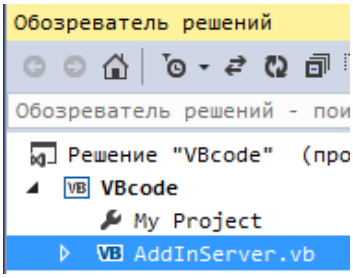


C# с названием проекта CScode:

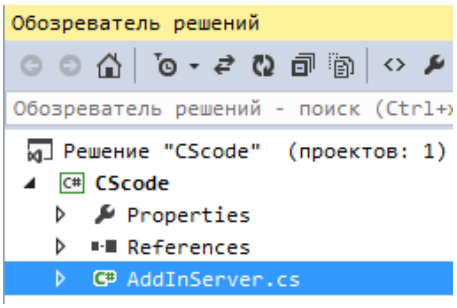


Переименуем автоматически созданный файл с классом в ***AddInServer***:

VB.NET:

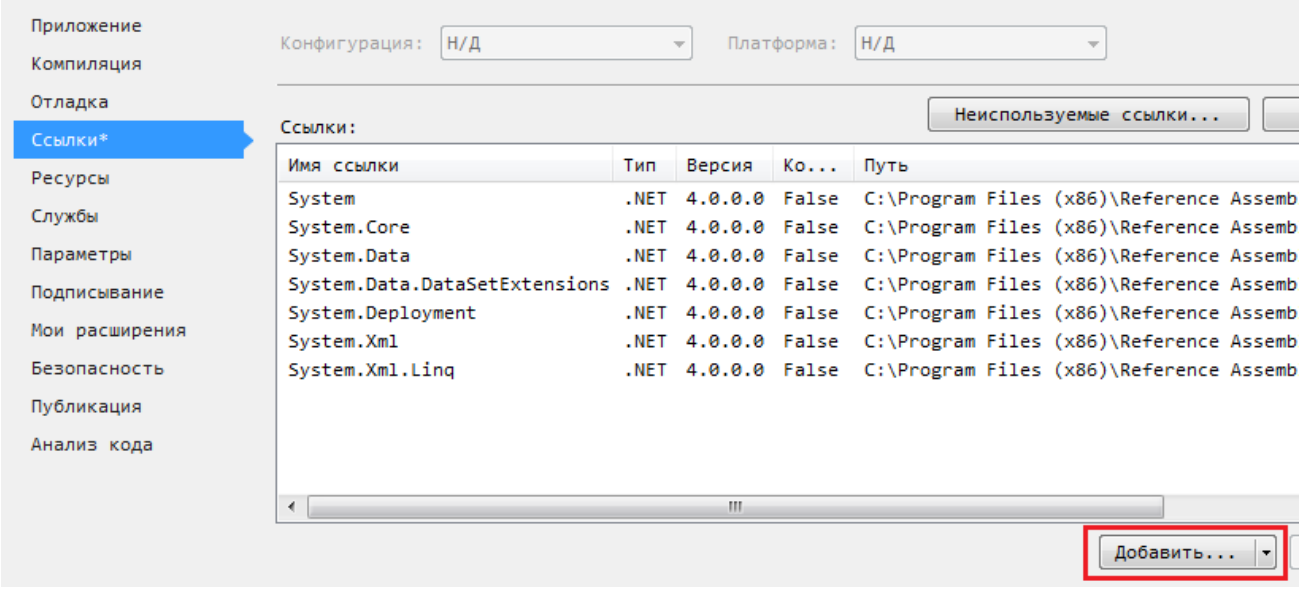


C#:

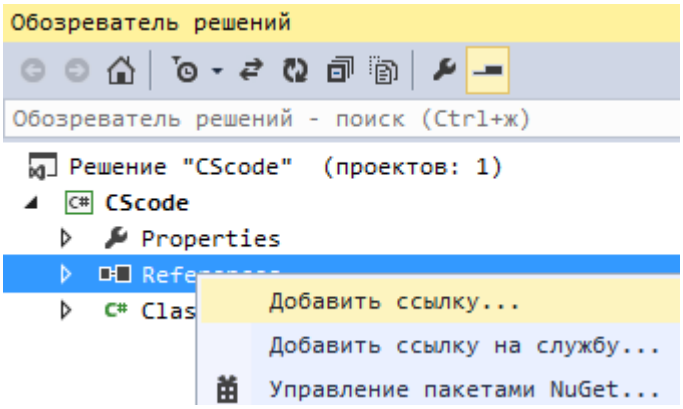


Далее необходимо подключить API Inventor. Это делается через ***«Менеджер ссылок»***.

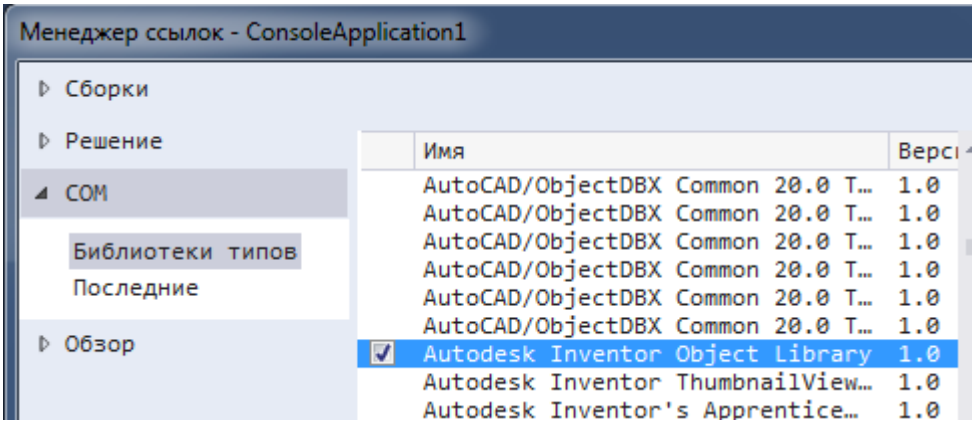
Для VB.NET ***«Менеджер ссылок»*** находится на одной из закладок в свойствах проекта.



Для C# ***«Менеджер ссылок»*** вызывается из контекстного меню в ***«Обозревателе решений»***

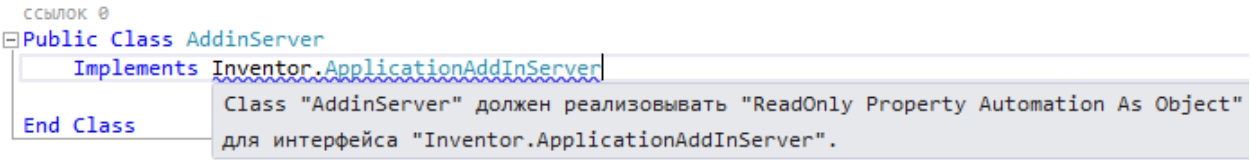


Появится окно, где в списке COM-приложений нужно выбрать ***«Autodesk Inventor Object Library»***.

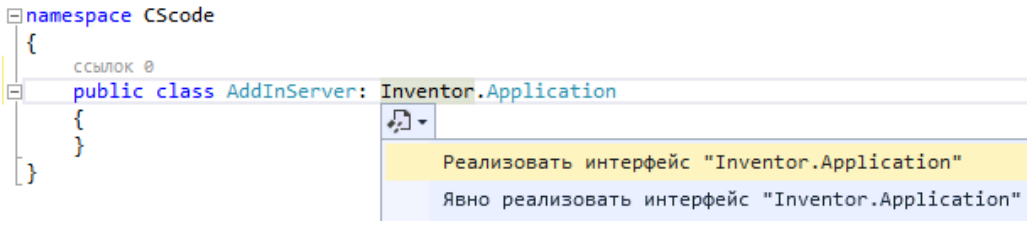


Для того что бы создаваемый AddIn и Inventor могли понимать друг друга необходимо в программном коде AddIn реализовать специальный интерфейс: ***Inventor.ApplicationAddInServer***

Код VB.NET:



Код C#:



Интеллектуальный помощник IntelliSense из Visual Studio, нам пытается помочь, объясняя, что унаследованный интерфейс должен быть реализован. Для реализации интерфейса в VB.NET достаточно нажать клавишу «Enter» в конце строки. В C# появится возможность вызвать контекстное меню с предложением реализации интерфейса. В результате чего программный код будет выглядеть так:

VB.NET (добавим пространство имен ***VBcode***, если его нет):

```
Namespace VBcode
Public Class AddinServer
    Implements Inventor.ApplicationAddInServer
    Public Sub Activate(AddInSiteObject As Inventor.ApplicationAddInSite, FirstTime As Boolean) _
        Implements Inventor.ApplicationAddInServer.Activate
    End Sub

    Public ReadOnly Property Automation As Object Implements _
       Inventor.ApplicationAddInServer.Automation
        Get
               Return Nothing
        End Get
    End Property

    Public Sub Deactivate() Implements Inventor.ApplicationAddInServer.Deactivate
    End Sub

    Public Sub ExecuteCommand(CommandID As Integer) _
        Implements Inventor.ApplicationAddInServer.ExecuteCommand
    End Sub
End Class

End Namespace
```

C# (здесь я немного убрал «лишний» сгенерированный код от IntelliSense):

```
namespace CScode
{
    public class AddInServer : Inventor.ApplicationAddInServer
    {
        public void Activate(Inventor.ApplicationAddInSite AddInSiteObject, bool FirstTime)
        {
        }

        public dynamic Automation
        { get { return null; } }

        public void Deactivate()
        {
        }

        public void ExecuteCommand(int CommandID)
        {
        }
    }
}
```

Скажем пару слов про эти три процедуры (или методы) и одно свойство. Актуальными для написания AddIn являются только две процедуры: ***Activate*** и ***Deactivate***.

- Процедура ***Activate***: запускается самим Inventor при загрузке AddIn или при ручном включении AddIn через пользовательский интерфейс в Inventor. В сигнатуре этой процедуры содержится два объекта ***AddInSiteObject*** и ***FirstTime***. Объект ***AddInSiteObject*** содержит в себе ссылку на объект ***AddInSiteObject.Application***, который в свою очередь является корневым объектом API Inventor, через который и будет управляться Inventor. Булевская переменная ***FirstTime=true***, сообщает о том, что процедура была вызвана самим Inventor первый раз, и поэтому её можно использовать как индикацию необходимости построения различных меню и пользовательских интерфейсов. Хотя из моей практики я не разу не встречал, что бы ***FirstTime=false***, он всегда был равен ***true*** даже при ручном включении/отключении AddIn через пользовательский интерфейс в Inventor.
- Процедура ***Deactivate***: запускается при закрытии Inventor или при отключении AddIn через пользовательский интерфейс. Здесь необходимо удалить все не нужные объекты. В отличии от программ на C++, при написании программы на NET-языке, нет необходимости тщательно следить за тем что бы все объекты были удалены и ссылки на них были установлены на null для C# и Nothing для VB.NET. Т.к. все «управляемые» ресурсы будут автоматически утилизироваться сборщиком мусора, встроенным в NET Framework, если же не были конечно задействованы «неуправляемые» ресурсы в памяти, через маршализацию, то их нужно будет обязательно освободить при деактивации.
- Процедура ***ExecuteCommand***: является устаревшей и на данный момент давно не используется.
- Свойство ***Automation***: служит для реализации управления данного AddIn другими программами, через свой дополнительный интерфейс управления для данной надстройки. В этом свойстве можно возвратить данный интерфейс на базе COM-технологии. Сущность свойства ***Automation*** будет видна при рассмотрении написания AddIn на чистом «нативном» C++.

К описанию созданного класса, перед его началом, необходимо добавить атрибут, который содержит в себе уникальный 128-ми битный идентификатор (GUID):

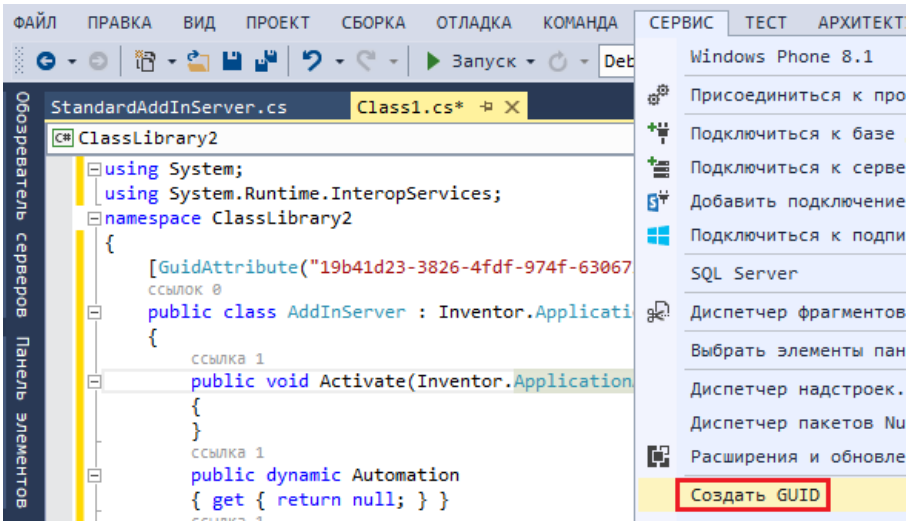
Фрагмент атрибута VB.NET:

```
Imports System.Runtime.InteropServices
Namespace VBcode
    <GuidAttribute("3c3167fa-d413-40d9-998e-43ab06bfc53e")> _
Public Class AddinServer
             Implements Inventor.ApplicationAddInServer
```

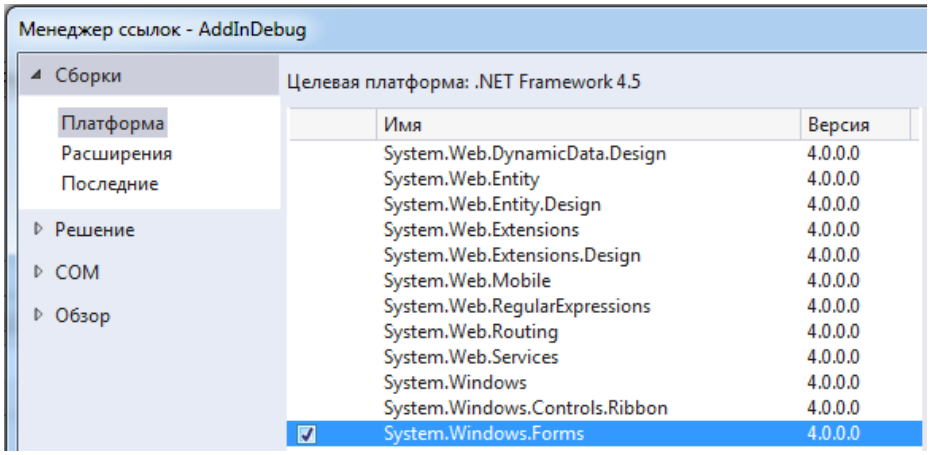
Фрагмент атрибута C#:

```
using System.Runtime.InteropServices;
namespace CScode
{
    [GuidAttribute("19b41d23-3826-4fdf-974f-630672ae3488")]
    public class AddInServer : Inventor.ApplicationAddInServer
```

Для генерирования GUID в Visual Studio есть встроенный инструмент:



Для завершения работы с данным классом, добавим окно с сообщением «Здравствуй мир!». Для окна с сообщением «Здравствуй мир!», подключим ссылку на ***System.Windows.Forms*** через ***«Менеджер ссылок»***:



Также добавим член класса для хранения ***Inventor.Application*** и в результате программный код будет выглядеть так:

Код VB.NET:

```
Imports System.Runtime.InteropServices
Namespace VBcode
        <GuidAttribute("3c3167fa-d413-40d9-998e-43ab06bfc53e")> _
    Public Class AddinServer
        Implements Inventor.ApplicationAddInServer
        Private InvApp As Inventor.Application
        Public Sub Activate(AddInSiteObject As Inventor.ApplicationAddInSite, FirstTime As Boolean)
_
            Implements Inventor.ApplicationAddInServer.Activate
            InvApp = AddInSiteObject.Application
            System.Windows.Forms.MessageBox.Show("VB.NET:Здравствуй " & InvApp.Caption)
        End Sub
        Public ReadOnly Property Automation As Object _
            Implements Inventor.ApplicationAddInServer.Automation
            Get
                Return Nothing
            End Get
        End Property
        Public Sub Deactivate() Implements Inventor.ApplicationAddInServer.Deactivate
        End Sub
        Public Sub ExecuteCommand(CommandID As Integer) _
            Implements Inventor.ApplicationAddInServer.ExecuteCommand
        End Sub
    End Class
End Namespace
```

Код C#:

```
using System.Runtime.InteropServices;
namespace CScode
{
    [GuidAttribute("19b41d23-3826-4fdf-974f-630672ae3488")]
    public class AddInServer : Inventor.ApplicationAddInServer
    {
        private Inventor.Application InvApp;
        public void Activate(Inventor.ApplicationAddInSite AddInSiteObject, bool FirstTime)
        {
            InvApp = AddInSiteObject.Application;
            System.Windows.Forms.MessageBox.Show("C#: Здравствуй " + InvApp.Caption);
        }
        public dynamic Automation
        { get { return null; } }
        public void Deactivate()
        {
        }
        public void ExecuteCommand(int CommandID)
        {
        }
    }
}
```

Манифест сборки.

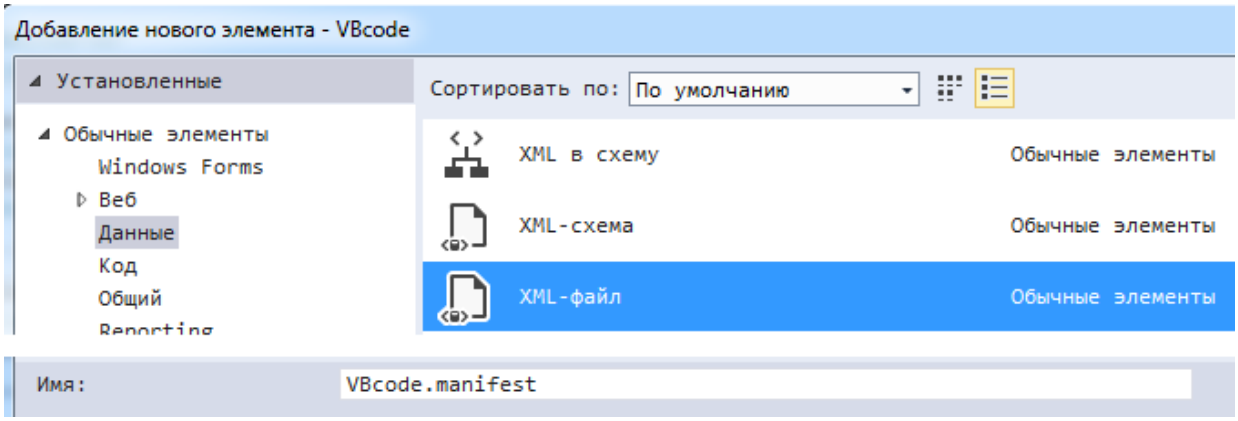
Для замены регистрации DLL-файла в реестре в технологии NET предусмотрен инструмент манифеста. Манифест представляет из себя текстовый файл в формате xml. Данные из манифеста внедряются в DLL-файл после его компиляции. По этим данным Inventor находит класс содержащий интерфейс для взаимодействия с AddIn.

Добавим файл манифеста к нашему проекту. В качестве файла манифеста изначально может быть любой xml-файл, и даже текстовый файл, который потом будет переименован. Задаем имя сразу или переименовываем потом в «Обозревателе решений». Для примера я создаю новый xml-файл и сразу назначаю ему имя и расширение. Хотя хэлп API Inventor рекомендует давать имя манифесту в формате:

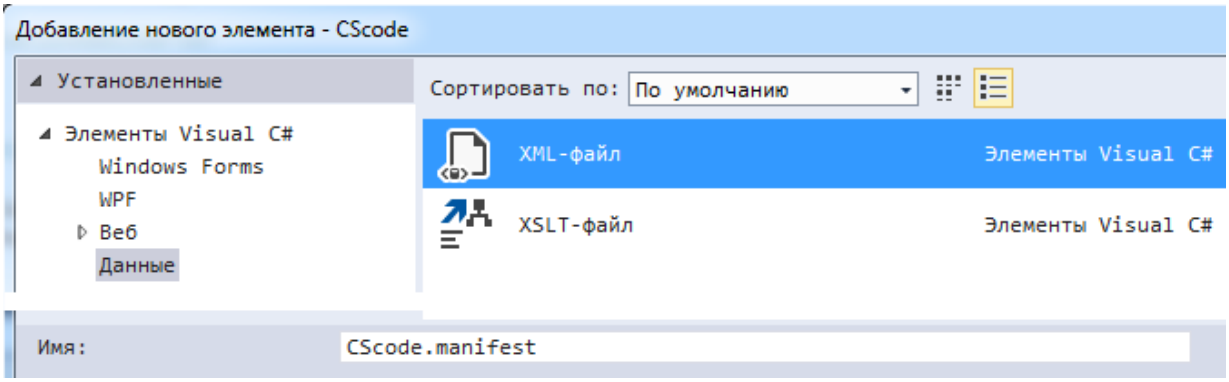
Имя DLL-файла.X.manifest

Такой формат нужен для автоматического внедрения манифеста в проектах на C++, а так как у нас другие языки программирования, то строго придерживаться этого правила не обязательно.

VB.NET:



C#:



Открываем созданный файл на редактирование и вставляем в него следующие данные

VB.NET:

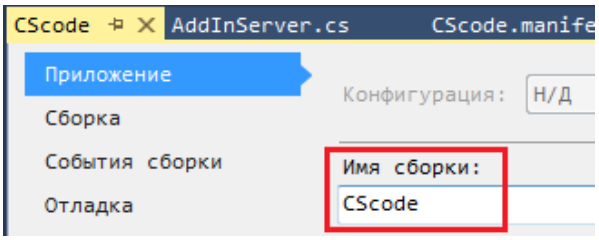
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity name="VBcode" version="1.0.0.0" />
  <clrClass clsid="{3c3167fa-d413-40d9-998e-43ab06bfc53e}"
            progid="VBcode.VBcode.AddinServer"
            threadingModel="Both"
            name="VBcode.VBcode.AddinServer"
            runtimeVersion="" />
  <file name="VBcode.dll" hashalg="SHA1" />
</assembly>
```

C#:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity name="CScode" version="1.0.0.0" />
  <clrClass clsid="{19b41d23-3826-4fdf-974f-630672ae3488}"
            progid="CScode.AddInServer"
            threadingModel="Both"
            name="CScode.AddInServer"
            runtimeVersion="" />
  <file name="CScode.dll" hashalg="SHA1" />
</assembly>
```

Разберем ключевые тэги и их атрибуты по порядку

- `<assemblyIdentity name="VBcode" version="1.0.0.0" />`
  `<assemblyIdentity name="CScode" version="1.0.0.0" />`
  -имя сборки (совпадает с именем DLL-файла), данные берутся из настроек проекта



- VB.NET: `clsid="{3c3167fa-d413-40d9-998e-43ab06bfc53e}"`
  C#: `clsid="{19b41d23-3826-4fdf-974f-630672ae3488}"`

  –уникальный GUID (или CLSID) берется из атрибута класса ***«GuidAttribute»***, который реализует интерфейс присоединения к Inventor.

```
Namespace VBcode
    <GuidAttribute("3c3167fa-d413-40d9-998e-43ab06bfc53e")> _
    ссылок 0
    Public Class AddinServer
        Implements Inventor.ApplicationAddInServer
```

```
namespace CScode
{
     [GuidAttribute("19b41d23-3826-4fdf-974f-630672ae3488")]
    ссылок 0
    public class AddInServer : Inventor.ApplicationAddInServer
```

- VB.NET: progid="VBcode.VBcode.AddinServer"
  C#: progid="CScode.AddInServer"
  - это более читабельный заменитель уникального 128-ми битного идентификатора GUID. Значение создается автоматически путем присоединения пространства имен к имени класса. Однако ***progid*** имеет ограничения на длину значения равную 39 символам и не может содержать знаков препинания кроме точки. В случае если при автоматическом генерировании ***progid*** эти правила не выполняются, то необходимо в ручную задать еще один атрибут классу:
  Для VB.NET:

```
Namespace VBcode
  <ProgIdAttribute("My.ProgId"), _
  GuidAttribute("3c3167fa-d413-40d9-998e-43ab06bfc53e")> _
           Public Class AddinServer
```

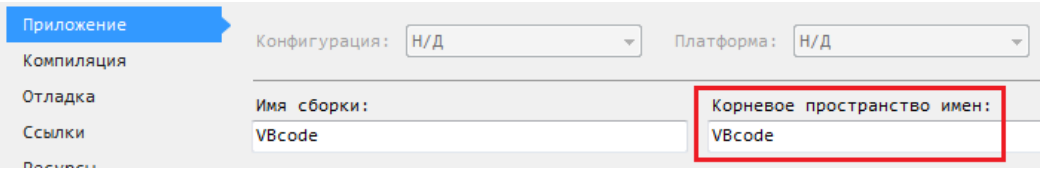
  Для C#:

```
namespace CScode
{
 [ProgIdAttribute("My.ProgId")]
 [GuidAttribute("19b41d23-3826-4fdf-974f-630672ae3488")]
           public class AddInServer : Inventor.ApplicationAddInServer
```

  тогда в файле манифеста должно стать:
  progid="My.ProgId"

- VB.NET: name="VBcode.VBcode.AddinServer"
  C#: name="CScode.AddInServer"

  Здесь, как и в предыдущем пункте, можно увидеть небольшое отличие в создании пространства имен между VB.NET и C#. В VB.NET существует понятие «Корневое пространство имен» («Root namespace»), это пространство имен является корневым (охватывает все пользовательские пространства имен) для всех участвующих пространствах имен во всем проекте.



  Поэтому «полное имя класса» на VB.NET в нашем случае будет состоять из формулы:

Корневое пространство имен. Пространство имен файла, где содержится класс. Имя Класса

  Т.е. в нашем случае, как раз, получается:
  VBcode.VBcode.AddinServer

В C# существует понятие «Пространство имен по умолчанию»

но это понятие не охватывает все остальные пространства имен, а существует рядом с ними и служит для служебных целей, например, для доступа к ресурсам проекта (VB.NET существует аналогичное пространство имен только со строгим именем ***My***). Поэтому «полное имя класса» на VB.NET в нашем случае будет состоять из формулы:

Пространство имен файла, где содержится класс. Имя Класса

В нашем случае получается:

CScode.AddInServer

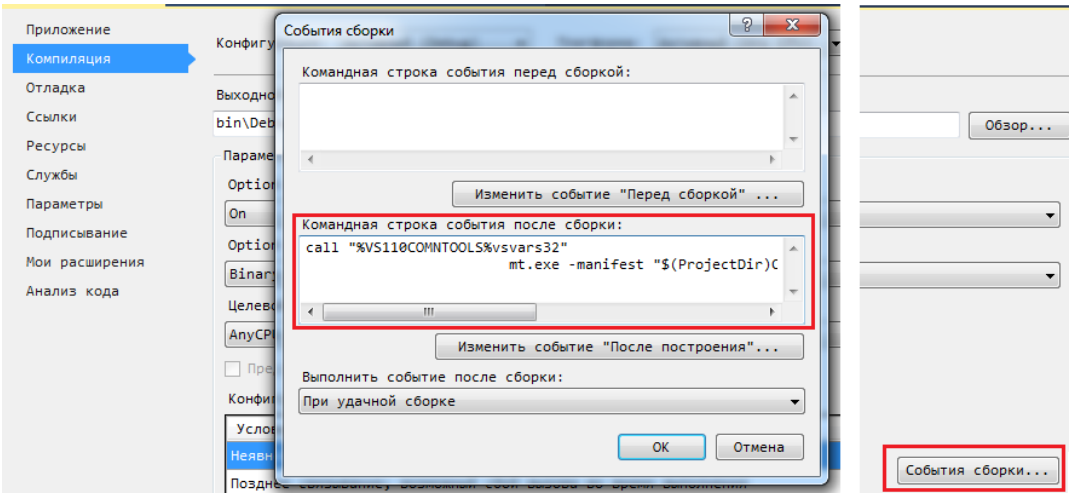
В данном примере на C# класс попал в «Пространство имен по умолчанию», но это не обязательно, класс может никак не пересекаться с «Пространством имен по умолчанию», часто это даже бывает полезно.

- VB.NET: `name="VBcode.dll"`
  C#: `name="CScode.dll"`
  Здесь просто имя откомпилированного файла.

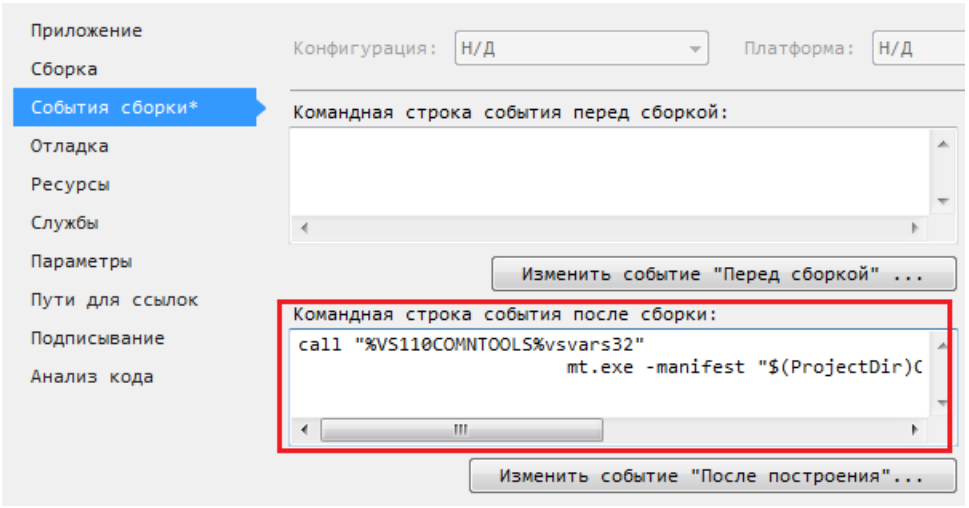
Внедрение манифеста сборки.

Созданный манифест сборки необходимо внедрить в DLL-файл. Это происходит в следующей последовательности: сначала Visual Studio компилирует DLL-файл и после компиляции происходит вызов специальной утилиты, которая и внедряет манифест. Для автоматизации процесса внедрения манифеста в Visual Studio существует специальный инструмент компиляции ***«События сборки»***.

VB.NET:



C#:



Рассмотрим синтаксис команды:

VB.NET:

call "%VS120COMNTOOLS%vsvars32"

mt.exe -manifest "$(ProjectDir) VBcode.manifest" -outputresource:"$(TargetPath)";#2

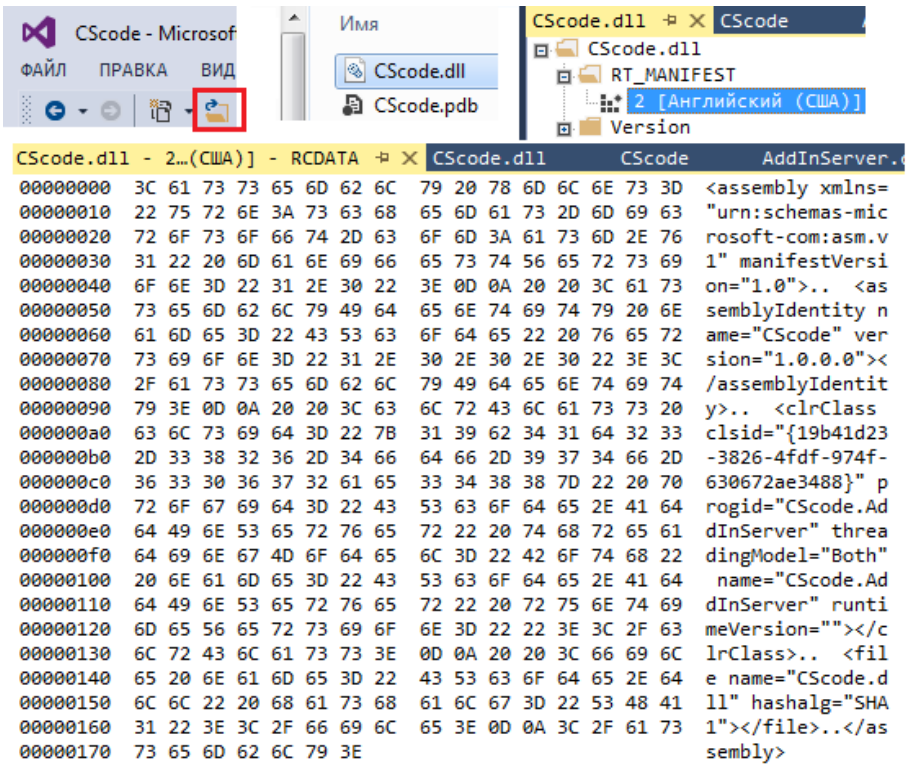
C#:

call "%VS120COMNTOOLS%vsvars32"

mt.exe -manifest "$(ProjectDir) CScode.manifest" -outputresource:"$(TargetPath)";#2

- ***%VS120COMNTOOLS%*** –это специальная папка Visual Studio, где находится файл ***vsvars32.bat***. Однако в командной строке события после сборке может возникать проблема, связанная с тем, что проект был создан в другой версии Visual Studio. Т.к ***«%VS120COMNTOOLS%»*** это просто часть пути к требуемому файлу ***vsvars32.bat***, то для исправления этого необходимо всего лишь исправить номер версии Visual Studio в этом месте, например:
  Visual Studio 2010: «%VS100COMNTOOLS%»
  Visual Studio 2012: «%VS110COMNTOOLS%»
  Visual Studio 2013: «%VS120COMNTOOLS%»
- ***$(ProjectDir)*** –это специальная переменная (аналог «макроса подстановки в C++») Visual Studio, которая имеет значение полного пути в корневой каталог программного проекта. Если файл манифеста будет находиться в подпапке, то это необходимо отразить в командной строке дополнительно дописав путь до подпапки.
- ***$(TargetPath)*** – это специальная переменная содержит полный путь к DLL-файлу и автоматически меняет свое значение при смене компиляции с ***Debug*** на ***Release*** и наоборот.
- ***#2*** – означает, что манифест будет прописываться в DLL-файла (например, в EXE-файл значение будет ***#1***)

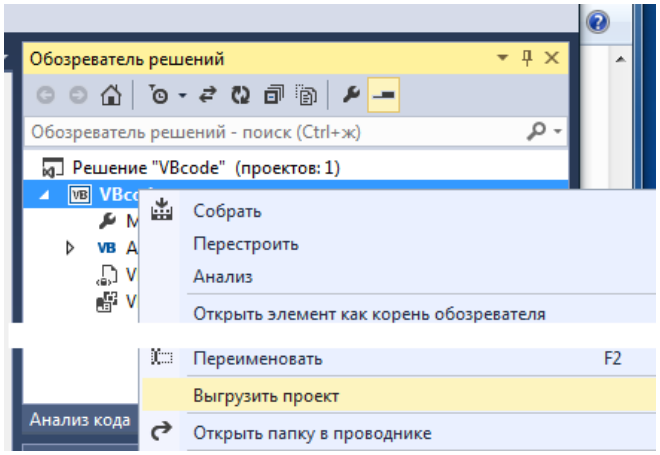
Для корректной работы ***mt.exe*** необходимо, чтобы между ***"%VS120COMNTOOLS%vsvars32"*** и ***mt.exe*** был разрыв строки, с перенесением текста на соседнюю строку, поскольку это две разные команды. Иначе внедрение манифеста не произойдет. Разрыв строки вставляется при помощи комбинации клавиш ***Ctrl+Enter***. И будьте аккуратны с пробелами в синтаксисе, иначе Visual Studio начнет выдавать ошибки при компиляции. Если все прошло успешно, то внедренный манифест можно легко увидеть, открыв откомпилированный DLL-файл при помощи Visual Studio (***не доступно*** в Visual Studio Express, альтернатива: посмотреть HEX-редактором):



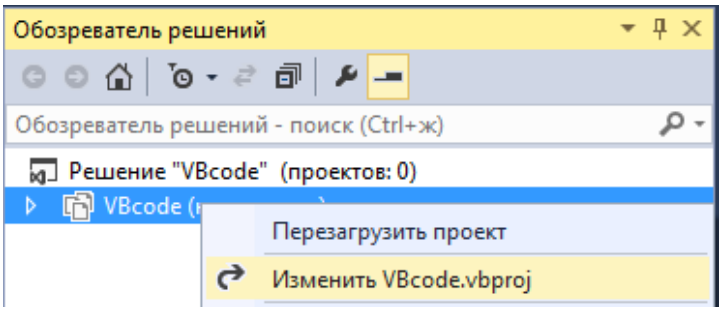
Внедрение манифеста на Visual Studio Express для VB.NET

VB.NET, в отличии от C#, не имеет возможности настройки ***обработки событий сборки*** на Visual Studio Express через пользовательский интерфейс. Настроить внедрение манифеста, скорее всего, придется только один раз за время работы с проектом.

Ограничение функциональности ***обработки событий сборки***, решается следующим образом. Выгружаем проект через контекстное меню «Обозревателя решений»:



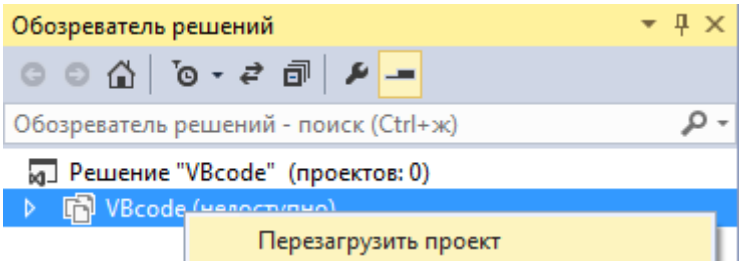
Далее здесь же загружаем проект на изменение:



Добавляем следующие данные в конец файла (выделено желтым фоном):

```
 </ItemGroup>
  <Import Project="$(MSBuildToolsPath)\Microsoft.VisualBasic.targets" />
  <PropertyGroup>
    <PostBuildEvent>call "%25VS120COMNTOOLS%25vsvars32"
mt.exe -manifest "$(ProjectDir)VBcode.manifest" -outputresource:"$(TargetPath)";#2
</PostBuildEvent>
  </PropertyGroup>
  <!-- To modify your build process, add your task inside one of the targets below and
uncomment it.
       Other similar extension points exist, see Microsoft.Common.targets.
  <Target Name="BeforeBuild">
  </Target>
  <Target Name="AfterBuild">
  </Target>
  -->
</Project>
```

Сохраняем и перезагружаем проект, через контекстное меню обозревателя решений:



Т.к. в Visual Studio Express просмотр внедренного манифеста после компиляции не доступен, то это можно сделать при помощи какого ни будь HEX-редактора

```
00013210  6E 00 00 00 31 00 2E 00  30 00 2E 00 30 00 2E 00  n...1...0...0...
00013220  30 00 00 00 3C 61 73 73  65 6D 62 6C 79 20 78 6D  0...<assembly xm
00013230  6C 6E 73 3D 22 75 72 6E  3A 73 63 68 65 6D 61 73  lns="urn:schemas
00013240  2D 6D 69 63 72 6F 73 6F  66 74 2D 63 6F 6D 3A 61  -microsoft-com:a
00013250  73 6D 2E 76 31 22 20 6D  61 6E 69 66 65 73 74 56  sm.v1" manifestV
00013260  65 72 73 69 6F 6E 3D 22  31 2E 30 22 3E 0D 0A 20  ersion="1.0">..
00013270  20 3C 61 73 73 65 6D 62  6C 79 49 64 65 6E 74 69  <assemblyIdenti
00013280  74 79 20 6E 61 6D 65 3D  22 56 42 63 6F 64 65 22  ty name="VBcode"
00013290  20 76 65 72 73 69 6F 6E  3D 22 31 2E 30 2E 30 2E  version="1.0.0.
000132a0  30 22 3E 3C 2F 61 73 73  65 6D 62 6C 79 49 64 65  0"></assemblyIde
000132b0  6E 74 69 74 79 3E 0D 0A  20 20 3C 63 6C 72 43 6C  ntity>..  <clrCl
000132c0  61 73 73 20 63 6C 73 69  64 3D 22 7B 33 63 33 31  ass clsid="{3c31
000132d0  36 37 66 61 2D 64 34 31  33 2D 34 30 64 39 2D 39  67fa-d413-40d9-9
000132e0  39 38 65 2D 34 33 61 62  30 36 62 66 63 35 33 65  98e-43ab06bfc53e
000132f0  7D 22 20 70 72 6F 67 69  64 3D 22 56 42 63 6F 64  }" progid="VBcod
00013300  65 2E 56 42 63 6F 64 65  2E 41 64 64 69 6E 53 65  e.VBcode.AddinSe
00013310  72 76 65 72 22 20 74 68  72 65 61 64 69 6E 67 4D  rver" threadingM
00013320  6F 64 65 6C 3D 22 42 6F  74 68 22 20 6E 61 6D 65  odel="Both" name
00013330  3D 22 56 42 63 6F 64 65  2E 56 42 63 6F 64 65 2E  ="VBcode.VBcode.
00013340  41 64 64 69 6E 53 65 72  76 65 72 22 20 72 75 6E  AddinServer" run
00013350  74 69 6D 65 56 65 72 73  69 6F 6E 3D 22 22 3E 3C  timeVersion=""><
00013360  2F 63 6C 72 43 6C 61 73  73 3E 0D 0A 20 20 3C 66  /clrClass>..  <f
00013370  69 6C 65 20 6E 61 6D 65  3D 22 56 42 63 6F 64 65  ile name="VBcode
00013380  2E 64 6C 6C 22 20 68 61  73 68 61 6C 67 3D 22 53  .dll" hashalg="S
00013390  48 41 31 22 3E 3C 2F 66  69 6C 65 3E 0D 0A 3C 2F  HA1"></file>..</
000133a0  61 73 73 65 6D 62 6C 79  3E 50 41 44 50 41 44 44  assembly>PADPADD
000133b0  49 4E 47 58 58 50 41 44  44 49 4E 47 50 41 44 44  INGXXPADDINGPADD
```

Файл описания подключаемого AddIn (*.addin*)

Для подключения, созданного AddIn по технологии ***«Registry free»*** необходимо дать понять Inventor, где искать DLL-файл. Для этого необходимо создать в проекте еще один файл в формате xml, но имеющего расширение: ***.addin*** . Этот файл тоже своего рода манифест, но что бы не путаться, термин манифест для ***.addin***-файла употребляться не будет.

Назовем его, к примеру, для VB.NET: VB.AddIn.addin, а для C#: cs.AddIn.addin.

Внесем в эти файлы следующую информацию:

VB.NET:

```
<?xml version="1.0" encoding="utf-8" ?>
<Addin Type="Standard">
  <!--Created for Autodesk Inventor Version 17.0-->
  <ClassId>{3c3167fa-d413-40d9-998e-43ab06bfc53e}</ClassId>
  <ClientId>{3c3167fa-d413-40d9-998e-43ab06bfc53e}</ClientId>
  <DisplayName>VB.NET AddIn</DisplayName>
  <Description>Мой VB.NET AddIn</Description>
  <Assembly>D:\VSProjects\VBcode\VBcode\bin\Debug\VBcode.dll</Assembly>
  <OSType>Win64</OSType>
  <LoadAutomatically>1</LoadAutomatically>
  <UserUnloadable>1</UserUnloadable>
  <Hidden>0</Hidden>
  <SupportedSoftwareVersionGreaterThan>16..</SupportedSoftwareVersionGreaterThan>
  <DataVersion>1</DataVersion>
  <LoadBehavior>0</LoadBehavior>
  <UserInterfaceVersion>1</UserInterfaceVersion>
</Addin>
```

C#:

```
<?xml version="1.0" encoding="utf-8" ?>
<Addin Type="Standard">
  <!--Created for Autodesk Inventor Version 17.0-->
  <ClassId>{19b41d23-3826-4fdf-974f-630672ae3488}</ClassId>
  <ClientId>{19b41d23-3826-4fdf-974f-630672ae3488}</ClientId>
  <DisplayName>cs AddIn</DisplayName>
  <Description>Мой C# AddIn</Description>
  <Assembly>D:\VSProjects\CScode\CScode\bin\Debug\CScode.dll</Assembly>
  <OSType>Win64</OSType>
  <LoadAutomatically>1</LoadAutomatically>
  <UserUnloadable>1</UserUnloadable>
  <Hidden>0</Hidden>
  <SupportedSoftwareVersionGreaterThan>16..</SupportedSoftwareVersionGreaterThan>
  <DataVersion>1</DataVersion>
  <LoadBehavior>0</LoadBehavior>
  <UserInterfaceVersion>1</UserInterfaceVersion>
</Addin>
```

Приведем описание использованных тэгов:

- `<ClassId>` (обязательный) - это уникальный 128-ми битный идентификатор GUID, который был прописан в атрибуте класса ***GuidAttribute***.
- `<ClientId>` (обязательный) - это уникальный 128-ми битный идентификатор GUID, который идентифицирует владельца-создателя панелей с кнопками, инструментальных панелей и пр. Это актуально если какой ни будь другой AddIn будет использовать ваши инструменты, в этом случае ***ClientId*** должен быть не изменным. Почти во всех случаях это идентификатор такой же, как ***ClassId***.

- <DisplayName> (обязательный) – название AddIn которое будет видеть пользователь в Inventor в окне управления надстройками. Без дополнительных атрибутов всегда должно быть только на английском языке иначе AddIn не подключится. Для подключения русского языка необходимо добавить атрибут ***Language*** в тэг, например:
```
<DisplayName Language="1033">My test</DisplayName>
<DisplayName Language="1049">Мой тест</DisplayName>
```
- <Assembly> (обязательный) – здесь прописывается полный путь к DLL-файлу. Однако не всегда нужно указывать полный путь, иногда это может быть относительный путь. Одно из таких мест это папка, где установлен Inventor, например: ***… Inventor 2015\bin***
Но устраивать «свалку» в папках Inventor это пример не самой хорошей культуры. Поэтому существует еще несколько мест, где можно прописать относительный путь к DLL-файлу. В случаях приведенных ниже Inventor будет получать путь к DLL-файлу относительно найденного ***.addin***-файла.
  1. Для всех пользователей не зависимо от версии Inventor

     %ALLUSERSPROFILE%\Autodesk\Inventor Addins
  2. Для всех пользователей в зависимости от версии Inventor

     %ALLUSERSPROFILE%\Autodesk\Inventor 2015\Addins
  3. Для конкретного пользователя в зависимости от версии Inventor

     %APPDATA%\Autodesk\Inventor 2015\Addins
  4. Для конкретного пользователя не зависимо от версии Inventor

     %APPDATA%\Autodesk\ApplicationPlugins

  т.е. если положить ***.addin***-файл, а рядом с ним DLL-файл, то тэг с путь этому DLL-файлу будет выглядеть просто, например:
```
<Assembly>VBcode.dll</Assembly>
```
  Если создать подпапку MyAddIn и поместить туда DLL-файл, то относительный путь будет выглядеть так:
```
<Assembly>MyAddIn\VBcode.dll</Assembly>
```
  Использование относительных путей, обычно удобнее при разворовывании AddIn на компьютере конечного пользователя. В данном случае при отладке, удобнее прописать путь к месту компиляции DLL-файла, как показано на моем примере, у вас этот путь может быть другим.
- <OSType> (необязательный) – определяет разрядность Windows 64 или 32 бита. Правильные значения для этого тега: ***Win32*** или ***Win64***. Если значение этого тэга не установлено, то предполагается что данный AddIn может работать для двух разрядностей.
- <LoadAutomatically> (необязательный) – определяет будет ли данный AddIn загружен автоматически при запуске Inventor. Значения могут быть либо ***1*** либо ***0***. Если значение равно ***1***, то AddIn запускается автоматически в соответствии с остальными настройками. Если значение равно ***0***, то AddIn можно будет запустить в ручную, через менеджер настроек в Inventor. Если данный тэг не определен, то умолчанию значение будет рано ***1***.
- <UserUnloadable> (необязательный) – определяет, будет ли позволено пользователю включать/отключать AddIn через менеджер надстроек во время работы Inventor. Значение может быть либо ***1*** либо ***0***. Если значение равно ***0***, то пользователь не может включать/отключать AddIn через менеджер надстроек.

Если данный тэг не определен, то значение по умолчанию равно ***1***.

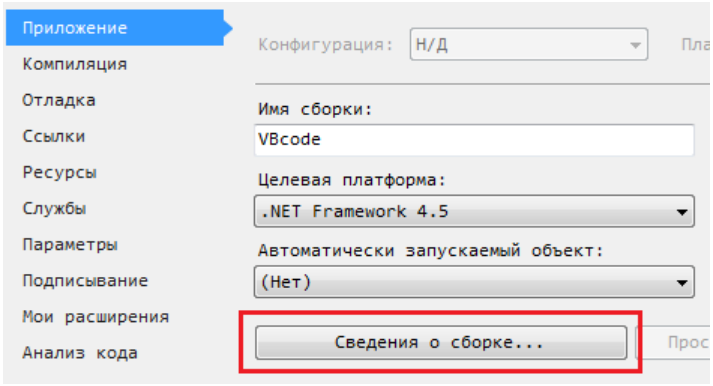
- <Hidden> (необязательный) – определяет, будет ли AddIn показан в списке менеджера надстроек. Значение может быть либо ***0*** либо ***1***. Если данный тэг не определен, то по умолчанию значение равно ***0***.
- Следующий необязательный тег может быть одним из четырех вариантов.
  <SupportedSoftwareVersionEqualTo> -AddIn работает только с заданной версией Inventor
  <SupportedSoftwareVersionGreaterThan> - AddIn работает только с версией Inventor выше заданной
  <SupportedSoftwareVersionLessThan> - AddIn работает только с версией Inventor не выше заданной
  <SupportedSoftwareVersionNotEqualTo > - AddIn работает только с версией Inventor не равной заданной.
  После задания значения номера версии обязательно поставить две точки, они означают, что AddIn будет применяться ко всем сервис-пакам, поставленным на данную версию Inventor.
  Этот тэг игнорируется если ***.addin***-файл положен в папку с учетом версии Inventor.
- <DataVersion> (необязательный) – этот параметр определяет версию AddIn для случая, если файлы документов Inventor хранят в себе данные от этого AddIn. По этому параметру определяется необходимость миграции (обновления до текущей версии), хранящихся в документах Inventor данных от этого AddIn. Индикация необходимости миграции данных будет показываться по объекту ***DocumentInterests*** из объектной модели API Inventor.
- <LoadBehavior> (необязательный) – этот параметр определяет, когда данный AddIn должен быть загружен:
  0 – загрузка немедленно после загрузки Inventor (не рекомендуется т.к. увеличивает время загрузки самого Inventor)
  1 – загрузка после открытия любого документа
  2 – загрузка поле открытия сборки
  3 - загрузка поле открытия презентации
  4 - загрузка поле открытия чертежа
  10 –загрузка только по требованию, через API Inventor или в ручную с помощью диспетчера надстроек.
  Если значение тэга не определено, то по умолчанию значение будет равно ***0***.
- <UserInterfaceVersion> (необязательный) – этот параметр определяет версию пользовательского интерфейса данного AddIn (панели, кнопки). При изменении этого параметра, все элементы пользовательского интерфейса (панели, кнопки) данного AddIn будут очищены при последующей загрузки Inventor.

Созданный и настроенный ***.addin***-файл необходимо положить в один из рекомендованных путей. И даже если DLL-файла еще нет, то после помещения ***.addin***-файла в одну из папок, где Inventor ищет надстройки, в ***«Менеджере надстроек»*** Inventor объявленный AddIn все равно появится в списке надстроек, но будет отключен.

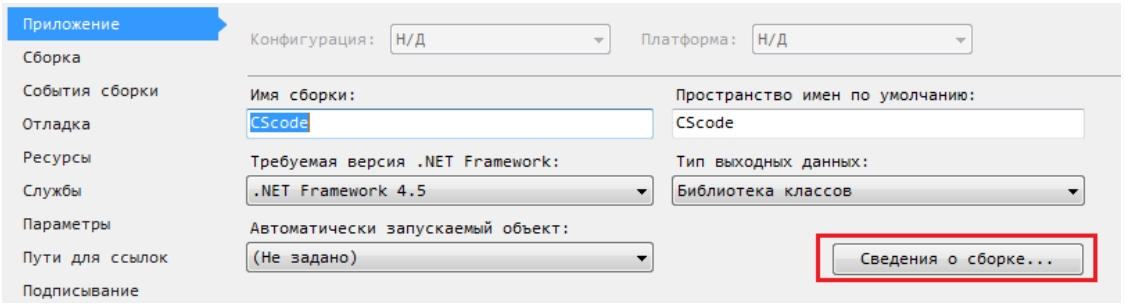
Настройки: «Видимость сборки для COM» и номер Framework

Перед первым тестированием AddIn, необходимо познакомится еще с парой настроек проекта. Т.к. работа API Inventor, все-таки, основана на COM-технологии, то необходимо сделать компилируемый DLL-файл видимый для COM-программ. Эта настройка находится в свойствах проекта:

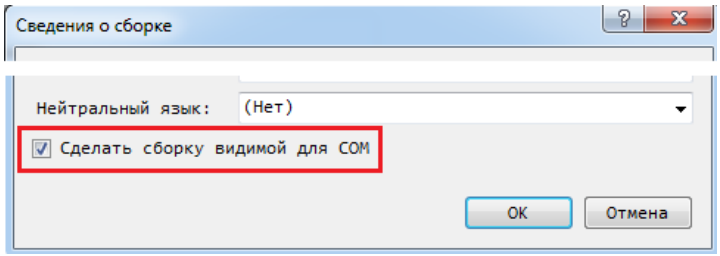
VB.NET:



C#:



В появившемся окне необходимо включить опцию: ***«Сделать сборку видимой для COM»***



Так же необходимо соблюдать правило: установленная при компиляции версия Framework, должна быть не новее прописанной версии Framework в специальном конфигурационном файле Inventor: ***Inventor.exe.config***

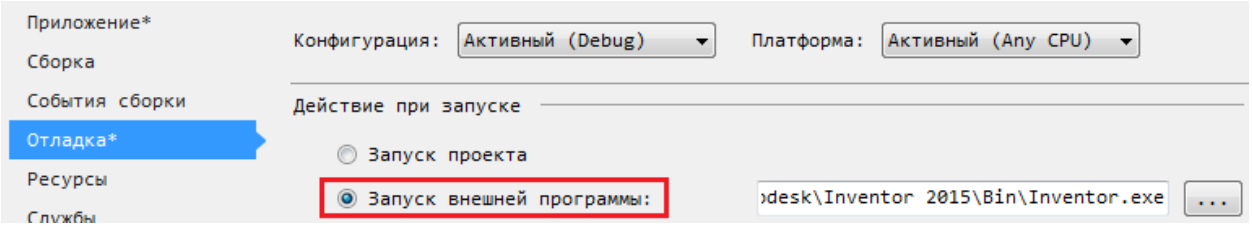
Файл ***Inventor.exe.config*** находится в папке «Bin» в месте, где установлен сам Inventor. Ниже приведен фрагмент из конфигурационного файла для Inventor 2015:

```
<startup useLegacyV2RuntimeActivationPolicy="true">
     <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"/>
</startup>
```

В данном примере номера Framework проекта и Inventor совпадают, но в случае если версии Framework не совпадают, то следует подобрать соответствующую версию Framework в данных настройках проекта в Visual Studio.

Запуск DLL в режиме DEBUG

Итак, мы подошли к завершающей стадии перед запуском AddIn на отладку. Для этого осталось прописать ***«Запуск внешней программы»*** в настройке проекта (для Visual Studio Express см. *Отладка Visual Studio и Visual Studio Express через «Запуск»*):



Устанавливаем точку останова в коде программы:

VB.NET:

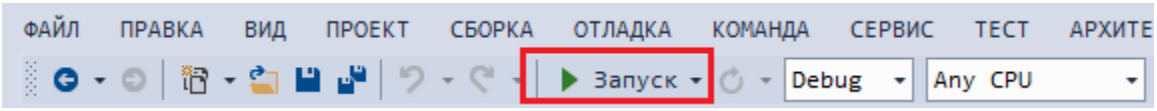
```
1  Imports System.Runtime.InteropServices
2  Namespace VBcode
3      <GuidAttribute("3c3167fa-d413-40d9-998e-43ab06bfc53e")> _
       ссылок 0
4      Public Class AddinServer
5          Implements Inventor.ApplicationAddInServer
6          Private InvApp As Inventor.Application
           ссылок 0
7          Public Sub Activate(AddInSiteObject As Inventor.ApplicationAddInSite, FirstTime As Boolean) _
8              Implements Inventor.ApplicationAddInServer.Activate
9              InvApp = AddInSiteObject.Application
10             System.Windows.Forms.MessageBox.Show("VB.NET:Здравствуй " & InvApp.Caption)
```

C#:

```
using System.Runtime.InteropServices;
namespace CScode
{
    [GuidAttribute("19b41d23-3826-4fdf-974f-630672ae3488")]
    ссылок 0
    public class AddInServer : Inventor.ApplicationAddInServer
    {
        private Inventor.Application InvApp;
        ссылок 0
        public void Activate(Inventor.ApplicationAddInSite AddInSiteObject, bool FirstTime)
        {
            InvApp = AddInSiteObject.Application;
            System.Windows.Forms.MessageBox.Show("C#: Здравствуй " + InvApp.Caption);
```

Компилируем проект в режиме Debug в Visual Studio.

Еще раз убеждаемся что ***.addin***-файл находится там, где надо и Inventor его сможет увидеть. Проверяем, что путь в ***.addin***-файле корректно указывает на только, что откомпилированный DLL-файл. И нажимаем в Visual Studio кнопку «Запуск»:



При этом должен начаться загружаться Inventor, и через некоторое время должен произойти останов его загрузки и в Visual Studio исполнение программы будет ожидать на точке останова, здесь уже можно начинать манипуляции с отладкой, просматривать состояние объектов и пр.

VB.NET:

```
ссылок 0
Public Sub Activate(AddInSiteObject As Inventor.ApplicationAddInSite, FirstTime As Boolean) _
    Implements Inventor.ApplicationAddInServer.Activate
    InvApp = AddInSiteObject.Application
    System.Windows.Forms.MessageBox.Show("VB.NET:Здравствуй " & InvApp.Caption)
End Sub
```

Локальные

Имя	Значение
Me	{ClassLibrary1.ClassLibrary1.AddinServer}
AddInSiteObject	{Inventor.ApplicationAddInSite}
FirstTime	True

C#:

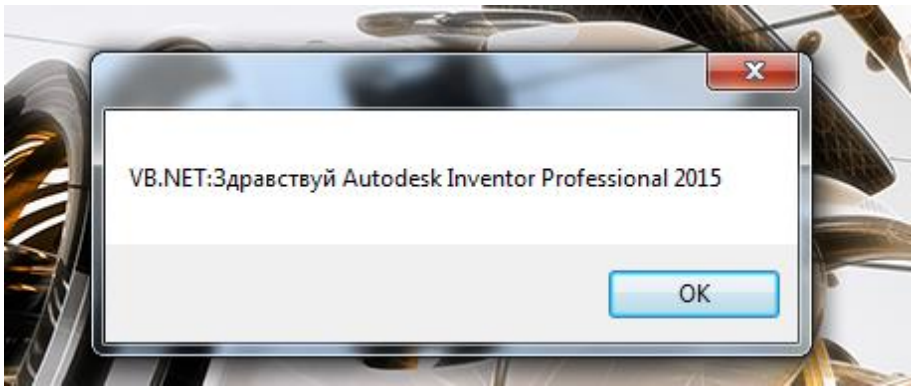
```
private Inventor.Application InvApp;
ссылок 0
public void Activate(Inventor.ApplicationAddInSite AddInSiteObject, bool FirstTime)
{
    InvApp = AddInSiteObject.Application;
    System.Windows.Forms.MessageBox.Show("C#: Здравствуй " + InvApp.Caption);
```

Локальные

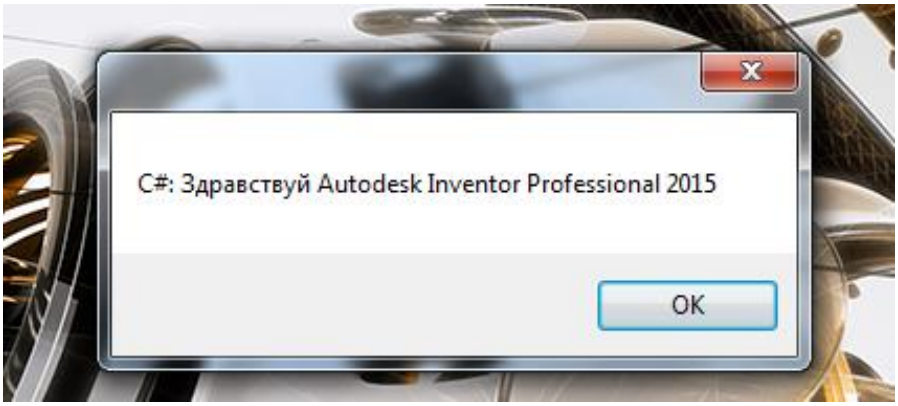
Имя	Значение
this	{ClassLibrary2.AddInServer}
AddInSiteObject	COM-объект
FirstTime	true

При выполнении следующего шага (пошаговое выполнение) появится окно с приветствием:

VB.NET:



C#:



Одно замечание, если у вас к Inventor подключено несколько ваших AddIn, а также в Visual Studio имеется программный код на эти AddIn в режиме отладки, то у меня появлялась ситуация, что Visual Studio начинала отлаживать соседние мои собственные AddIn, хотя соседние программные проекты не были открыты. Так, чтобы не было такого неудобства, не используемые собственные AddIn лучше отключить на время отладки конкретного AddIn.

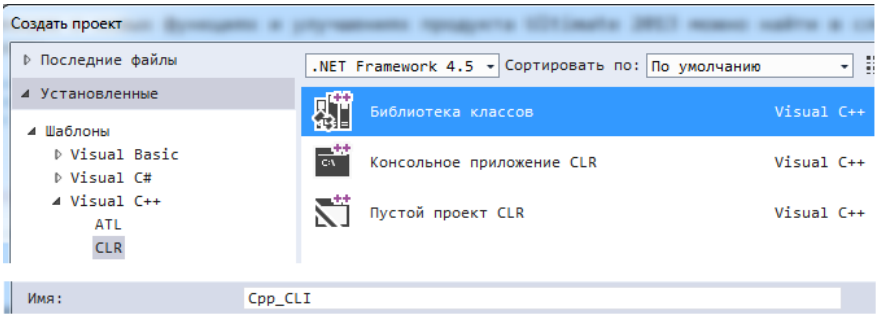
Готовые примеры проектов на VB.NET и C#, которые мы так подробно рассмотрели, можно скачать здесь (для VS2013):

AddIn для C++/CLI

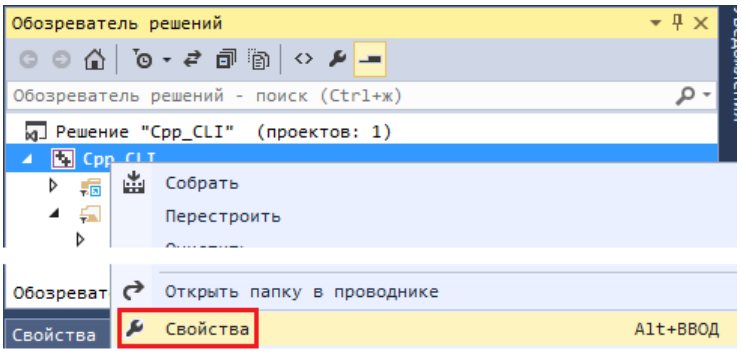
Язык C++/CLI является разработкой Microsoft и включается в поставку Visual Studio. C++/CLI является комбинацией «нативного» C++ (классический C++) и «управляемого» CLI (платформа NET). В результате чего C++/CLI может легко работать как NET-объектами из Framework, так и с бинарными DLL. В результате чего C++/CLI является достаточно универсальным языком и если бы не действия Microsoft «по сдерживанию» развития C++/CLI, то думаю, что C++/CLI составил бы серьезную конкуренцию C#. «Сдерживание» C++/CLI Microsoft-ом, заключается в урезании различных визуальных конструкторов на технологии WPF, удалении готовых шаблонов Windows Forms, не такой дружественный IntelliSense, низкий приоритет исправления багов в C++/CLI во всем, что связано с аналогичным инструментарием в C# и пр. Хотя возможность всего функционала C# в C++/CLI остается, но только за частую реализация этого функционала производится «в ручную». По сути, в последнее время Microsoft позиционирует C++/CLI как язык для легкой «склейки» «нативного» и «управляемого» кода.

С учетом того, что некоторые вещи в Inventor при помощи VB.NET и C# трудноосуществимы (чаще всего взаимодействие с «нативными» DLL Windows), то, все-таки, C++/CLI представляет определенный интерес особенно для AddIn более профессионального уровня. Т.к. C++/CLI не сложен и очень похож на C# по синтаксису, то разберем пример создание AddIn на C++/CLI.

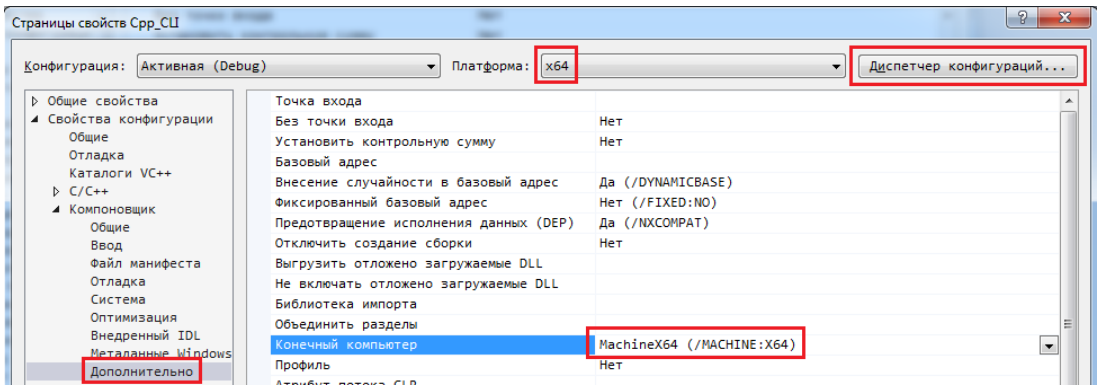
Создаем новый проект, назовем его Cpp_CLI:



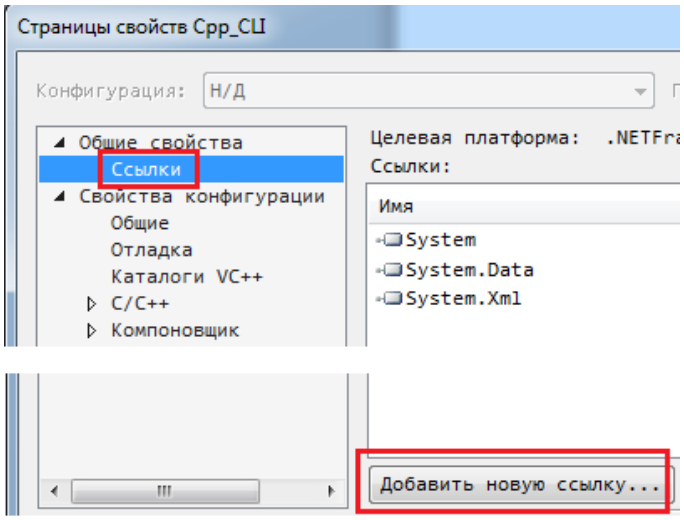
В «Обозревателе решений» вызовем страницу свойств проекта:



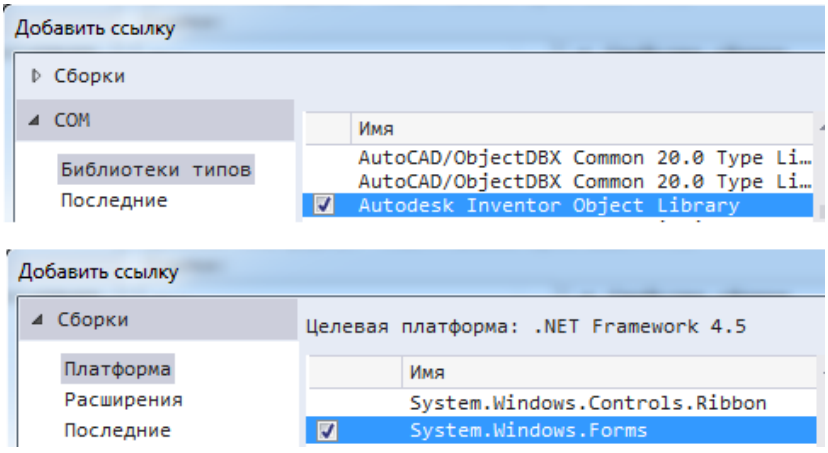
Переключим проект на 64-ех битную компиляцию, обычно для это достаточно переключится в ***«Диспетчере конфигураций»***, но также можно найти переключение в настройках проекта:



Далее вызовем окно для подключения дополнительных библиотек:



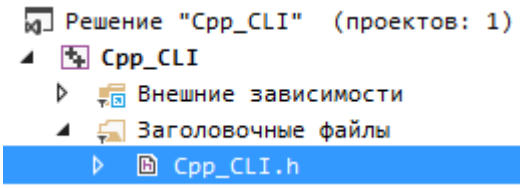
Подключим ссылки на объектные библиотеки ***Autodesk Inventor Object Library*** и ***System.Windows.Forms***:



Теперь необходимо реализовать интерфейс ***Inventor::ApplicationAddInServer***, но сначала пару слов, про особенность организации файлов C++/CLI. Т.к. C++/CLI это все таки модифицированный C++. Поэтому в нем, так же как в классическом C++, объявление объектов отделено от их реализации, и объект-класс состоит из двух фалов: заголовочного файла (расширение ***.h***) и файла исходного кода (расширение ***.cpp***). В заголовочном файле обычно происходит объявления различных функций, переменных и пр. В файле ***.cpp*** находится реализованные объекты. Это удобно при командной работе, потому как, кто то «конструирует»

классы, а кто то параллельно может их реализовывать. Аналогичный подход есть и в C#, в работе через абстрактные классы.

Поэтому сначала откроем на редактирование файл ***Cpp_CLI.h***:

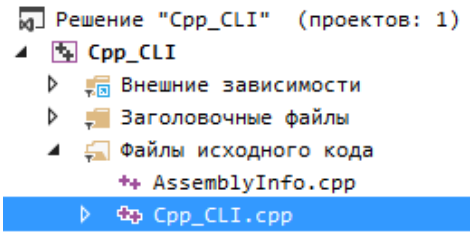


И оформим объявление класса ***AddInServer***:

```
// C++_CLI.h
#pragma once
using namespace System;
using namespace System::Runtime::InteropServices;

namespace Cpp_CLI
{
	[GuidAttribute("31D3DD15-654F-46A4-83E1-281D9D819EB2")]
	public ref class AddinServer : Inventor::ApplicationAddInServer
	{
	private:
		Inventor::Application^ InvApp;
	public:
		virtual void Activate(Inventor::ApplicationAddInSite^ invApp, bool firstTime);
		virtual void Deactivate();
		virtual void ExecuteCommand(int CommandID);

		virtual property Object^  Automation
		{
			Object^ get();
		}
	};
}
```

IntelliSense не особо помогает в C++/CLI создать список реализуемых объектов интерфейса, по крайней мере в Visual Studio 2013. Поэтому для удобства можно просматривать аналогичный код в C#, т.к. синтаксис очень похож и IntelliSense в C# работает хорошо, и далее конвертировать код в C++/CLI, вместо того, что бы изучать ***«Обозреватель объектов»***.

Далее переходим к файлу исходного кода через «Обозреватель решений»:



Но т.к. «перепрыгивать» между файлами ***.h*** и ***.cpp*** может понадобиться часто, то лучше привыкнуть к комбинации горячих клавиш: ***Ctrl+K+O***.

Реализация будет выглядеть следующим образом:

```
#include "stdafx.h"
#include "Cpp_CLI.h"

namespace Cpp_CLI
{
	void AddinServer::Activate(Inventor::ApplicationAddInSite^ invApp, bool firstTime)
	{
		AddinServer::InvApp = invApp->Application;
		System::Windows::Forms::MessageBox::Show(L"C++/CLI : Здравствуй " +
							AddinServer::InvApp->Caption);
	}

	void AddinServer::Deactivate() {}

	void AddinServer::ExecuteCommand(int CommandID){}

	Object^ AddinServer::Automation::get()
	{
		return nullptr;
	}
}
```

Язык C++/CLI концептуально отличается от C#, C++/CLI почти все построено на работе с указателями, в то в то время как в C# указатели («не безопасный» код) почти не используются. В VB.NET вообще нет такого понятия как указатель. Объект:

Inventor::ApplicationAddInSite^ invApp

в сигнатуре функции ***AddinServer::Activate***, является указателем, но «управляемым» указателем NET-типа, поэтому проблем из-за отсутствия корректного удаления такого указателя в программе не возникнет. Для доступа к объектам применяется синтаксис разъименования указателя

invApp->Application

Манифест и *.addin*-файл и остальные настройки

Итак, тестовый код создан, переходим к регистрации AddIn в стиле ***«Registry free»***. Для этого создаем манифест, делаем, так же как и ранее, файл в формате xml, переименовываем этот файл и меняем ему расширение: ***Cpp_CLI.X.manifest***, добавляем в содержимое файла:

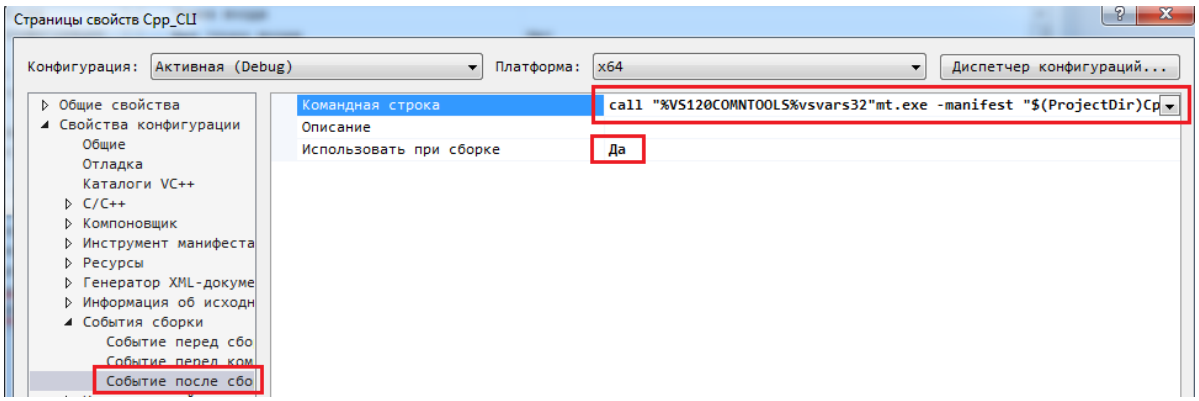
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity name="Cpp_CLI" version="1.0.0.0" />
  <clrClass clsid="{31D3DD15-654F-46A4-83E1-281D9D819EB2}"
            progid="Cpp_CLI.AddinServer"
            threadingModel="Both"
            name="Cpp_CLI.AddinServer"
            runtimeVersion="" />
  <file name="Cpp_CLI.dll" hashalg="SHA1" />
</assembly>
```

Обратите внимание, что стиль организации пространств имен в C++/CLI и C# полностью совпадает.

Как я писал выше, что файл с таким форматом имени:

Имя DLL-файла.X.manifest

в C++/CLI будет внедряться автоматически, без дополнительных команд в ***«Событиях построения»***, но можно, при желании, прописать внедрение манифеста, как мы делали ранее через настройки проекта:



Текст в командной строке должен быть аналогичный, как и для C#:

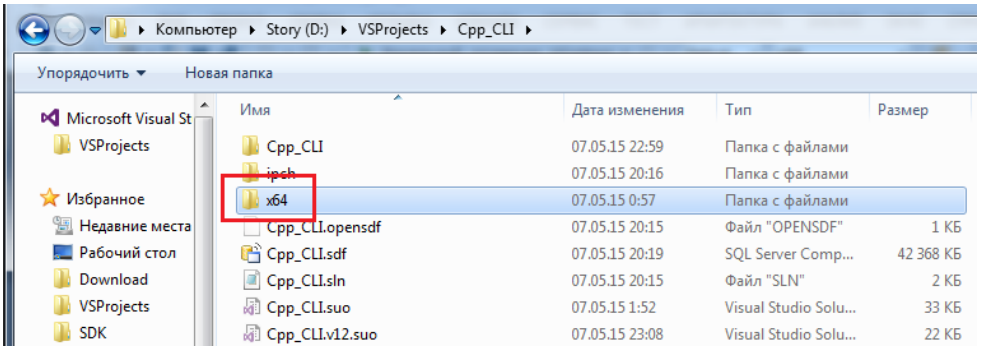
call "%VS120COMNTOOLS%vsvars32"

mt.exe -manifest "$(ProjectDir)Cpp_CLI.X.manifest" -outputresource:"$(TargetPath)";#2

Теперь создаем ***.addin***-файл, все сказанное ранее про этот файл полностью актуально и для данного примера. Текст ***addin***-файла:

```
<?xml version="1.0" encoding="utf-8" ?>
<Addin Type="Standard">
  <!--Created for Autodesk Inventor Version 17.0-->
  <ClassId>{31D3DD15-654F-46A4-83E1-281D9D819EB2}</ClassId>
  <ClientId>{31D3DD15-654F-46A4-83E1-281D9D819EB2}</ClientId>
  <DisplayName>Cpp AddIn</DisplayName>
  <Description>Мой C++/CLI AddIn</Description>
  <Assembly>D:\VSProjects\Cpp_CLI\x64\Debug\Cpp_CLI.dll</Assembly>
  <OSType>Win64</OSType>
  <LoadAutomatically>1</LoadAutomatically>
  <UserUnloadable>1</UserUnloadable>
  <Hidden>0</Hidden>
  <SupportedSoftwareVersionGreaterThan>16..</SupportedSoftwareVersionGreaterThan>
  <DataVersion>1</DataVersion>
  <LoadBehavior>0</LoadBehavior>
  <UserInterfaceVersion>1</UserInterfaceVersion>
</Addin>
```

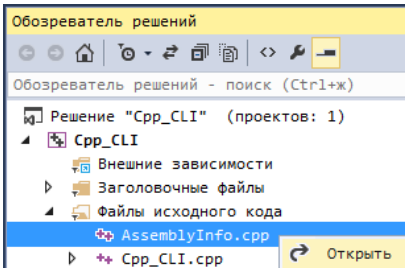
Компилируем DLL-файл. Обратите внимание, что C++/CLI, компилирует DLL-файл не в папку с проектом, а немного выше по файловой иерархии, в папке с решением.



Перемещаем ***addin***-файл в одну из папок, где Inventor будет производить поиск AddIn. Напомню эти папки:

1. Для всех пользователей не зависимо от версии Inventor
%ALLUSERSPROFILE%\Autodesk\Inventor Addins
2. Для всех пользователей в зависимости от версии Inventor
%ALLUSERSPROFILE%\Autodesk\Inventor 2015\Addins
3. Для конкретного пользователя в зависимости от версии Inventor
%APPDATA%\Autodesk\Inventor 2015\Addins
4. Для конкретного пользователя не зависимо от версии Inventor
%APPDATA%\Autodesk\ApplicationPlugins

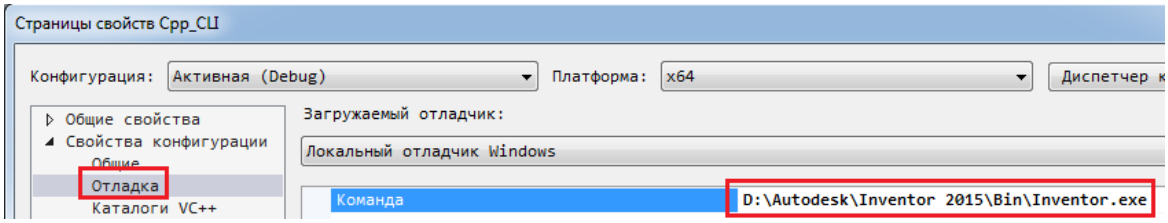
Открываем файл ***AssemblyInfo.cpp*** на редактирование:



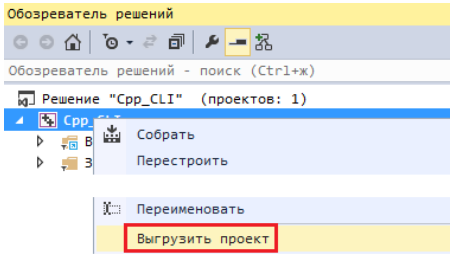
ищем атрибут ***ComVisible*** и менеям значение ему на ***true***:

```
[assembly:ComVisible(true)];
```

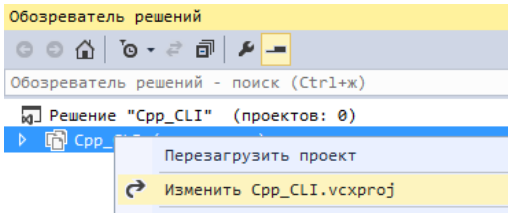
Опять идем в настройки проекта и прописываем путь к Inventor для выполнения отладки.



Проверяем назначенную версию Framework, для этого выгружаем проект, используя контекстное меню ***«Обозревателя решений»***:



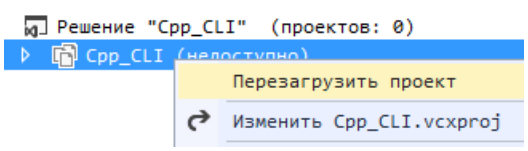
Здесь, то же через контекстное меню, загружаем файл проекта на редактирование:



Ищем тэг: ***TargetFrameworkVersion***:

```
<TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
```

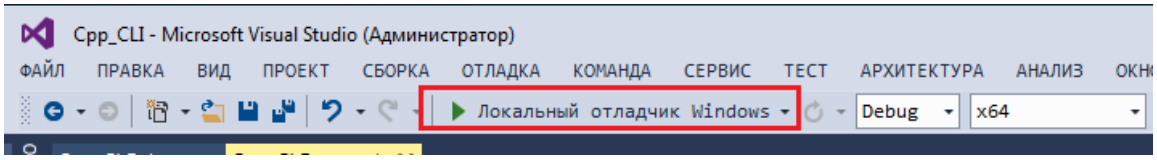
В нашем случае все нормально – версия Framework проекта совпадает с версией Framework в файле ***Inventor.exe.config***, о котором было написано выше. Закрываем файл проекта и загружаем проект снова в «Обозревателе решения» через контекстное меню:



Устанавливаем точку останова:

```
7   {
8       void AddinServer::Activate(Inventor::ApplicationAddInSite^ invApp, bool firstTime)
9       {
10          AddinServer::InvApp = invApp->Application;
11          System::Windows::Forms::MessageBox::Show(L"C++/CLI : Здравствуй " + AddinServer::InvApp->Caption);
12      }
```

Все готово для запуска C++/CLI проекта в режиме ***Debug***. Запускаем:



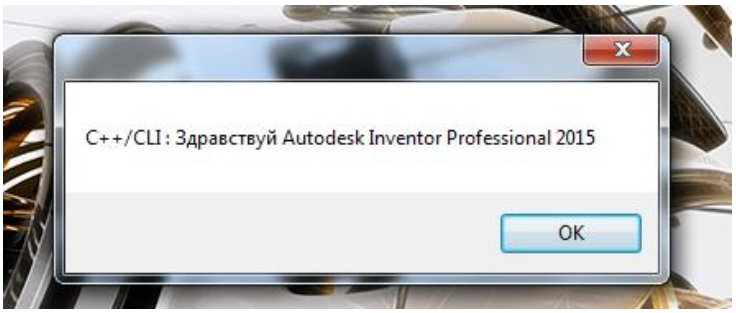
Программа останавливается в точке останова, где мы можем наблюдать значение объектов:

```
8       void AddinServer::Activate(Inventor::ApplicationAddInSite^ invApp, bool firstTime)
9       {
10          AddinServer::InvApp = invApp->Application;
11          System::Windows::Forms::MessageBox::Show(L"C++/CLI : Здравствуй " + AddinServer::InvApp->Caption);
12      }
```

Локальные

Имя	Значение
this	0x000000001444c788 { InvApp=0x0000000014101d48 }
invApp	0x00000000144742a8
firstTime	true

Дальнейшее продолжение работы программы вызовет появления окна:



Готовый пример можно скачать здесь

Смешанный C# и C++

В ряже случаев, таких как, повышение производительности, работа с библиотеками Windows, может возникнуть необходимость использовать «нативный» C++. Однако при использовании для AddIn «нативного» C++ в чистом виде возникают определенные затруднения:

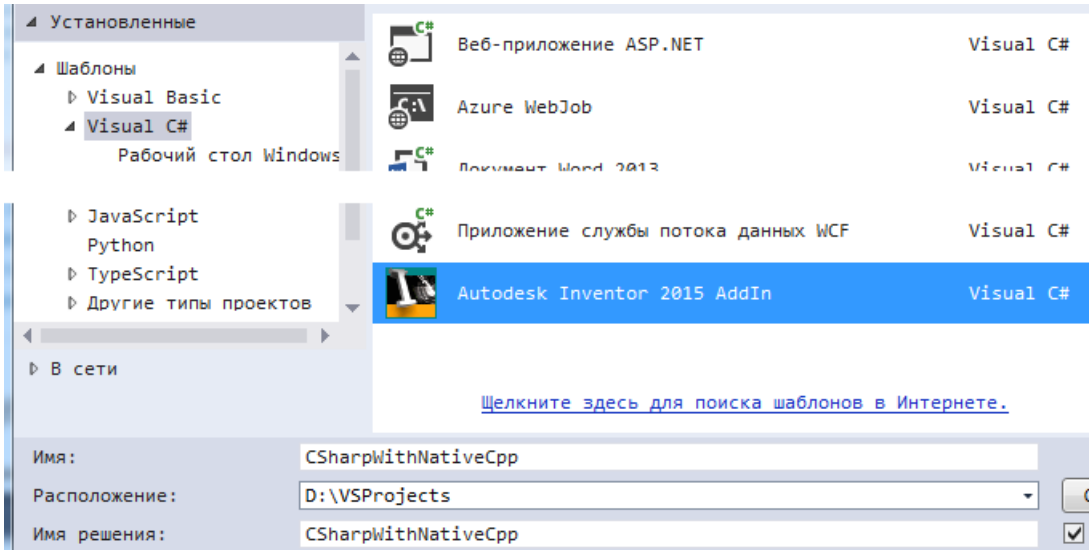
- Сложность в понимании всех тонкостей создания AddIn при использовании COM-технологии чистого «нативного» C++. Это все усугубляется заголовочными файлах Microsoft, которые вместо нормального программного кода, в большом количестве содержат «подстановочные макросы», что значительно затрудняет их понимание. Хотя без базового понимания COM-технологии программный код для работы с самим Inventor через «нативный» C++ создать не получится.
- Работа современными оконными технологиями, такими как WPF (Windows Presentation Foundation) и Windows Forms в «нативном» C++ напрямую никак не организована. Отсюда следует, что если есть необходимость создать объект типа DataGrid, то нужно искать либо подходящие COM-контролы, либо писать это все в ручную, либо использовать WInAPI, либо MFC (Microsoft Foundation Classes). Но это все врят ли вызовет восторг.

Можно было бы воспользоваться одним смешанным языком C++/CLI, но у него тоже есть недостатки, о которых я писал в предыдущем разделе. Большинство недостатков C++/CLI, конечно же решаемы. И вариант использование одного смешанного C++/CLI вполне работоспособен. Однако время, затраченное на написание программного кода, при работе только на одном C++/CLI может оказаться больше, чем при использовании связки C# и C++/CLI. Исходя из общепринятого подхода, что для уменьшения трудозатрат при решении проблем нужно бороться с «узкими местами». Для NET-языков «узким местом» может быть производительность скомпилированного кода и ограниченная функциональность, а для «нативного» C++ «узким местом», по сравнению с NET, является почти вся функциональность NET (по тому, как NET с самого начала и создавался как более высокоуровневый язык для упрощения программирования).

Поэтому рассмотрим создания решения, в котором вызов неуправляемых функций «нативного» C++ будет производиться из проекта, написанного на C#.

Создание основного проекта C# при помощи шаблона.

Приступим. Создадим новый проект в новом решении с названием ***CSharpWithNativeCpp***. Для этого будем использовать готовый шаблон для языка C#.

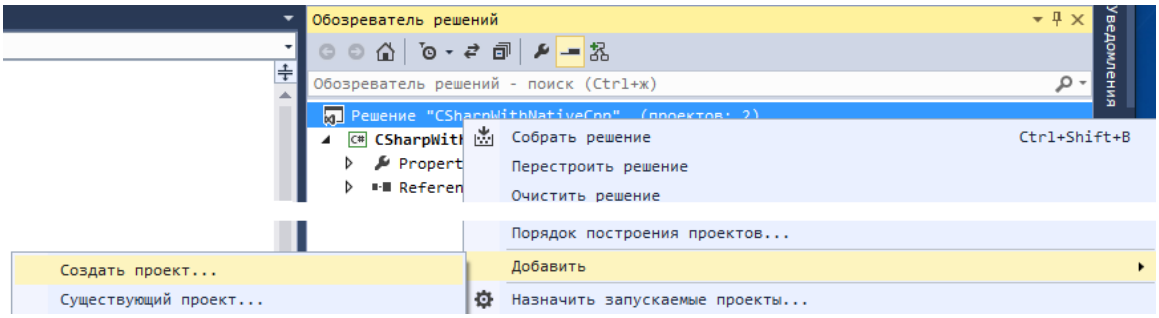


Этот C#-проект будет непосредственно подключаться к Inventor. В данном случае для корректного подключения к Inventor, достаточно модифицировать ***.addin***-файл: прописать в теге ***<Assembly>*** полный путь к откомпилированному DLL-файлу и положить этот ***.addin***-файл в папку, где Inventor сможет его найти и подключить. Напомню эти папки:

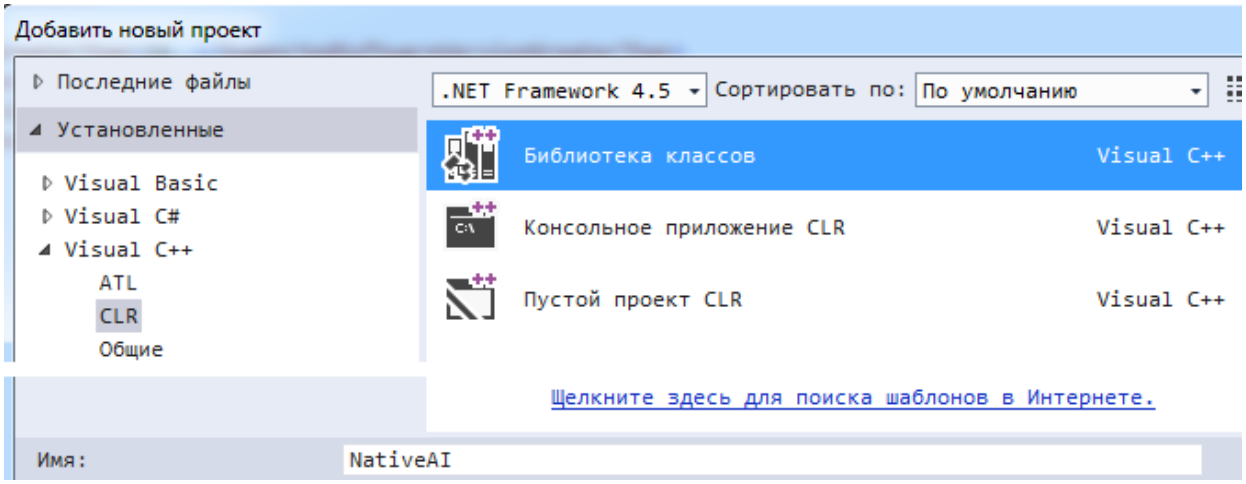
1. Для всех пользователей не зависимо от версии Inventor
   %ALLUSERSPROFILE%\Autodesk\Inventor Addins
2. Для всех пользователей в зависимости от версии Inventor
   %ALLUSERSPROFILE%\Autodesk\Inventor 2015\Addins
3. Для конкретного пользователя в зависимости от версии Inventor
   %APPDATA%\Autodesk\Inventor 2015\Addins
4. Для конкретного пользователя не зависимо от версии Inventor
   %APPDATA%\Autodesk\ApplicationPlugins.

Создание проекта на «нативном» C++

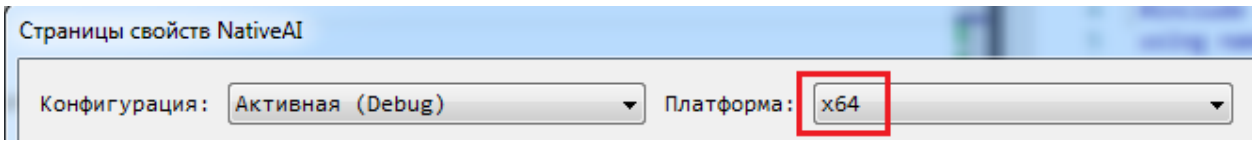
Далее создаем проект для «нативного» C++ программного кода. Для этого в «Обозревателе решений» вызовем контекстное меню с командой создания нового проекта:



Создавать проект для «нативного» C++ кода будем, используя библиотеку классов смешанного C++/CLI. Дадим имя этой библиотеке ***NativeAI***:



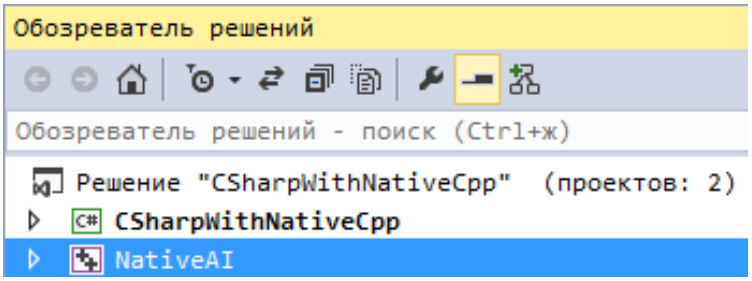
Переключим проект ***NativeAI*** на 64-ех битную компиляцию:



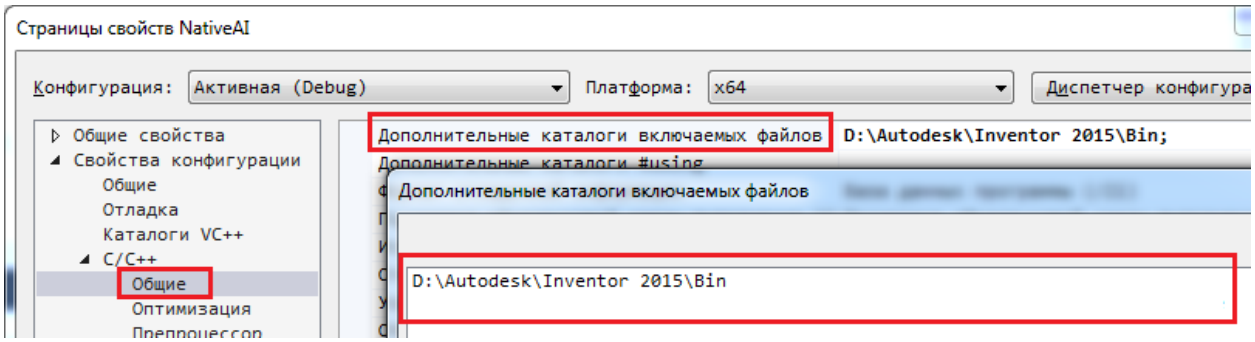
Легкого прямого доступа «управляемый» C# к «нативным» функциям C++ не имеет. Поэтому в данном проекте, «управляемый» код C++/CLI будет использоваться, только как обертка, для доступа из «управляемого» C# к «нативным» функциям C++.

Существует альтернативный вариант взаимодействия через маршализацию с «нативным» C++. Но на практике маршализация «нативного» кода это не очень удобная технология из-за не возможности отладки AddIn целиком, поэтому мы не будем её затрагивать.

И так после добавления проекта C++/CLI «Обозреватель решений» должен содержать в себе два проекта и выглядеть так:



Перед созданием «нативного» кода необходимо подключить библиотеки API Inventor. Процедура подключения библиотек API Inventor отличается от того что мы видели на «управляемых» языках (VB.NET, C# и C++/CLI). Вызовем ***«Страницы свойств»*** проекта NativeAI. Здесь необходимо прописать дополнительные пути поиска нужных нам файлов, это можно сделать в следующем месте:



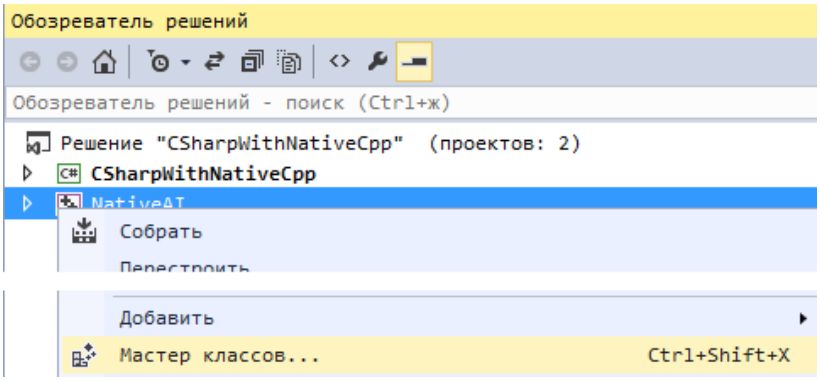
В данном случае у меня дополнительно папка, где установлен Inventor:

D:\Autodesk\Inventor 2015\Bin

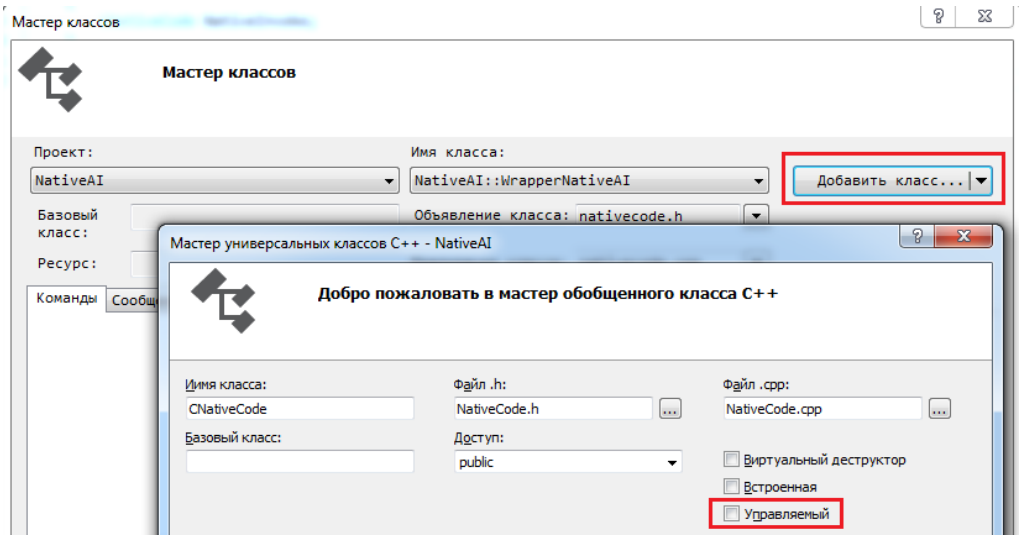
в ней находится файл библиотеки типов ***RxInventor.tlb***, в ***RxInventor.tlb*** находится описание всех объектов API Inventor.

В нашем простом примере мы обойдемся без заголовочных файлов из этой папки, поэтому пока её можно не подключать.

Т.к. файлы «управляемого» класса ***NativeAI.h*** и ***NativeAI.cpp*** должны были создаться автоматически, то нам необходимо теперь создать «нативный» класс. Можно создать их по отдельности, но удобнее воспользоваться ***«Мастером классов»***. Для этого в ***«Обозревателе решений»*** на названии проекта ***NativeAI*** вызовем контекстное меню:



В ***«Мастере классов»*** жмем кнопку «Добавить класс» и в появившемся окне вписываем имя будущего класса. Если опция «Управляемый» включена, то её необходимо отключить. В «нативном» C++ обычно принято, что бы все имена классов начинались с заглавной латинской буквы «C».



Теперь у нас есть заголовочный файл для «нативного» кода, откроем его на редактирование.

Подключение библиотеки типов Inventor в «нативном» C++

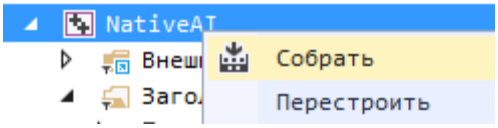
Далее можно импортировать библиотеки API Inventor, и добавить прототип функции вызова тестового окна. Так же добавим смарт-указатель на COM-объект ***pInvApp*** и библиотеку для работы со смарт-указателем ***<atlbase.h>***. В результате программный код заголовочного файла будет выглядеть так:

```
#pragma once
#include <atlbase.h>
#import <RxInventor.tlb> \
	rename_namespace("InventorNative") \
	named_guids raw_dispinterfaces \
	high_method_prefix("Method") \
	rename("DeleteFile", "APIDeleteFile") \
	rename("CopyFile", "APICopyFile") \
	rename("MoveFile", "APIMoveFile")  \
	rename("SetEnvironmentVariable", "APISetEnvironmentVariable") \
	rename("GetObject", "APIGetObject") \
	exclude("_FILETIME", "IStream","ISequentialStream", \
		"_LARGE_INTEGER", "_ULARGE_INTEGER", "tagSTATSTG", \
		 "IEnumUnknown","IPersistFile", "IPersist", "IPictureDisp")
using namespace InventorNative;
class CNativeCode
{
public:
	CNativeCode();//Конструктор
	~CNativeCode();//Деструктор
	void ShowMassage();//Функция вызова окна
private:
	CComPtr<Application> pInvApp;//указатель на объект Inventor
};
```

Здесь нужно сказать несколько слов по директиве ***#import***. После того, как эта директива будет добавлена, Visual Studio будет выдавать ошибку, сообщая, что отсутствует файл:

rxinventor.tlh

Это производный заголовочный файл, который сгенерируется автоматически после удачной компиляции проекта. Этот файл находится в одной из папок внутри проекта. Поэтому скомпилируем проект ***NativeAI***.



После чего ошибка в директиве ***#import*** должна исчезнуть. Атрибуты директивы ***#import*** позволяют произвести определенную настройку при импорте объектов API Inventor. Например:

- атрибут ***rename_namespace***, переименовывает пространство имен для импортированных объектов.
- Применение атрибута ***rename*** при работе с API Inventor является обязательным для указанных в примере имен функций, из-за возникающего конфликта имен с функциями из других библиотек.
- Атрибут ***exclude*** предотвращает импорт типов уже имеющихся в библиотеке от Microsoft ***<atlbase.h>.*** Хотя при совпадении импортированных типов с уже имеющимися типами, Visual Studio импорт таких типов игнорирует, но постоянно об этом сообщает. Если бы в примере использовались обычные указатели, и не было бы подключены библиотеки от Microsoft с описанием этих типов, то атрибут ***exclude*** нужно удалить, т.к. эти типы данных могут пригодиться.

В принципе если файл ***RxInventor.tlb*** не находится среди прописанных путей поиска файлов можно прописать абсолютный путь.

```
#import "D:\Autodesk\Inventor 2015\Bin\RxInventor.tlb" \
       rename_namespace("InventorNative") \
       named_guids raw_dispinterfaces \
       high_method_prefix("Method") \
       rename("DeleteFile", "APIDeleteFile") \
       rename("CopyFile", "APICopyFile") \
       rename("MoveFile", "APIMoveFile")  \
       rename("SetEnvironmentVariable", "APISetEnvironmentVariable") \
       rename("GetObject", "APIGetObject") \
       exclude("_FILETIME", "IStream","ISequentialStream", \
            "_LARGE_INTEGER", "_ULARGE_INTEGER", "tagSTATSTG", \
             "IEnumUnknown","IPersistFile", "IPersist", "IPictureDisp")
```

Но такой подход не рекомендуется потому, что в случае переноса проекта на другой компьютер, путь к файлу ***RxInventor.tlb*** придется корректировать. Поэтому лучше добавить еще один дополнительный путь для поиска файлов.

Переходим к реализации объявленных прототипов. Для этого откроем файл ***NativeCode.cpp***. Напомню, что горячая комбинация клавиш между заголовочным файлом и файлом исходного кода ***Ctrl +K+O***. Оформляем следующую реализацию программного кода:

```
#include "stdafx.h"
#include "NativeCode.h"
#include <WTypes.h>
#include <string>
using namespace InventorNative;

CNativeCode::CNativeCode()//Конструктор
{
	HRESULT Result = NOERROR;
	//Получение идентификатора Inventor
	CLSID InvAppClsid;
	Result = CLSIDFromProgID(L"Inventor.Application", &InvAppClsid);
	//Получение смарт-указателя на IUnknown
	CComPtr<IUnknown> pInvAppUnk;
	Result = ::GetActiveObject(InvAppClsid, __nullptr, &pInvAppUnk);
	//Запрос интерфейса Inventor
	Result = pInvAppUnk->QueryInterface(__uuidof(Application), (void **)&pInvApp);
}
CNativeCode::~CNativeCode()//Деструктор
{
}
void CNativeCode::ShowMassage()//Функция вызова окна
{
	HRESULT Result = NOERROR;
	//Получения названия
	CComBSTR bstrCaption;
	Result = pInvApp->get_Caption(&bstrCaption);
	//Создание текста сообщения
	CComBSTR bstrMsgText(L"Native C++: Здравствуй ");
	bstrMsgText.AppendBSTR(bstrCaption);
	//Вывод окна
	int msgboxID = MessageBoxW(__nullptr, bstrMsgText, L"Приветствие", MB_OK);
	//Изменим значение свойства Inventor.Caption
	CComBSTR bstrNewCaption(L"Моя модификация Inventor");
	Result = pInvApp->put_Caption(bstrNewCaption);
}
```

По сравнению с C#, написанный код более «объемный», для примера в C# программный эквивалент, написанный в конструкторе класса, занял бы всего одну строчку:

```
	InvApp = System.Runtime.InteropServices.Marshal.GetActiveObject("Inventor.Application")
as Inventor.Application;
```

и тело процедуры `CNativeCode::ShowMassage()` выглядело бы тоже одной строчкой:

```
	System.Windows.Forms.MessageBox.Show("Native C++: Здравствуй " + InvApp.Caption);
```

Но таковы реалии «нативного» C++ и технологии COM.

Кстати, для получения структуры ***CLSID*** не обязательно заставлять Windows перебирать базу данных ключей ***CLSID*** в реестре. Перебор ключей ***CLSID*** может негативно сказаться на производительности, если такой код попадет в цикл. Поэтому можно посмотреть значение ***CLSID*** в заголовочных файлах SDK в файле ***ServerCLSIDs.h*** и сразу инициализировать структуру ***CLSID***. В файле ***ServerCLSIDs.h*** значения ***CLSID*** определены в качестве подстановочных макросов.

Можно заменить часть кода в конструкторе класса на следующий код:

```
	//Получение идентификатора Inventor
	//{B6B5DC40-96E3-11d2-B774-0060B0F159EF}
	const CLSID InvAppClsid = { 0xB6B5DC40, 0x96E3, 0x11d2,
	{ 0xB7, 0x74, 0x00, 0x60, 0xB0, 0xF1, 0x59, 0xEF } };
	//Result = CLSIDFromProgID(L"Inventor.Application", &InvAppClsid);
```

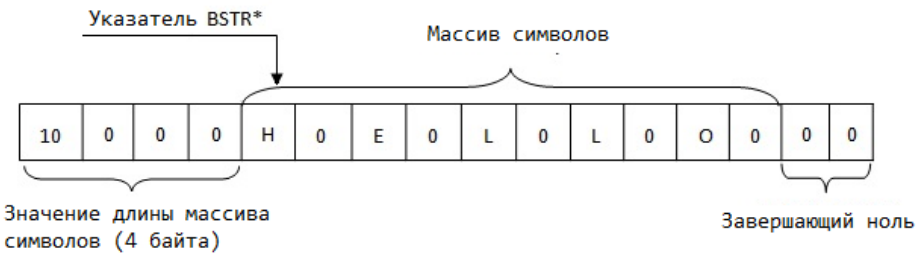
Хотя использование подстановочных макросов несколько надежнее, т.к. разработчики API Inventor теоретически могут изменить данные ***CLSID***, и придется корректировать программу.

Строковый типа *BSTR* в «нативном» C++

Интересной особенностью работы COM, применимо для нашего примера, является работа со строковыми переменными. В COM для этих целей имеется объект ***BSTR*** (строка в формате Unicode), в нашем случае использован класс-обертка ***CComBSTR*** для того, что бы не заботится об очитки памяти от в ручную. Хотя ***BSTR*** по факту это такой же обычный указатель как указатель на строку в стиле Си, но инициализировать его в стиле Си нельзя:

```
BSTR str = L"Test";//Неправильно
```

Это связано с тем, что перед указателем на массив текстовых символов, в ***BSTR*** должно быть выделено 4 байта памяти, в которых записано количество символов выделенных под строку ***BSTR***. Графически это можно показать так:



Поэтому для создания ***BSTR*** существует специальная функция:

```
BSTR str =SysAllocString( L"Test");//Правильно
```

После, того как надобность в данной строковой переменной отпадет, её так же необходимо удалить специальной функцией:

```
SysFreeString(str);
```

Особенность работой со строковыми переменными в «нативном» C++ является, то, что уже созданную строковую переменную нельзя модифицировать, т.е. нельзя ни добавлять, ни удалять символы. Привычный на C# стиль синтаксиса выдаст ошибку:

```
wchar_t* str1 = L"Привет ";
wchar_t* str2 = L"Inventor";
wchar_t* str3 = str1 + str2;//выдаст ошибку
```

Чтобы изменить строку её всегда нужно создавать новую. Это связано с тем, что строковая переменная в «нативном» C++ это массив, который расположен в адресном пространстве, которая называется «куча» и количество выделенной компилятором памяти зависит от длины конкретной строки. Поэтому ей поведение отличается от переменой типа ***int*** под которую всегда выделено одно и тоже строго определенное количество памяти, поэтому переменную типа ***int*** мы всегда можем легко модифицировать обычным присвоением другого числа, а уже созданную строковую переменную модифицировать нельзя. Конечно же, в «нативном» C++ уже предусмотрены дополнительные стандартные библиотеки для облегчения работы со строковыми переменными, например, ***<string>.***

Это выглядит необычно после работы на VB.NET и C#, но это так. На самом деле и VB.NET и C#, за «кулисами» NET имеют в точности такуюже технологию работы со строковыми

переменными, как и в «нативном» C++, но реализация этого умело, замаскирована высокоуровневыми языками программирования.

Альтернативой ***BSTR*** является использование класса-обертки ***CComBSTR*** ,который мы как раз использовали в нашем примере. Класс-обертка ***CComBSTR*** сам отслеживает, когда необходимо очищать память. В добавок, этот класс-обертка содержит методы, которые позволяю модифицировать строки типа ***BSTR***.

Считывание и присвоение свойств для Inventor здесь так же отличается от привычного подхода на VB.NET и C#. На VB.NET и C# с любым свойством мы работаем как с обычным членом класса. Здесь же в «нативном» C++ работа со свойствами происходит через функции, имеющие определенный зарезервированный префикс: ***get_*** и ***put_***. В нашем примере получить значение свойства:

```
Result = pInvApp->get_Caption(&bstrCaption);
```

, а присвоить значение:

```
Result = pInvApp->put_Caption(bstrNewCaption);
```

Этот подход происходит из особенностей устройства COM-технологии (работа происходит только через функции). Но на VB.NET и C# этот процесс скрыт за «кулисами» NET.

Я специально подробно остановился на разъяснении строковых переменных, в том числе и ***BSTR***. Потому как, у начинающего изучать работу Inventor на «нативном» C++, с небольшой базой знаний по самому «нативному» C++, это может вызвать значительное неприятие работы связки Inventor + «нативный» C++. Конечно же, я не собираюсь рассказывать здесь про все особенности COM и Dispatch - технологий. Я только помогаю сделать небольшой шаг в этом направлении.

«Нативный» класс ***CNativeCode*** готов к работе.

Класс-обертка между «нативным» и «управляемым» кодом

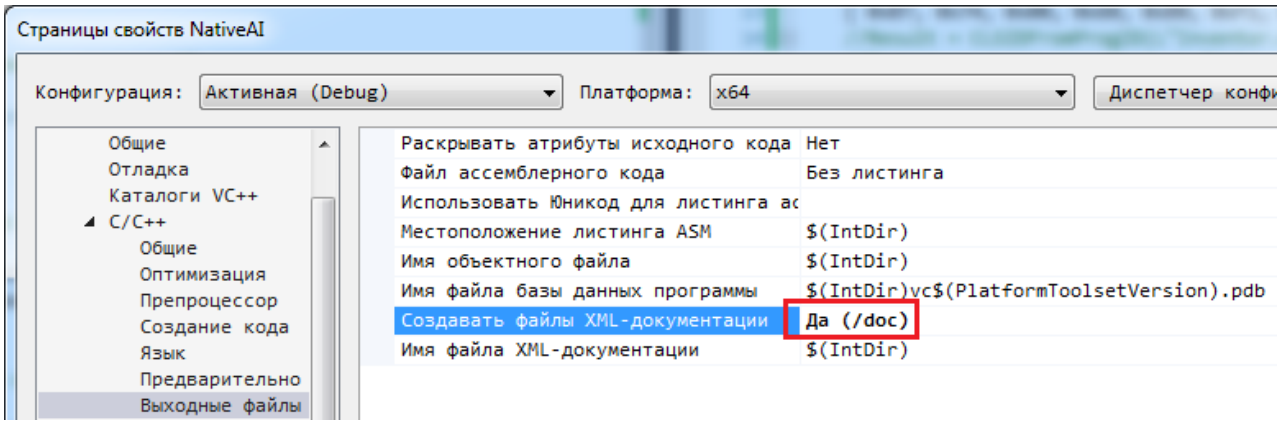
Перейдем к созданию «управляемого» класса-обертки. Откроем на редактирование заголовочный файл ***NativeAI.h*** (т.к. мы брали шаблон для DLL на C++/CLI , то этот класс должен быть автоматически сгенерирован в проекте ***NativeAI***). В этом файле модифицируем программный код, что бы он выглядел так:

```
// NativeAI.h
#pragma once
#include "NativeCode.h"
namespace NativeAI {
	///<summary>
	///Обертка для связи с нативным AI
	///</summary>
	public ref class WrapperNativeAI{
	public:
		///<summary>
		///Конструктор
		///</summary>
		WrapperNativeAI();
		~WrapperNativeAI();//Деструктор
		///<summary>
		///Вызов окна с сообщением и модифицирование заголовка Inventor
		///</summary>
		void ShowMessage();
	private:
		System::IntPtr mpNativeCode;
		!WrapperNativeAI();//Финализатор (вызывается автоматически)
	};
}
```

Опишем существующие «тонкости» в этой части программного кода.

Т.к. класс ***WrapperNativeAI*** «управляемый», то его функции с модификатором ***public*** будет видеть любой другой NET-проект.

Т.к. работать с данными объектами будем из другого проекта, то желательно добавить xml-комментарии и включить создание xml-документации. Xml-комментарии могут быть весьма разнообразны, но для простоты ограничился только тегом ***<summary>.*** Это все необходимо для того что бы иметь всплывающие подсказки от IntelliSense, когда мы будем вызывать функции из проекта C#. Для этого необходимо включить создание xml-документации на странице свойств проекта C++ ***NativeAI***:



Забегу немного вперед и скажу, что после компиляции проекта в выходном каталоге должен появится xml-файл с таким же именем, как и у выходного DLL-файла:

Благодаря xml-документации в проекте C# ***CSharpWithNativeCpp*** мы будем видеть следующее, т.е. как раз то что мы написали в теге ***<summary>*** в соседнем проекте ***NativeAI***:

```
public void Activate(Inventor.ApplicationAddInSite AddInSiteObject,
{
    InvApp = AddInSiteObject.Application;
    //Вызов неуправляемого AI
    NativeAI.WrapperNativeAI NativeAI = new NativeAI.WrapperNativeAI
    NativeAI.ShowMessage();
}
ссылок 0
public dynami
{ get { return null; } }
```

```
void WrapperNativeAI.ShowMessage()
Вызов окна с сообщением и модифцирвоание заголовка
```

Но все это будет потом после компиляции и подключения проекта ***NativeAI*** к проекту ***CSharpWithNativeCpp.***

Вернемся к заголовочному файлу ***NativeAI.h***.

Т.к. класс объявлен как «управляемый», то C++/CLI не позволяет создать «нативный» член класса, поэтому создана «управляемую» обертку указателя для хранения «нативного» объекта.

```
System::IntPtr mpNativeCode;
```

Т.к. мы будем использовать «нативный» экземпляр класса ***CNativeCode***, то необходимо объявить финализатор для того что бы потом сборщик «мусора» из Framework , мог вызвать деструктор класса и освободить память из «неуправляемой кучи». Если память в «неуправляемой кучи» не освобождать, то может возникнуть явление называемое «утечкой памяти». Память может освобождаться автоматически при использовании смарт-указателей. Но в данном примере я использую обычный указатель, поэтому память будет высвобождаться в финализаторе:

```
// Главный DLL-файл.

#include "stdafx.h"
#include "NativeAI.h"

namespace NativeAI{
	WrapperNativeAI::WrapperNativeAI()//Конструктор
	{
		//Создаем экземпляр неуправляемого класса в куче
		CNativeCode* pNative = new CNativeCode;
		System::IntPtr mpNative(pNative);
		//Присваиваем управляемому указателю указатель класса
		this->mpNativeCode = mpNative;

	}
	WrapperNativeAI::~WrapperNativeAI()//Деструктор
	{
		//Приведение указателя к нужному типу
		CNativeCode* pNative = static_cast<CNativeCode*> (this->mpNativeCode.ToPointer());
		delete pNative; pNative = __nullptr;//удаляем экземпляр класса из кучи
	}

	WrapperNativeAI::!WrapperNativeAI()//Финализатор
	{
		WrapperNativeAI::~WrapperNativeAI();//вызов деструктора
	}

	void WrapperNativeAI::ShowMessage()//Функции с окном сообщения
	{
		//Приведение указателя к нужному типу
		CNativeCode* pNative = static_cast<CNativeCode*> (this->mpNativeCode.ToPointer());
		pNative->ShowMassage();//Вызов сообщения
	}

	}
```

Кратко опишем работу программного кода.

В конструкторе производится создание экземпляра «нативного» класса в `CNativeCode` «неуправляемой куче». Далее идет передача значения указателя «нативного» класса в «управляемый» указатель `mpNativeCode`.

При вызове функции:

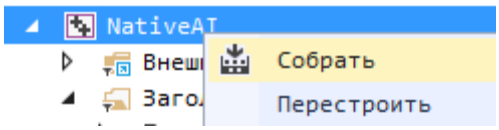
```
void WrapperNativeAI::ShowMessage();
```

«управляемый» указатель будет преобразован в указатель «нативного» класса ***CNativeCode**** и будет вызвана функция

```
pNative->ShowMassage();
```

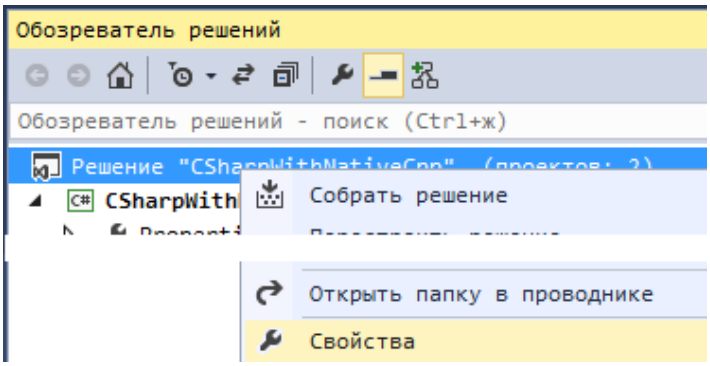
которая покажет сообщение и переименует заголовок Inventor.

Все проект с «нативным» вызовом функция для Inventor готов. Компилируем его:

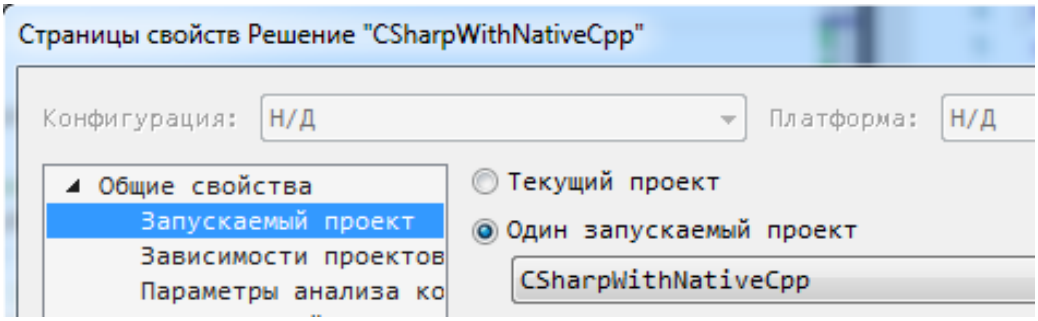


Окончательная настройка основного проекта на C#

Теперь вернемся к проекту C# ***CSharpWithNativeCpp***. Проверим назначен ли данный проект запускаемым, для этого на заголовке решения вызовем контекстное меню и вызовем команду свойства:

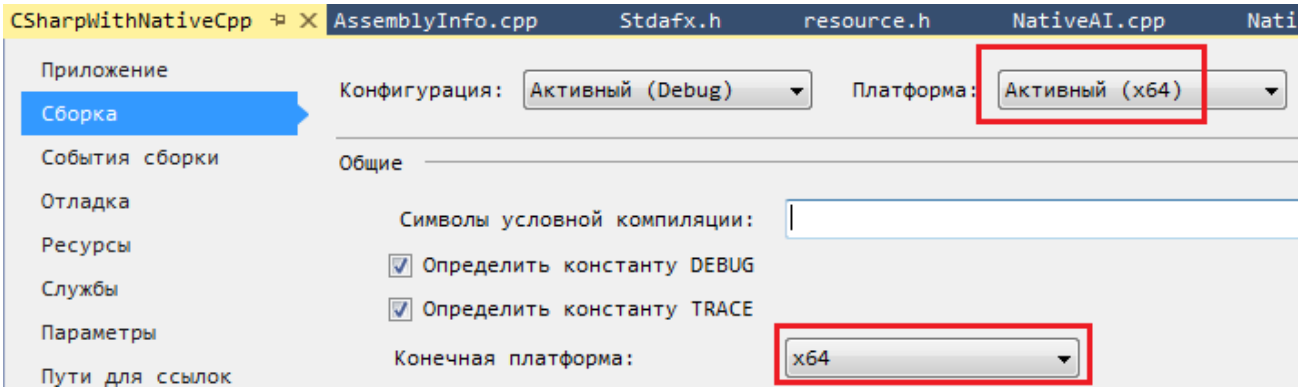


Убедимся, что проект ***CSharpWithNativeCpp*** является запускаемым.

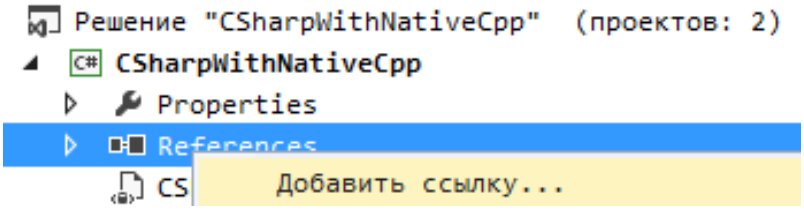


Как я писал в самом начале этой главы, что проект ***CSharpWithNativeCpp*** должен быть уже настроен в режиме ***Debug*** и готов к запуску для отладки, с учетом всего сказанного в данной статье.

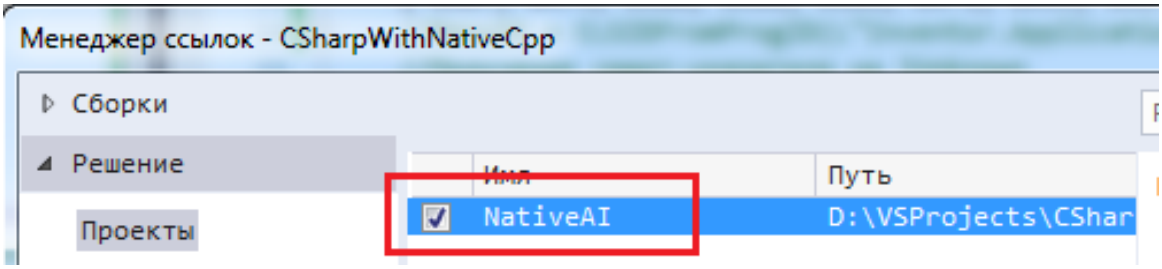
Переключим конечную платформу проекта ***CSharpWithNativeCpp*** на 64 бита:



Открываем менеджер ссылок проекта ***CSharpWithNativeCpp***:



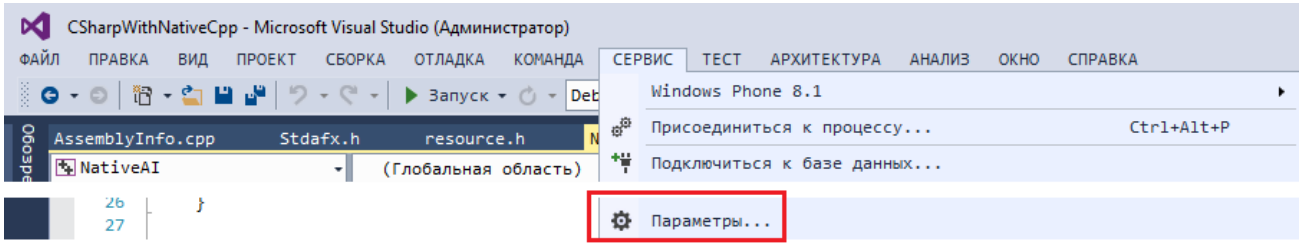
Добавляем ссылку на проект ***NativeAI***:



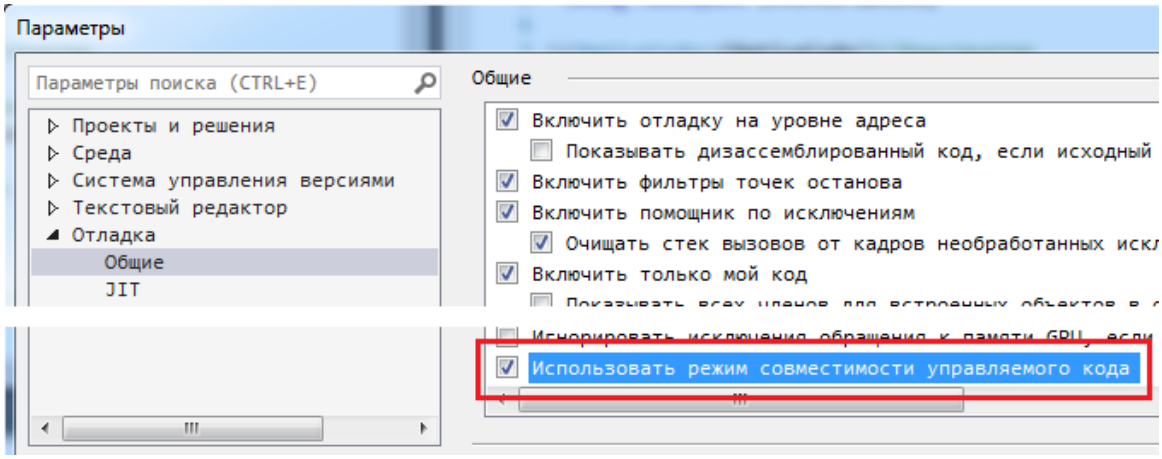
Открываем файл ***CSharpWithNativeCpp.cs*** на редактирование и добавляем вызов «нативных» функций:

```
using System;
using System.Runtime.InteropServices;

namespace CSharpWithNativeCpp
{
    [GuidAttribute("1CC3802A-18B2-4CD4-A5F5-C4223F53C0E3")]
    public class AddInServer : Inventor.ApplicationAddInServer
    {
        private Inventor.Application InvApp;
        public void Activate(Inventor.ApplicationAddInSite AddInSiteObject, bool FirstTime)
        {
            InvApp = AddInSiteObject.Application;
            //Вызов неуправляемого AI
            NativeAI.WrapperNativeAI NativeAI = new NativeAI.WrapperNativeAI();
            NativeAI.ShowMessage();
        }
        public dynamic Automation
        { get { return null; } }
        public void Deactivate()
        {
        }
        public void ExecuteCommand(int CommandID)
        {
        }
    }
}
```

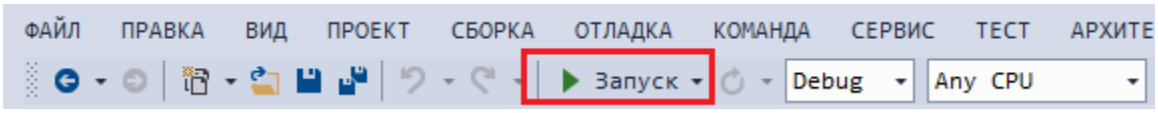
Перед запуском включим опцию «режима совместимости управляемого кода» в настройках Visual Studio. Для этого зайдем в меню «Параметры»:



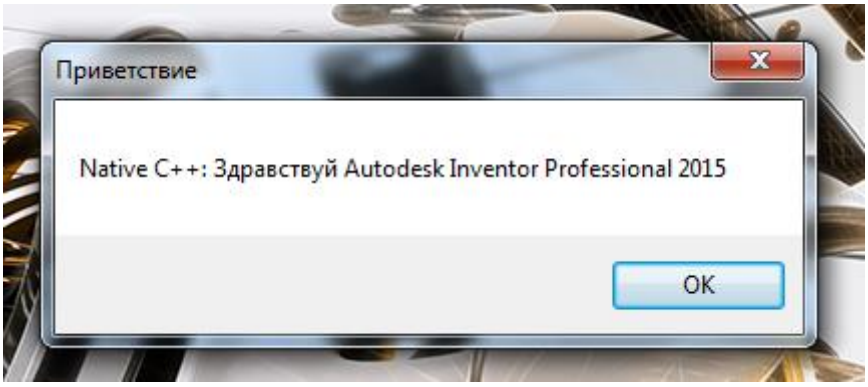
Найдем узел ***«Отладка->Общие»,*** в списке опций находим нужную нам опцию:



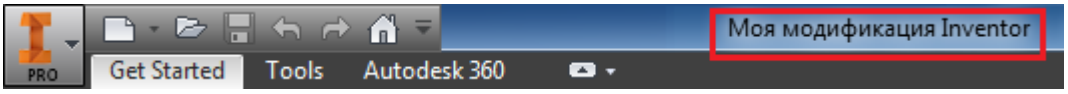
Теперь компилируем проект и запускаем его в режиме Debug:



В результате чего должно появится окно с сообщением:



После полной загрузки Inventor, можно будет увидеть изменившийся заголовок программы:



Готовый пример можно скачать здесь.

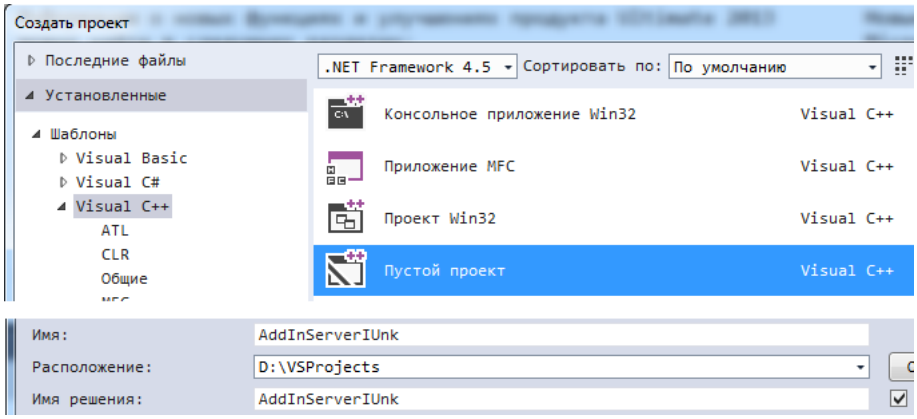
Чистый «нативный» C++: AddIn на базе *IUnknown*

Теперь перейдем к написанию AddIn для Inventor используя только чистый «нативный» C++ (без элементов Framework). Эти последние разделы рассчитаны на особо любознательных программистов-прикладников, которым рамки Framework кажутся тесными. К сожалению MSDN достаточно скупой источник информации по поводу COM-технологии, самая лучшая книга, которая мне попалась на просторах интернета, это «Основы COM» автор Дейл Роджерсон.

Подключение к Inventor может производиться через любой из двух COM-интерфейсов: ***IUnknown*** или ***IDispatch***. Интерфейс ***IUnknown*** является базовым для любого COM-компонента, в том числе и для компонентов на ***IDispatch***. Реализовать ***IUnknown*** гораздо проще, т.к. в нем меньше количество чистых виртуальных функций, чем в ***IDispatch***. В COM-технологии приняты термины ***клиент*** (тот, кто управляет) и ***сервер*** (тот, кем правляют). При подключении AddIn, клиентом у нас будет выступать Inventor, а сервером будет выступать AddIn. Но после подключения AddIn ситуация изменится на противоположную.

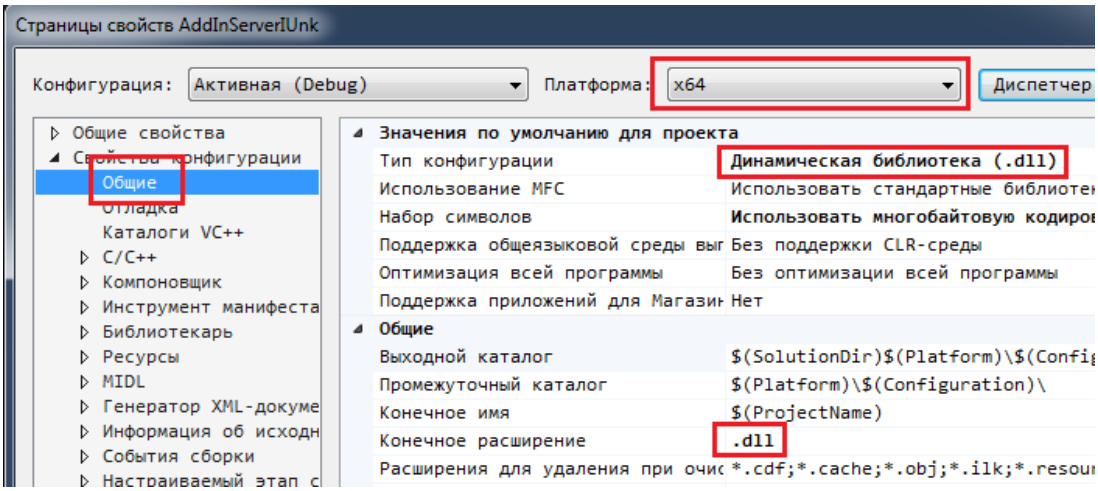
Хотя Autodesk предоставляет шаблоны для генерации AddIn на базе обоих интерфейсов ***IUnknown*** и ***IDispatch***, разобраться в самом принципе, как это все работает очень трудно. Проблемы в понимании вызывают многократные переопределения подстановочных макросов, большая часть из них унаследована от заголовочных файлов Windows. А так же генерируемый набор файлов далек от строгой структурированности. Поэтому мы этими шаблонами пользоваться не будем, а будем разбирать все по полочкам.

И так создаем новый пустой проект с названием ***AddInServerIUnk*** (не спутайте с «управляемым» пустым проектом ***CLR***):

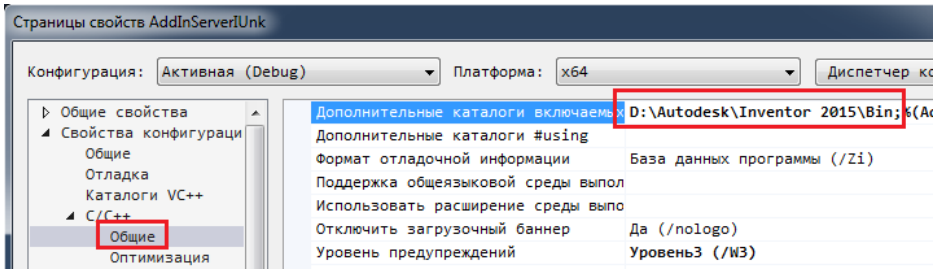


Настройки проекта

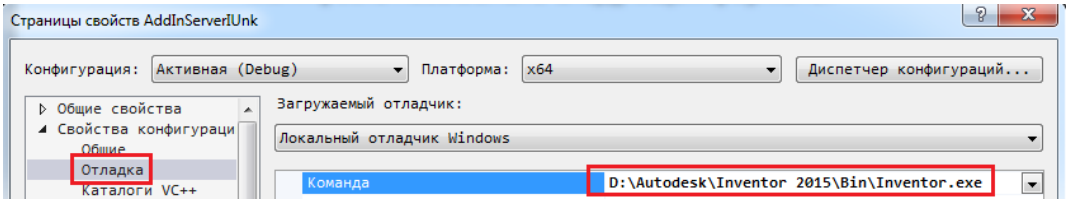
Перед созданием кода настроем проект. На странице свойств сначала переключим проект на платформу ***x64***, а затем тип конфигурацию переключим на ***DLL***:



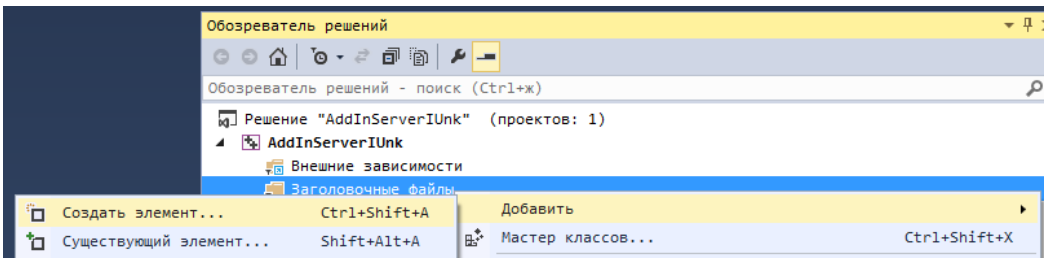
Пропишем дополнительные пути поиска для нахождения файла библиотеки типов ***RxInventor.tlb***:



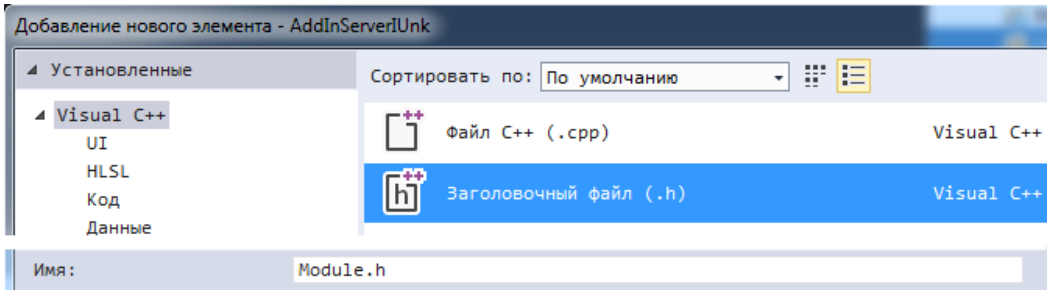
Пропишем путь к запускаемому файлу Inventor для будущей отладки:



Сохраняем и закрываем окно свойств. В «Обозревателе решений» добавляем новый заголовочный файл:



Исторически так сложилось, что синонимом DLL-файлу является слово ***Module***. Поэтому новый файл называем ***Module.h***:



Описание экспортируемых функций

Немного обратимся к теории. Изначально для подключения COM-компонента необходимо было реализовать 5 экспортируемых функций:

- `BOOL __stdcall DllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved);`
  Это точка входа для любой DLL. Эта функция вызывается самая первая при компоновке DLL в Inventor.
  Через ***DllMain*** передается описатель `hModule` для загруженного DLL-файла. `hModule` служит для получения информации о загруженном DLL-файле, например, можно узнать путь и имя загруженного DLL-файла через функцию WinAPI ***GetModuleFileName***, или через функцию ***GetModuleInformation*** узнать по какому адресу в процессе Inventor расположен загруженный DLL-файл.
  Параметр (флаг) `dwReason` передает причину вызова ***DllMain***, причины описаны через макросы подстановки (названия говорят сами за себя): ***DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH, DLL_PROCESS_DETACH*** и ***DLL_THREAD_DETACH***.
  `lpReserved -зарезервированный параметр.`
- `HRESULT __stdcall DllGetClassObject(const CLSID& clsid, const IID& iid, void** ppv);`
  Функция для создания фабрики классов. Принимает ***clsid*** от Inventor, значение должно соответствовать ***clsid*** AddIn, который прописывается в манифесте и ***.addin***-файле. AddIn обязан проверить принятый ***clsid*** и если он соответствует ***clsid*** самого AddIn, то необходимо будет возвратить реализованный интерфейс ***IClassFactory***, через указатель на указатель *****ppv***. Реализация ***IClassFactory*** это обязательная функциональность для COM-интерфейсов. Фабрика классов позволяет управлять созданием экземпляра класса самого компонента по запрашиваемому интерфейсу через параметр ***iid***. Это связано с тем, что в одном DLL-файле может находиться несколько компонентов. Для малоопытного человека это все звучит запутанно, но позже, что бы лучше понять, я приведу схему вызовов.
- `HRESULT __stdcall DllCanUnloadNow();`
  Функция для проверки возможности выгрузки DLL. Позволяет разрешить или запретить выгрузку DLL из памяти.
- `HRESULT __stdcall DllRegisterServer();`
  Функция для регистрации компонента в реестре Windows.
- `HRESULT __stdcall DllUnregisterServer();`
  Функция для отмены регистрации компонента в реестре Windows.

В случае использования технологии «***Registry-free***» необходимость реализации двух последних функций ***DllRegisterServer*** и ***DllRegisterServer*** отпадает, т.к. работать с реестром мы не будем. Поэтому необходимо реализовать только три функции ***DllMain***, ***DllGetClassObject*** и ***DllCanUnloadNow***.

Глобальные объекты: Module

Приведу текст заголовочного файла ***Module.h***:

```
//Модуль для подключения COM-компонента
#include <windows.h>
#import <RxInventor.tlb> \
	rename_namespace("InventorNative") \
	named_guids raw_dispinterfaces \
	high_method_prefix("Method") \
	rename("DeleteFile", "APIDeleteFile") \
	rename("CopyFile", "APICopyFile") \
	rename("MoveFile", "APIMoveFile")  \
	rename("SetEnvironmentVariable", "APISetEnvironmentVariable") \
	rename("GetObject", "APIGetObject") \
	exclude("_FILETIME", "IStream","ISequentialStream", \
		"_LARGE_INTEGER", "_ULARGE_INTEGER", "tagSTATSTG", \
		"IEnumUnknown","IPersistFile", "IPersist", "IPictureDisp")

extern HMODULE g_hModule;//Описатель загруженной DDL
extern long g_cComponents; // Количество работающих компонентов
extern const IID CLSID_component;//Идентификатор AddIn

extern "C"
{
	//Точка входа при загрузке DLL в память
	BOOL __stdcall DllMain(HMODULE hModule, DWORD  dwReason, LPVOID  lpReserved);
	//Точка входа для получения фабрики класса
	HRESULT __stdcall DllGetClassObject(const CLSID& clsid, const IID& iid, void** ppv);
	//Точка входа для проверки возможности выгрузки DDL
	HRESULT __stdcall DllCanUnloadNow();
}
//Вместо создания DEF-файла
#pragma comment(linker, "/EXPORT:DllMain=DllMain,PRIVATE")
#pragma comment(linker, "/EXPORT:DllCanUnloadNow=DllCanUnloadNow,PRIVATE")
#pragma comment(linker, "/EXPORT:DllGetClassObject=DllGetClassObject,PRIVATE")
```

Как я писал выше, эта директива ***#import <RxInventor.tlb>*** необходима для использования библиотеки типов Inventor. Файл ***RxInventor.tlb*** находится в папке, где и запускной файл ***Inventor.exe***. Но эта директива будет показывать ошибку из-за отсутствия файла ***rxinventor.tlh*** в проекте, но беспокоиться не нужно, этот файл появится после первой компиляции проекта. Бывает, что ***IntelliSense*** продолжает не воспринимать появившийся файл ***rxinventor.tlh***, в таком случае помогает перезагрузка Visual Studio.

Далее идут три глобальных переменных (префикс g_ от слова global):

- `extern HMODULE g_hModule;`
  Будет хранить описатель загруженного DLL в процесс Inventor.
- `extern long g_cComponents;`
  Счетчик работающих компонентов, в нашем случае, будет только один работающий компонент, теоретически AddIn параллельно может использовать еще какая-нибудь третья программа. И будет очень не хорошо, если DLL будет выгружена из памяти раньше времени.
- `extern const IID g_CLSID_component;`
  Уникальный 128-ми битный идентификатор AddIn, который мы будем прописывать в манифесте и ***.addin***-файле.

Далее идут объявления трёх экспортируемых функций:

```
BOOL __stdcall DllMain(HMODULE hModule, DWORD  dwReason, LPVOID  lpReserved);
HRESULT __stdcall DllGetClassObject(const CLSID& clsid, const IID& iid, void** ppv);
HRESULT __stdcall DllCanUnloadNow();
```

Здесь может возникнуть вопрос почему эти три функции перед объявлением не имею не имеют ключевого слова ***__declspec(dllexport)***? Дело в том, что у нас подключен заголовочный файл ***#include <windows.h>***, и в нем прототипы функции ***DllGetClassObject*** и ***DllCanUnloadNow*** уже определены как (макросы я развернул):

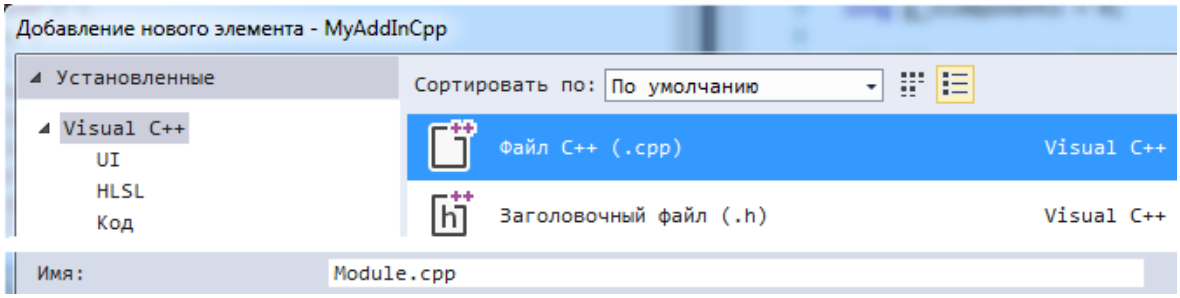
```
extern "C" HRESULT __stdcall DllCanUnloadNow(void);
```

ключевое слово ***__declspec(dllexport)*** вызовет ошибки при компиляции. Но функции экспортировать все таки необходимо. Для этих целей можно создать .DEF-файл. В .DEF-файле будут прописаны имена экспортируемых функций с точными их названиями, без декорирования имени (декорирование – это изменение имени функции компилятором в конечном DLL-файле).

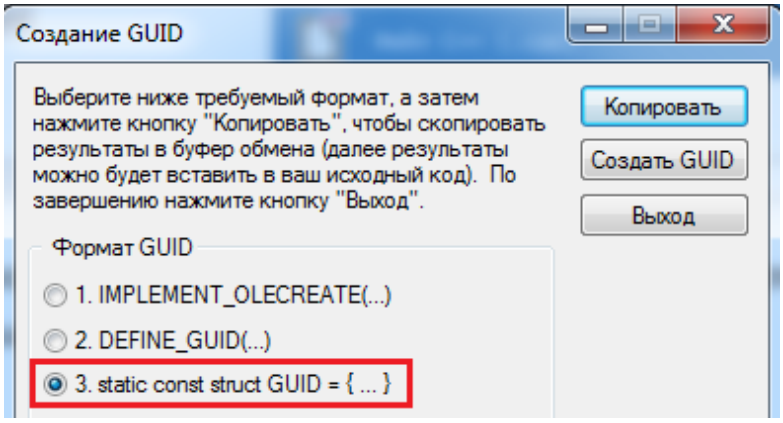
Но Visual C++ имеет альтернативу .DEF-файлу. Это директива ***#pragma comment***. Поэтому в самом конце находятся три директивы для экспортируемых функций. Эти директивы заставляю компилятор экспортировать функции и не применять декорирование имен к этим функциям в откомпилированном DLL-файле.

```
#pragma comment(linker, "/EXPORT:DllMain=DllMain,PRIVATE")
#pragma comment(linker, "/EXPORT:DllCanUnloadNow=DllCanUnloadNow,PRIVATE")
#pragma comment(linker, "/EXPORT:DllGetClassObject=DllGetClassObject,PRIVATE")
```

Добавим в проект файл исходного кода с названием Module.cpp:



Т.к. в данном коде будем определять ***clsid*** для AddIn, то воспользуемся утилитой для генерации GUID из Visual Studio:



Воспользуемся форматом номер три, т.к. нам будет нужно представление и в виде строки и в виде структуры.

Добавляем следующий текст программы для реализации заголовочного файла:

```
#include "Module.h"
#include "ClassFactory.h"
//Инициализацяи глобальных переменных
HMODULE g_hModule = __nullptr;
long g_cComponents = 0;
//CLSID компонента {3AD1A692-1E0F-4A12-9AD6-20B0512CE743}
const IID CLSID_component = { 0x3ad1a692, 0x1e0f, 0x4a12,
{ 0x9a, 0xd6, 0x20, 0xb0, 0x51, 0x2c, 0xe7, 0x43 } };
BOOL __stdcall  DllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved)
{
	if (dwReason == DLL_PROCESS_ATTACH)
	{
		g_hModule = hModule;
		g_cComponents++;
	}
	else if (dwReason == DLL_PROCESS_DETACH)
	{
		g_cComponents--;
	}
	return TRUE;
}
HRESULT __stdcall DllGetClassObject(const CLSID& clsid, const IID& iid, void** ppv)
{
	// Проверка на возможность создания компонта
	if (clsid != CLSID_component)
	{ return CLASS_E_CLASSNOTAVAILABLE; }

	// Создать фабрику класса
	CClassFactory* pFactory = new CClassFactory;

	// Получить требуемый интерфейс
	HRESULT hr = pFactory->QueryInterface(iid, ppv);
	pFactory->Release();
	return hr;
}
HRESULT __stdcall DllCanUnloadNow()
{
	if (g_cComponents == 0)
	{
		return S_OK;
	}
	else
	{
		return S_FALSE;
	}
}
```

В данном программном коде видно, что когда DLL присоединяется к процессу Inventor, мы в функции ***DllMain*** присваиваем:

```
g_cComponents = 1
```

Это не даст Inventor выгрузить DLL из памяти без разрешения AddIn.

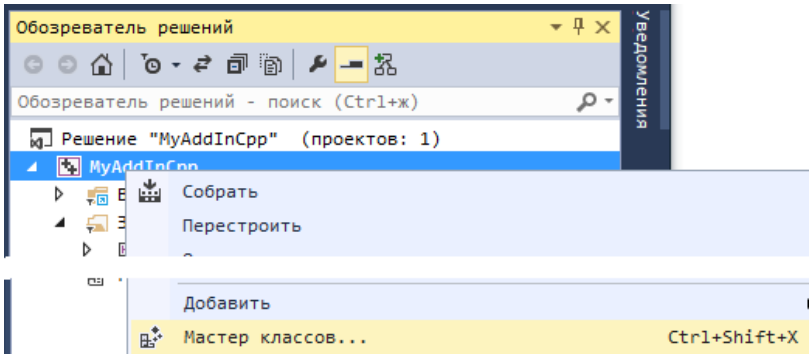
Функция ***DllGetClassObject*** проверяет запрашиваемый ***clsid*** на соответствие данному AddIn:

```
if (clsid != CLSID_component)
{ return CLASS_E_CLASSNOTAVAILABLE; }
```

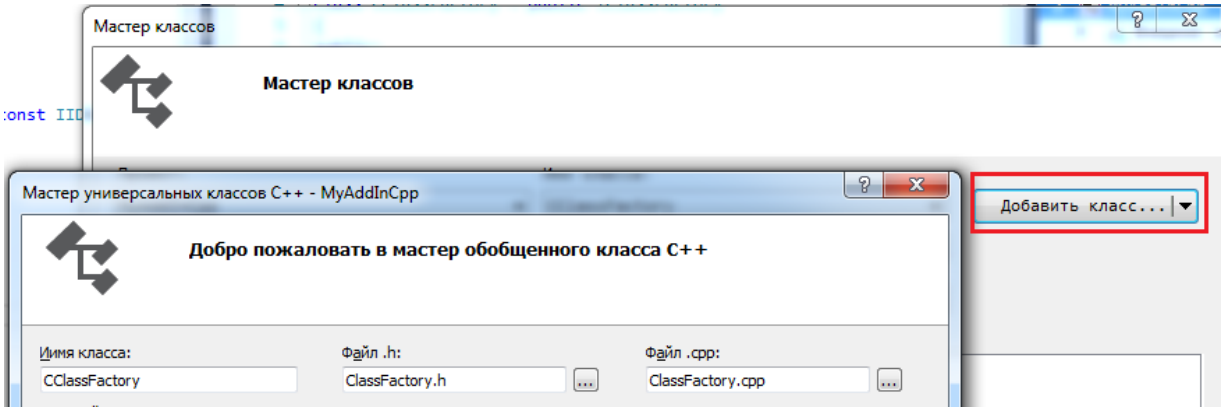
Объявление фабрики классов

Если все нормально, то создается экземпляр фабрики классов. И запрашивается и возвращается в Inventor указатель на интерфейс фабрики классов ***IClassFactory*** или базовый интерфейс ***IUnknown***.

Перейдем к реализации фабрики класса при помощи «Мастера классов»:



Дадим название новому классу ***CClassFactory***:



Откроем заголовочный файл ***ClassFactory.h*** и впишем туда следующий код:

```
#pragma once
#include "Module.h"

class CClassFactory: public IClassFactory
{
public:
	virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv);
	virtual ULONG __stdcall AddRef();
	virtual ULONG __stdcall Release();
	virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,//__nullptr
                                                 const IID& iid,//[in]
                                                 void** ppv);//[out]
	virtual HRESULT __stdcall LockServer(BOOL bLock);

	CClassFactory();
private:
	long m_cRef;
};
```

Описание интерфейса *IUnknown*

Небольшая экскурсия в теорию. ***IClassFactory*** наследуется от ***IUnknown***, как все в COM-технологии. Классическое определение ***IUnknown*** выглядит так:

```
struct IUnknown
{
virtual HRESULT __stdcall QueryInterface(const IID& iid, void** ppv)=0;
virtual ULONG __stdcall AddRef()=0;
virtual ULONG __stdcall Release()=0;
}
```

Функция ***QueryInterface*** принимает уникальный 128-ми битный идентификатор и возвращает указатель на интерфейс компонента. Функции ***AddRef*** и ***Release*** занимаются отслеживанием жизни компонента в памяти. ***AddRef*** добавляет 1 к счетчику ссылок на компонент, а ***Release*** уменьшает счетчик на 1. И когда счетчик ссылок становится равным нулю, функция ***Release*** уничтожает компонент, в памяти вызывая: ***delete this***. Функция ***AddRef*** обычно вызывается автоматически при запросе интерфейса через ***QueryInterface***, а вот вызов функции ***Release*** приходится делать программисту самому.

Однако существуют «умные» указатели в виде классов-оберток типа ***CComPtr***, которые сами отслеживают необходимость. Единственное условие автоматической работы такого указателя, это размещение его в стеке, т.е. нельзя создавать его в «куче» при помощи оператора ***new***. Принцип работы таких «умных» указателей прост, во время вызова «умного» конструктора происходит вызов ***AddRef***, а когда стек, по окончанию работы функции, очищается, то автоматически вызывается «умный» деструктор класса, откуда вызывается ***Release***.

Но Microsoft считает, что указатель на виртуальную таблицу функций и сама виртуальная таблица функций, для интерфейсных классов не нужны, а также можно сразу задать GUID для ***IUnknown***:

```
struct
__declspec(uuid("00000000-0000-0000-C000-000000000046"))
__declspec(novtable)
	IUnknown
{
	virtual HRESULT __stdcall QueryInterface(const IID& iid,//[in]
						void** ppv) = 0;//[out]
	virtual ULONG __stdcall AddRef() = 0;
	virtual ULONG __stdcall Release() = 0;
};
```

Определение абстрактного базового класса, в стиле Microsoft, для ***IClassFactory*** будет выглядеть так:

```
struct
__declspec(uuid("00000001-0000-0000-C000-000000000046"))
__declspec(novtable)
IClassFactory : public IUnknown
{
virtual HRESULT __stdcall CreateInstance(IUnknown* pUnknownOuter,//__nullptr
					const IID& iid,//[in]
					void** ppv)=0;//[out]
virtual HRESULT __stdcall LockServer(BOOL bLock)=0;
};
```

Интерфейс ***IClassFactory***, так же как и интерфейс ***IUnknown***, имеет описание в заголовочных файлах Windows. Поэтому мы воспользовались наследованием готового абстрактного класса. При сильном желании абстрактные классы COM-интерфейсов можно описать самому и они будут так же успешно работать, главное при этом не перепутать порядок следования чистых виртуальных функций.

Реализация фабрики классов

Вернемся к нашему коду. Пришло время реализовать класс ***CClassFactory*** в файле ***ClassFactory.cpp***. Откроем этот файл и добавим в него реализацию класса:

```
#include "ClassFactory.h"
#include "AddInServerIUnk.h"
HRESULT __stdcall    CClassFactory::QueryInterface(const IID& iid, void** ppv)
{
	if (iid == IID_IUnknown)
	{ *ppv = static_cast<IUnknown*>(this);}

	else if (iid == IID_IClassFactory)
	{ *ppv = static_cast<IClassFactory*>(this);}

	else
	{ *ppv = __nullptr; return E_NOINTERFACE;}
	reinterpret_cast<IUnknown*>(*ppv)->AddRef();
	return S_OK;
}
ULONG __stdcall CClassFactory::AddRef()
{ return InterlockedIncrement(&m_cRef); }

ULONG __stdcall CClassFactory::Release()
{
	if (InterlockedDecrement(&m_cRef) == 0)
	{ delete this; return 0;}
	return m_cRef;
}
HRESULT __stdcall CClassFactory::CreateInstance(IUnknown* pUnknownOuter,
                                                const IID& iid,
                                                void** ppv)
{
	// Агрегирование не поддерживается
	if (pUnknownOuter != __nullptr)
	{
		return CLASS_E_NOAGGREGATION;
	}
	// Создать компонент
	InventorNative::CAddInServerIUnk* pAddInServer =
	 new InventorNative::CAddInServerIUnk;

	// Вернуть запрошенный интерфейс
	HRESULT hr = pAddInServer->QueryInterface(iid, ppv);
	// Освободить указатель на IUnknown
	// (При ошибке в QueryInterface компонент разрушит сам себя)
	pAddInServer->Release();
	return hr;
}

HRESULT __stdcall CClassFactory::LockServer(BOOL bLock){ return S_OK; }

CClassFactory::CClassFactory() : m_cRef(1) {}
```

Приведем описание, что происходит в реализации фабрики классов. Функция ***QueryInterface*** возвращает в Inventor указатель, либо на ***IUnknown*** либо ***IClassFactory***. Inventor, используя, полученный указатель на интерфейс фабрики класса ***IClassFactory***, вызывает функцию ***CreateInstance***, для создания экземпляра класса самого компонента. Далее, используя параметр ***iid***, ***CreateInstance*** передает запрос на получения интерфейса самого компонента:

```
HRESULT hr = pAddInServer->QueryInterface(iid, ppv);
```

Этот интерфейс опять же возвращается Inventor для возможности вызова знакомых нам, по C# и VB.NET, функций: ***Activate***, ***Deactivate***, ***ExecuteCommand*** и ***get_Automation***. Только в нашем случае у этих функций будут префиксы.

Функция ***CreateInstance*** содержит параметр ***IUnknown* pUnknownOuter***, он служит для создания агрегатированных компонентов, это когда AddIn может содержать в себе еще один COM-компонент и предоставляет интерфейс для управления этим встроеным компонентом. Но у нас такого, нет и мы этим заниматься не будем. Поэтому этот параметр у нас всегда будет равен ***__nullptr***.

Функции ***AddRef*** и ***Release***, как и положено, управляют временем жизни компонента. При этом они используют функции ***InterlockedDecrement*** и ***InterlockedIncrement***. Эти функции обеспечивают обязательный инкремент и декремент при многопоточном подключении компонента.

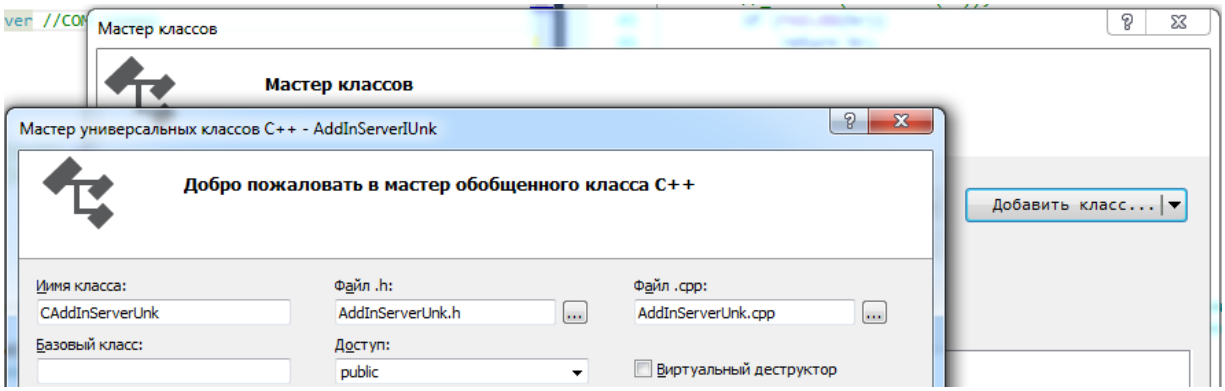
Функция ***LockServer*** обеспечиватет дополнительное блокирование выгрузки DLL через экспортируемую из нашей DLL функцию ***DllCanUnloadNow***. Реализация этой функции была произведена в файле ***Module.cpp***. Функция ***LockServer*** применяется когда компонент работает во внешнем процессе, а это не наш случай, поэтому мы её использовать не будем.

Последняя строка это конструктор «по умолчанию», который инициализирует счетчик ссылок:

```
CClassFactory::CClassFactory(): m_cRef(1) {}
```

Создание сервера подключения к Inventor *CAddInServerUnk*

Теперь переходим к написанию программного кода самого компонента. Для этого вызываем «Матер классов» и добавляем класс ***CAddInServerUnk***:



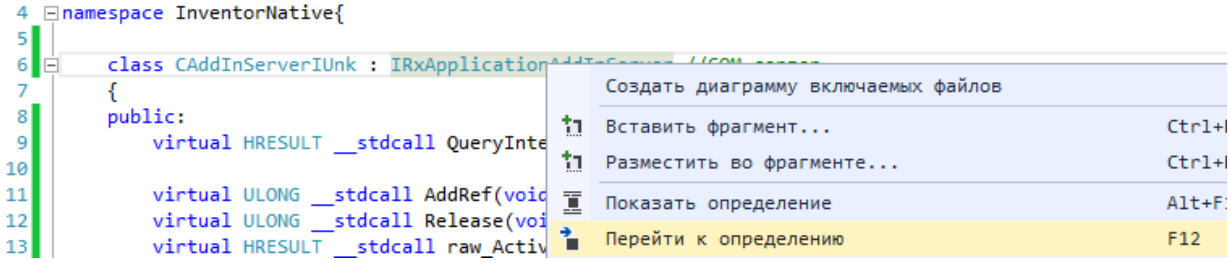
В заголовочном файле класса ***CAddInServerUnk.h***, пишем следующий код:

```
#pragma once

#include "Module.h"

namespace InventorNative{

	class CAddInServerIUnk : IRxApplicationAddInServer //COM сервер
	{
	public:
		virtual HRESULT __stdcall QueryInterface(const IID& iid,/*[in]*/
                                                 void **ppv); /*[iid_is][out]*/
		virtual ULONG __stdcall AddRef(void);
		virtual ULONG __stdcall Release(void);
		virtual HRESULT __stdcall raw_Activate(IRxApplicationAddInSite *
                                               pAddInSite, /*[in]*/
                                               BooleanType FirstTime);/*[in]*/
		virtual HRESULT __stdcall raw_Deactivate();
		virtual HRESULT __stdcall raw_ExecuteCommand(long CommandID);
		virtual HRESULT __stdcall get_Automation(IUnknown **ppResult);

		CAddInServerIUnk();
	private:
		long m_cRef;//счетчик ссылок
		Application * m_pInvApp;
	};
}
```

При создании этого класса просто был унаследован интерфейс ***IRxApplicationAddInServer***. Но возникает вопрос: если стандартные COM-интерфейсы ***IUnknown*** и ***IClassFactory*** всем хорошо известны, то откуда взять информацию о его существовании ***IRxApplicationAddInServer***? В хэлпе API Inventor этот объект отсутствует. Все очень просто, достаточно открыть файл ***rxinventor.tlh***. Этот файл находится внутри проекта и образуется после успешной компиляции проекта из файла библиотек типов ***RxInventor.tlb***, который был подключен через директиву ***#import***. Выполняем поиск на наличие текста ***ApplicationAddInServer***. Среди результатов поиска будет и ***IRxApplicationAddInServer***.

Так же нам встретится и интерфейс ***ApplicationAddInServer*** на базе интерфейса ***IDispatch***. Но о нем в следующей главе.

Приведу выдержку из файла ***rxinventor.tlh***. Можно быстро перейти к нужно описанию через контекстное меню:



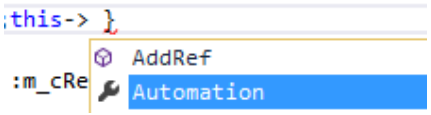
```
IRxApplicationAddInServer : IUnknown
{
    //
    // Property data
    //
    __declspec(property(get=GetAutomation))
    IUnknownPtr Automation;
    //
    // Wrapper methods for error-handling
    //
    HRESULT MethodActivate (
        struct IRxApplicationAddInSite * pAddInSite,
        BooleanType FirstTime );
    HRESULT MethodDeactivate ( );
    HRESULT MethodExecuteCommand (
        long CommandID );
    IUnknownPtr GetAutomation ( );
    //
    // Raw methods provided by interface
    //
    virtual HRESULT __stdcall raw_Activate (
        /*[in]*/ struct IRxApplicationAddInSite * pAddInSite,
        /*[in]*/ BooleanType FirstTime ) = 0;
    virtual HRESULT __stdcall raw_Deactivate ( ) = 0;
    virtual HRESULT __stdcall raw_ExecuteCommand (
        long CommandID ) = 0;
    virtual HRESULT __stdcall get_Automation (
        /*[out,retval]*/ IUnknown * * ppUnk ) = 0;
};
```

Из этого описания видно, что это объявление структуры, в которой реализован знакомый нам базовый COM-интерфейс: ***IUnknown***.

Так же здесь видно, как создаются, привычные для программистов на VB, свойства:

```
__declspec(property(get=GetAutomation))
    IUnknownPtr Automation;
```

В последствии, мы видим свойства отдельной строкой.



Для AddIn необходимо реализовать четыре виртуальные функции:

- ***raw_Activate***
- ***raw_Deactivate***,
- ***raw_ExecuteCommand***
- ***get_Automation***.

Что собственно и было сделано в фале ***CAddInServerUnk.h***.

Здесь же в описании уже определены еще четыре функции-обертки. Хотя одна из этих функций будет нужна для создания свойств, но нас они интересовать не будут, и мы их игнорируем.

```
#include "AddInServerIUnk.h"
#include <atlbase.h>

namespace InventorNative{

	HRESULT __stdcall CAddInServerIUnk::QueryInterface(const IID& iid, void **ppv)
	{
		if (iid == IID_IUnknown)
		{
			*ppv = static_cast<IUnknown*>(this);
		}
		else if (iid == IID_IRxApplicationAddInServer)
		{
			*ppv = static_cast<IRxApplicationAddInServer*>(this);
		}
		reinterpret_cast<IUnknown*>(*ppv)->AddRef();
		return S_OK;
	}

	ULONG __stdcall CAddInServerIUnk::AddRef()
	{ return InterlockedIncrement(&m_cRef); }

	ULONG __stdcall CAddInServerIUnk::Release()
	{
		if (InterlockedDecrement(&m_cRef) == 0)
		{ delete this; return 0; }
		return m_cRef;
	}

	HRESULT __stdcall CAddInServerIUnk::raw_Activate(IRxApplicationAddInSite *
pAddInSite, BooleanType FirstTime)
	{
	if (pAddInSite == NULL) return E_INVALIDARG;
	//Получить IUnknown
	IUnknown* pAppUnk;
	HRESULT hr = pAddInSite->get_Application(&pAppUnk);
	if (FAILED(hr)) return hr;
	//Получить объект-приложение
	hr = pAppUnk->QueryInterface(&m_pInvApp);
	//Получения названия
	CComBSTR bstrCaption;
	hr = m_pInvApp->get_Caption(&bstrCaption);
	//Создание текста сообщения
	CComBSTR bstrMsgText(L"IUnknown C++: Здравствуй ");
	bstrMsgText.AppendBSTR(bstrCaption);
	//Вывод окна
	int msgboxID = MessageBoxW(__nullptr, bstrMsgText, L"Приветствие", MB_OK);
	return S_OK;
	}

	HRESULT __stdcall CAddInServerIUnk::raw_Deactivate()
	{
	g_cComponents--;//уменьшаем счетчик работающих компонентов (выгрузка DLL)
	this->Release();//уменьшаем счетчик ссылок на компонент (уничтожение компонента)
		return S_OK;
	}
	//Устаревшая функция, на данный момент не используется
	HRESULT __stdcall CAddInServerIUnk::raw_ExecuteCommand(long CommandID)
	{ return S_OK; }

	HRESULT __stdcall CAddInServerIUnk::get_Automation(IUnknown **ppResult)
	{ ppResult = __nullptr;  return S_OK; }

	CAddInServerIUnk::CAddInServerIUnk() :m_cRef(1){}
}
```

Здесь мы видим сначала все туже реализацию интерфейса ***IUnknown***. Как и в C# и VB.NET в вызове функции ***raw_Activate*** передается управление над Inventor. После чего считывается заголовок окна Inventor и выводится на экран в виде сообщения.

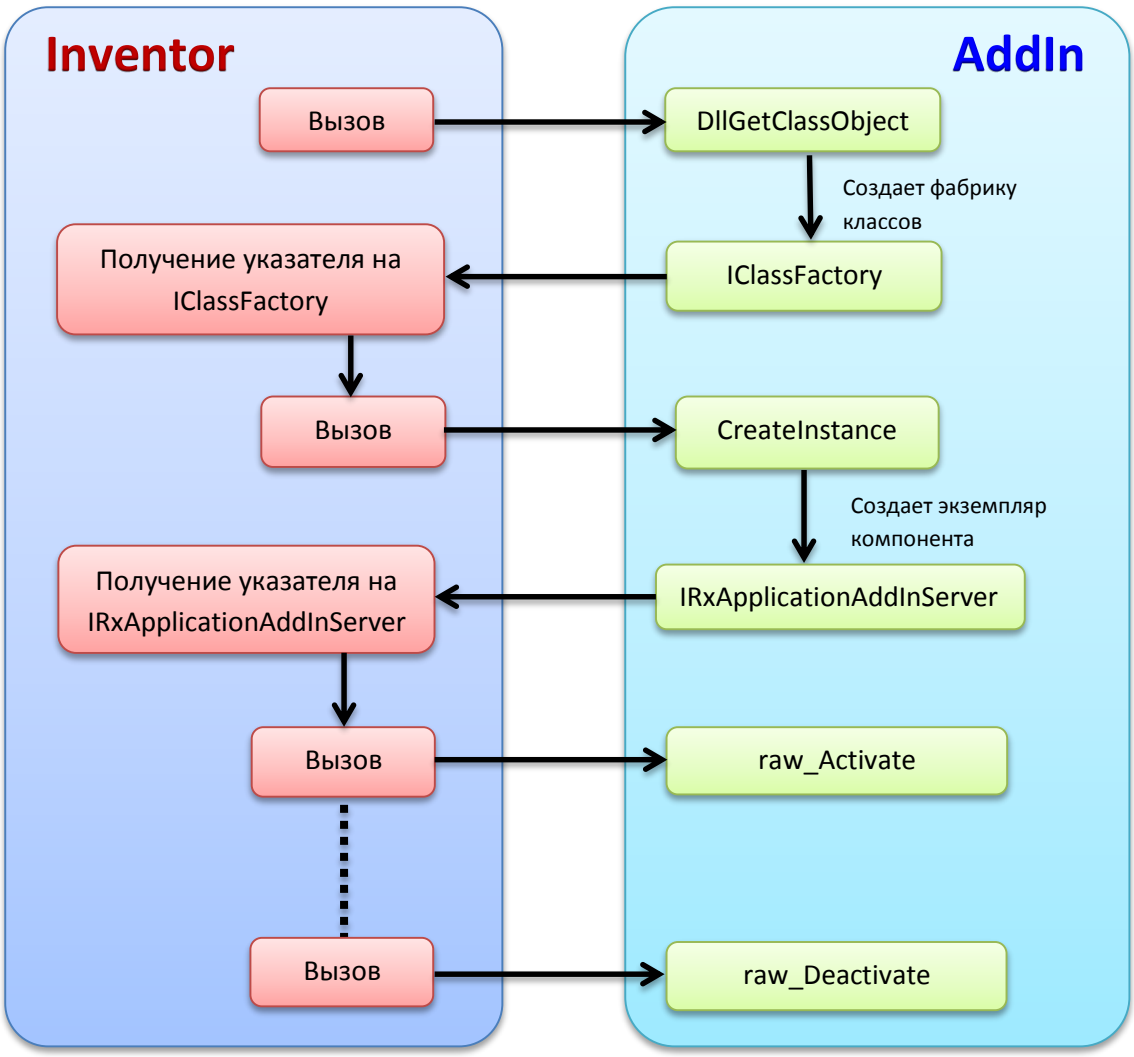
Функция ***raw_Deactivate*** уменьшает счетчик ссылок и дает разрешение на выгрузку DLL через глобальную переменную ***g_cComponents***.

По функции хорошо ***get_Automation*** видно, как должно управлять AddIn дополнительное стороннее приложение. Стороннее приложение должно будет получить указатель на интерфейс ***IUnknown***, который должен быть реализован для управления AddIn. В данный момент его нет, поэтому мы возвращаем пустой указатель.

И конструктор класса, как и в случае с фабрикой классов, используется для инициализации счетчика ссылок на компонент.

Схема вызовов

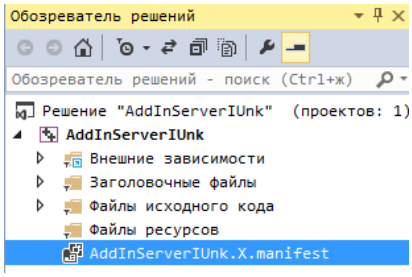
Программный код на этом закончен, теперь рассмотрим схему вызовов:



Манифест и *.addin*-файл

Теперь создадим файл манифеста. Напомню т.к. язык программирования у нас Visual C++, то для того что бы манифест автоматически внедрялся при компиляции нужно, что бы файл манифеста назывался в следующем формате: ***Имя DLL-файла.X.manifest***

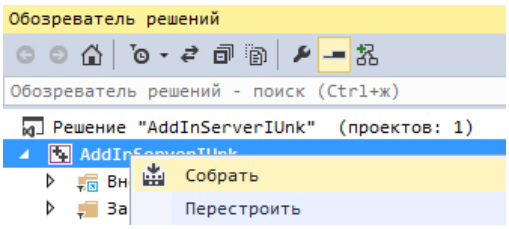
Создаем в коренном каталоге проекта файл (текстовый или в формате xml с расширением ***.manifest***) с названием ***AddInServerIUnk.X.manifest***. Добавляем его в проект:



Содержимое этого файла будет следующее:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity   type="win32" name="AddInServerIUnk" version="1.0.0.0" />
  <file name="AddInServerIUnk.dll">
    <comClass clsid="{3AD1A692-1E0F-4A12-9AD6-20B0512CE743}"
              progid="AddInServerIUnk.AddInServer"
              description="AddIn на интерфейсе IUnknown"
              threadingModel="Both"/>
  </file>
</assembly>
```

Все, можно компилировать.



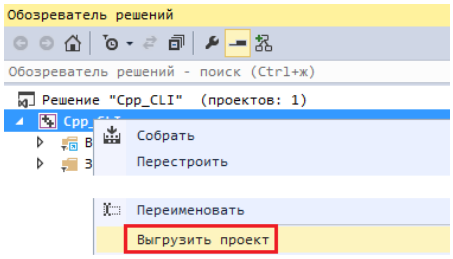
Теперь настала очередь создания ***.addin***-файла. Так же создаем в корневом каталоге файл (текстовый или в формате xml с расширением ***.addin***). Добавляем содержимое в этот файл:

```
<?xml version="1.0" encoding="utf-8"?>
<Addin Type="Standard">
    <!--Created for Autodesk Inventor Version 19.0-->
    <ClassId>{3AD1A692-1E0F-4A12-9AD6-20B0512CE743}</ClassId>
    <ClientId>{3AD1A692-1E0F-4A12-9AD6-20B0512CE743}</ClientId>
    <DisplayName>CppIUnlnown</DisplayName>
    <Description>AddIn на базе IUnknown</Description>
    <Assembly>D:\VSProjects\AddInServerIUnk\x64\Debug\AddInServerIUnk.dll</Assembly>
    <LoadOnStartUp>1</LoadOnStartUp>
    <UserUnloadable>1</UserUnloadable>
    <Hidden>0</Hidden>
    <SupportedSoftwareVersionGreaterThan>18..</SupportedSoftwareVersionGreaterThan>
    <DataVersion>1</DataVersion>
    <UserInterfaceVersion>1</UserInterfaceVersion>
</Addin>
```

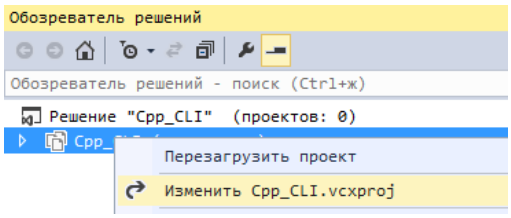
Останавливаться на содержимом ***.addin***-файла нет смысла, о нем уже все сказано в предыдущих главах. Напомню, что необходимо в ***теге <Assembly>*** прописать полный путь к откомпилированному DLL-файлу. И положить этот ***.addin***-файл в одну из следующих папок:

1. Для всех пользователей не зависимо от версии Inventor
%ALLUSERSPROFILE%\Autodesk\Inventor Addins
2. Для всех пользователей в зависимости от версии Inventor
%ALLUSERSPROFILE%\Autodesk\Inventor 2015\Addins
3. Для конкретного пользователя в зависимости от версии Inventor
%APPDATA%\Autodesk\Inventor 2015\Addins
4. Для конкретного пользователя не зависимо от версии Inventor
%APPDATA%\Autodesk\ApplicationPlugins

Проверяем назначенную версию Framework, для этого выгружаем проект, используя контекстное меню ***«Обозревателя решений»***:



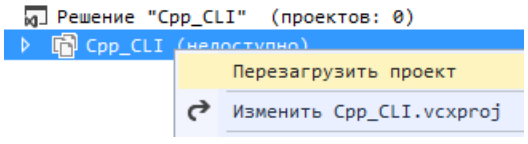
Здесь, то же через контекстное меню, загружаем файл проекта на редактирование:



Ищем тэг: ***TargetFrameworkVersion***:

```
<TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
```

В нашем случае все нормально – версия Framework проекта совпадает с версией Framework в файле ***Inventor.exe.config***, о котором было написано выше. Закрываем файл проекта и загружаем проект снова в «Обозревателе решения» через контекстное меню:



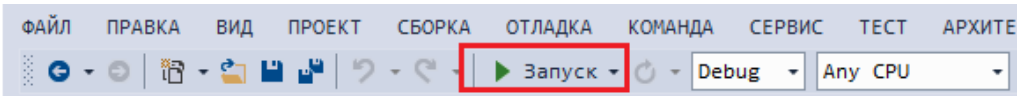
Устанавливаем точку останова в функции сначала на функции ***DllMain***, которая будет вызываться самая первая:

```
12
13  BOOL __stdcall  DllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved)
14  {
15      if (dwReason == DLL_PROCESS_ATTACH)
16      {
```

Так же устанавливаем точку останова в функции ***raw_Activate***:

```
34  HRESULT __stdcall CAddInServerIUnk::raw_Activate(IRxApplicationAddInSite * pAddInSite, BooleanType FirstTime)
35  {
36      if (pAddInSite == NULL) return E_INVALIDARG;
37      //Получить IUnknown
38      IUnknown* pAppUnk;
39      HRESULT hr = pAddInSite->get_Application(&pAppUnk);
40      if (FAILED(hr)) return hr;
41      //Получить объект-приложение
42      hr = pAppUnk->QueryInterface(&m_pInvApp);
43      //Получения названия
44      CComBSTR bstrCaption;
45      hr = m_pInvApp->get_Caption(&bstrCaption);
46      //Создание текста сообщения
47      CComBSTR bstrMsgText(L"IUnknown C++: Здравствуй ");
48      bstrMsgText.AppendBSTR(bstrCaption);
49      //Вывод окна
50      int msgboxID = MessageBoxW(__nullptr, bstrMsgText, L"Приветствие", MB_OK);
51      return S_OK;
52  }
```

Теперь можно запускать на отладку:



Первый запуск отладки во время текущей сессии Visual Studio запуск может быть не очень быстрый, при последующих запусках эта проблема исчезает.

После того как исполнение программы дойдет до точки останова, можно будет производить отладку:

Первый останов будет на самой первой вызываемой функции:

```
12
13  BOOL __stdcall  DllMain(HMODULE hModule, DWORD dwReason, LPVOID lpReserved)
14  {
15      if (dwReason == DLL_PROCESS_ATTACH)
16      {
17          g_hModule = hModule;
18          g_cComponents++;
```

Локальные

Имя	Значение
hModule	0x000007fe00000001 {unused=??? }
dwReason	0
lpReserved	0x01d09c80abeb972d

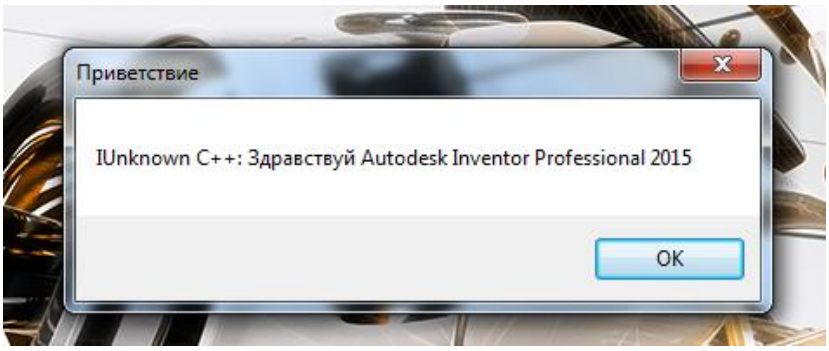
Далее программа остановится на второй точке:

```
49          //Вывод окна
50          int msgboxID = MessageBoxW(__nullptr, bstrMsgText, L"Приветствие", MB_OK);
51          return S_OK;
52      }
```

Локальные

Имя	Значение	Тип
this	0x000000010b3e9f0 {m_cRef=3 m_pInvApp=0x000000000cb999e0 <Информация	InventorNative::CAddIn
pAddInSite	0x0000000010b38dc0 <Информация недоступна, символы для RxAddIns.dll н	InventorNative::IRxApp
FirstTime	1 '\x1'	char
bstrCaption	L"Autodesk Inventor Professional 2015"	ATL::CComBSTR
hr	S_OK	HRESULT
bstrMsgText	L"IUnknown C++: Здравствуй Autodesk Inventor Professional 2015"	ATL::CComBSTR

Дальнейшее продолжение программы приведет к вызову приветствия:



Готовый пример можно скачать здесь.

Чистый «нативный» C++: AddIn на базе *IDispatch*

И так мы познакомились к способу подключения AddIn к Inventor через базовый COM-интерфейс ***IUnknown***. Однако существует альтернативный способ подключения AddIn. Этот способ основан на использовании COM-интерфейса ***IDispatch*** (автоматизация). ***IDispatch*** это интерфейс который является надстройкой над ***IUnknown***. Как я написал ранее, ***IDispatch*** изначально был создан для работы Visual Basic (и VBA). API Inventor, тоже разработан на базе ***IDispatch***. ***IDispatch*** призван помочь, программистам не высокого уровня, быстро создавать не большие макросы для мелкой автоматизации рутинных операций.

Но за неумело используемый «искусственный интеллект» ***IDispatch***, как всегда, приходится платить производительностью компьютера. К числу таких медленных инструментов ***IDispatch*** относятся: «позднее связывание» в VB, перекладывание определения типа результата выражений при использовании переменных разных типов.

Тем не менее если вы попробуете создать проект на C++ с использованием шаблона от Autodesk, то в мастере создания проекта можно убудет что рекомендуется использование именно интерфейса ***IDispatch***, а не ***IUnknown***. Лично мое мнение по этому вопросу такое: если приложение планируется небольшое, то лучше использовать ***IUnknown***, а если планируется, что-то более грандиозное с предоставлением возможности написания простых макроподобных программ, конечно, лучше использовать ***IDispatch***. Ну а если производительность компьютера не ограничивает, то лучший вариант — это конечно NET.

Описание интерфейса *IDispatch*

Перейдем к рассмотрению интерфейса ***IDispatch***. На C++ в версии от Microsoft ***IDispatch*** будет выглядеть вот так:

```
struct
	__declspec(uuid("00020400-0000-0000-C000-000000000046"))
	__declspec(novtable)
IDispatch : IUnknown
{
	virtual HRESULT __stdcall GetTypeInfoCount(UINT *pctinfo) = 0;/*[out]*/

	virtual HRESULT __stdcall GetTypeInfo(UINT iTInfo,/*[in]*/
	                                      LCID lcid,/*[in]*/
	                                      ITypeInfo **ppTInfo) = 0; /*[out]*/

	virtual HRESULT __stdcall GetIDsOfNames(const IID & riid,/*[in]*/
	                                        LPOLESTR *rgszNames,/*[size_is][in]*/
	                                        UINT cNames,/*[range][in]*/
	                                        LCID lcid,/*[in]*/
	                                        DISPID *rgDispId) = 0;/*[size_is][out]*/

	virtual HRESULT __stdcall Invoke(DISPID dispIdMember,/*[in]*/
	                                 const IID & riid,/*[in]*/
	                                 LCID lcid,/*[in]*/
	                                 WORD wFlags,/*[in]*/
	                                 DISPPARAMS *pDispParams,/*[out][in]*/
	                                 VARIANT *pVarResult,/*[out]*/
	                                 EXCEPINFO *pExcepInfo,/*[out]*/
	                                 UINT *puArgErr) = 0;/*[out]*/

};
```

Т.е. добавляется четыре функции:

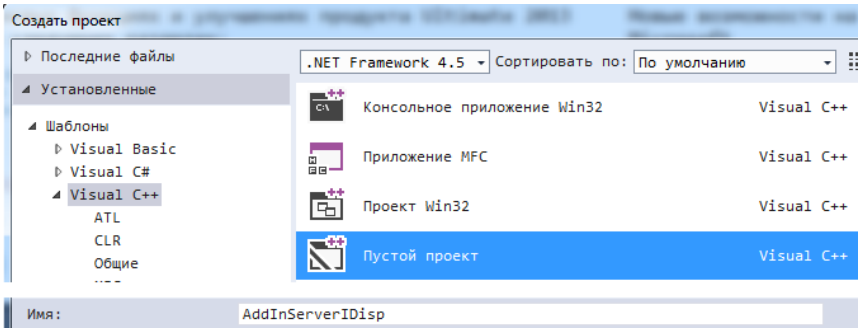
- ***GetTypeInfoCount***
  данная функция оповещает будет ли предоставляться информация о типах или нет, в нашем случае использоваться не будет.

- ***GetTypeInfo***
  Данная функция предоставляет информацию о реализуемых типах в данном ***IDispatch***, так же не будет нами использоваться.
- ***GetIDsOfNames***
  данная функция преобразует имя функций из библиотеки типов в порядковый номер, ***DISPID*** это просто порядковый номер (не путать с 128-ми битным GUID), для подключения к Inventor эта функция использоваться не будет.
- ***Invoke***
- данная функция производит вызов функций, в данном случае, Inventor будет просить нашу программу вызвать знакомые нам 4 функции: ***Activate***, ***Deactivate***, ***ExecuteCommand*** и ***GetAutomation***.

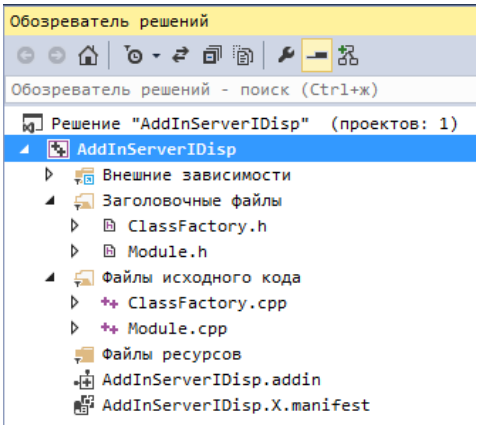
На первый взгляд параметров у этих четырех функций достаточно много, но нам практически ничего не придется реализовывать, только принять два параметра и распределить вызовы на 4 наши функции.

Подготовка к созданию подключения к Inventor

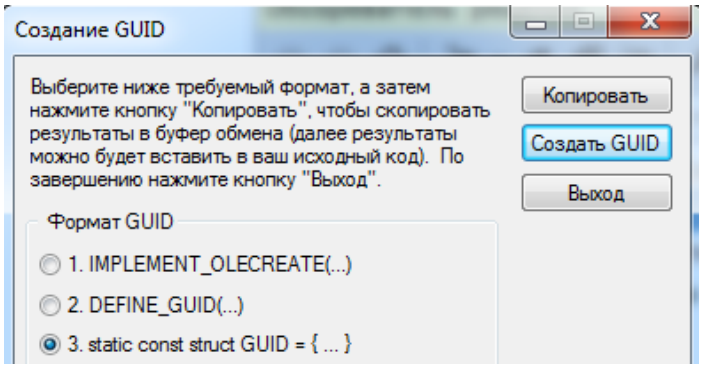
Все выше сказанное про интерфейс ***IUnknown***, в предыдущем разделе до фабрики классов включительно (***IClassFactory***), справедливо и для ***IDispatch***. Поэтому сделаем новый проект на базе пустого проекта C++:



Повторяем все шаги, вплоть до создания файлов ***CAddInServerUnk.h*** и ***CAddInServerUnk.cpp***. Можно скопировать часть файлов из предыдущего проекта с переименованием файла манифеста и ***.addin***-файла. Полученный проект должен выглядеть примерно так:



Сгенерируем новый GUID:



и заменим его значение в файлах ***Module.cpp*** (мой вариант):

```
//CLSID компонента {24AE65B5-65A9-40C9-9D5C-20DAD4738559}
const GUID CLSID_component =
       { 0x24ae65b5, 0x65a9, 0x40c9, { 0x9d, 0x5c, 0x20, 0xda, 0xd4, 0x73, 0x85, 0x59 } };
```

Манифест и *.addin*-файл

А так же заменим GUID в файле манифеста, заодно изменим и другие значения тегов. Файл ***AddInServerIDisp.X.manifest***:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity   type="win32" name="AddInServerIDisp" version="1.0.0.0" />
  <file name="AddInServerIDisp.dll">
    <comClass clsid="{24AE65B5-65A9-40C9-9D5C-20DAD4738559}"
              progid="AddInServerIDisp.AddInServer"
              description="AddIn на интерфейсе IDisp"
              threadingModel="Both"/>
  </file>
</assembly>
```

В ***.addin***-файл должен выглядеть так ***AddInServerIDisp.addin***:

```
<?xml version="1.0" encoding="utf-8"?>
<Addin Type="Standard">
    <!--Created for Autodesk Inventor Version 19.0-->
    <ClassId>{24AE65B5-65A9-40C9-9D5C-20DAD4738559}</ClassId>
    <ClientId>{24AE65B5-65A9-40C9-9D5C-20DAD4738559}</ClientId>
    <DisplayName>CppIDisp</DisplayName>
    <Description>AddIn на базе IDispatch</Description>
    <Assembly>D:\VSProjects\AddInServerIDisp\x64\Debug\AddInServerIDisp.dll</Assembly>
    <LoadOnStartUp>1</LoadOnStartUp>
    <UserUnloadable>1</UserUnloadable>
    <Hidden>0</Hidden>
    <SupportedSoftwareVersionGreaterThan>18..</SupportedSoftwareVersionGreaterThan>
    <DataVersion>1</DataVersion>
    <UserInterfaceVersion>1</UserInterfaceVersion>
 </Addin>
```

Напомню, что тег ***<Assembly>*** будет окончательно определен после компиляции проекта.

Реализация функции *CreateInstance* фабрики классов для *IDispatch*

В файле реализации фабрики классов ***ClassFactory.cpp*** заменяем подключаемый файл:

```
#include "ClassFactory.h"
#include "AddInServerIDisp.h"
```

И заменяем реализацию функции ***CreateInstance***:

```
HRESULT __stdcall CClassFactory::CreateInstance(IUnknown* pUnknownOuter,
                                                const IID& iid,
                                                void** ppv)
{       // Агрегирование не поддерживается
        if (pUnknownOuter != __nullptr)
        { return CLASS_E_NOAGGREGATION;}
        // Создать компонент
        InventorNative::CAddInServerIDisp* pAddInServer =
                new InventorNative::CAddInServerIDisp;
        HRESULT hr = NOERROR;
        // Вернуть запрошенный интерфейс
         hr = pAddInServer->QueryInterface(iid, ppv);
        // Освободить указатель на IUnknown
        // (При ошибке в QueryInterface компонент разрушит сам себя)
        pAddInServer->Release();
        return hr;
}
```

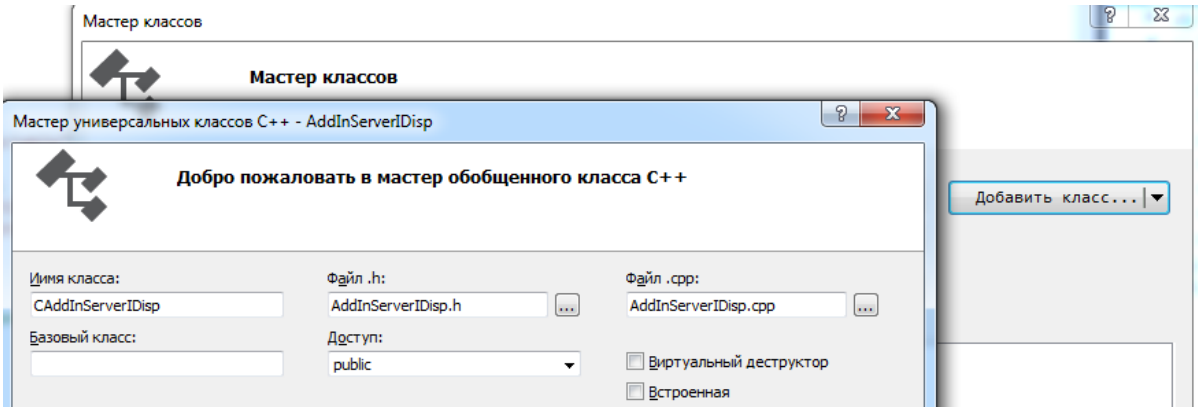
Нам продолжает мешать отсутствие файла ***rxinventor.tlh***, можно временно закомментировать проблемные участки в реализации функции ***CreateInstance***:

```
HRESULT __stdcall CClassFactory::CreateInstance(IUnknown* pUnknownOuter,
                                                Const IID& iid,
                                                void** ppv)
{       // Агрегирование не поддерживается
        if (pUnknownOuter != __nullptr)
        {return CLASS_E_NOAGGREGATION;}
        // Создать компонент
        //InventorNative::CAddInServerIDisp* pAddInServer =
                new InventorNative::CAddInServerIDisp;
        HRESULT hr = NOERROR;
        // Вернуть запрошенный интерфейс
        // hr = pAddInServer->QueryInterface(iid, ppv);
        // Освободить указатель на IUnknown
        // (При ошибке в QueryInterface компонент разрушит сам себя)
        //pAddInServer->Release();
        return hr;
}
```

И произвести компиляцию. Не забудьте раскомментировать программный код.

Создание сервера подключения к Inventor *ApplicationAddInServer*

Далее при помощи «Мастера классов» создаем новый класс ***CAddInServerIDisp***:



Содержимое файла ***CAddInServerIDisp.h***:

```
#pragma once
#include "Module.h"
#include  <atlbase.h>
namespace InventorNative{
	class CAddInServerIDisp : ApplicationAddInServer { // IDispatch
	public:
		CAddInServerIDisp();
		//IUnknown
		virtual HRESULT __stdcall QueryInterface(const IID& iid,/*[in]*/
                                                 void **ppv); /*[iid_is][out]*/
		virtual ULONG __stdcall AddRef(void);
		virtual ULONG __stdcall Release(void);
		//IDispath
		virtual HRESULT __stdcall GetTypeInfoCount(UINT *pctinfo);/*[out]*/

		virtual HRESULT __stdcall GetTypeInfo(UINT iTInfo,/*[in]*/
                                              LCID lcid, /*[in]*/
                                             ITypeInfo **ppTInfo); /*[out]*/

		virtual HRESULT __stdcall GetIDsOfNames( const IID& riid,/*[in]*/
                                                 LPOLESTR *rgszNames,/*[size_is][in]*/
                                                 UINT cNames,/*[range][in]*/
                                                 LCID lcid,/*[in]*/
                                                 DISPID *rgDispId);/*[size_is][out]*/

		virtual  HRESULT __stdcall Invoke( DISPID dispIdMember,/*[in]*/
                                           const IID& riid,/*[in]*/
                                           LCID lcid,/*[in]*/
                                           WORD wFlags,/*[in]*/
                                           DISPPARAMS *pDispParams,/*[out][in]*/
                                           VARIANT *pVarResult,/*[out]*/
                                           EXCEPINFO *pExcepInfo,/*[out]*/
                                           UINT *puArgErr);/*[out]*/
		//Методы Инвентора
		HRESULT __stdcall MethodActivate(ApplicationAddInSite * pAddInSite,
                                         VARIANT_BOOL FirstTime);
		HRESULT __stdcall MethodDeactivate();
		HRESULT __stdcall MethodExecuteCommand(long CommandID);
		IDispatchPtr __stdcall  GetAutomation();
	private:
		long m_cRef;//счетчик ссылок
		Application * m_pInvApp;//Ссылка на приложение
	};
	}
```

Здесь я, как и в случае с ***IUnknown***, воспользовался исследованием файла ***rxinventor.tlh*** на предмет наличия готового интерфейса ***ApplicationAddInServer*** на базе ***IDispatch***. И такой интерфейс нашелся, вот его описание:

```
struct __declspec(uuid("e3571293-db40-11d2-b783-0060b0f159ef"))
ApplicationAddInServer : IDispatch
{
    //
    // Property data
    //
    __declspec(property(get=GetAutomation))IDispatchPtr Automation;
    //
    // Wrapper methods for error-handling
    //
    // Methods:
    HRESULT MethodActivate (struct ApplicationAddInSite * AddInSiteObject,
                            VARIANT_BOOL FirstTime );
    HRESULT MethodDeactivate ( );
    HRESULT MethodExecuteCommand ( long CommandID );
    IDispatchPtr GetAutomation ( );
    //
    // Wrapper methods without error-handling
    //
    // Methods:
   HRESULT raw_Activate ( struct ApplicationAddInSite * AddInSiteObject,
        VARIANT_BOOL FirstTime );
   HRESULT raw_Deactivate ( );
   HRESULT raw_ExecuteCommand (long CommandID );
   HRESULT get_Automation ( IDispatch * * _result = 0 );
};
```

В этом интерфейсе нет виртуальных функций, т.к для ***IDispatch*** дополнительный набор виртуальных функций не нужен. Тогда на этот раз Autodesk дает нам на выбор, какой набор функций мы будем использовать. На этот раз мы выберем функции с префиксом ***Method***, т.к. это не увеличит код программы и вместе с тем обеспечит работу через свойство:

```
__declspec(property(get=GetAutomation))IDispatchPtr Automation;
```

Приведем реализацию объявленных функций из ***CAddInServerIDisp.h*** в файле ***AddInServerIDisp.cpp***:

```
#include "AddInServerIDisp.h"
namespace InventorNative
{

	//Конструктор
	CAddInServerIDisp::CAddInServerIDisp() :m_cRef(1){}

	//IUnknown
	HRESULT __stdcall CAddInServerIDisp::QueryInterface(const IID& iid,

	void **ppv)
	{
		if (iid == IID_IUnknown)
		{
			*ppv = static_cast<IUnknown*>(this);
		}
		else if (iid == IID_IDispatch)
		{
			*ppv = static_cast<IDispatch*>(this);
		}
		else if (iid == DIID_ApplicationAddInServer)
		{
			*ppv = static_cast<ApplicationAddInServer*>(this);
		}
		else
		{
			*ppv = __nullptr; return E_NOINTERFACE;
		}
		reinterpret_cast<IUnknown*>(*ppv)->AddRef();
		return S_OK;
	}

	ULONG __stdcall CAddInServerIDisp::AddRef()
	{
		return InterlockedIncrement(&m_cRef);
	}

	ULONG __stdcall CAddInServerIDisp::Release()
	{
		if (InterlockedDecrement(&m_cRef) == 0)
		{
			delete this;
			return 0;
		}
		return m_cRef;
	}

	//IDispath
	HRESULT __stdcall CAddInServerIDisp::GetTypeInfoCount(UINT *pctinfo)/*[out]*/
	{ *pctinfo = 1; return S_OK; }

	HRESULT __stdcall CAddInServerIDisp::GetTypeInfo(UINT iTInfo,/*[in]*/
	                                                 LCID lcid,/*[in]*/
	                                                 ITypeInfo **ppTInfo)/*[out]*/

	{
		return S_OK;
	}

	HRESULT __stdcall CAddInServerIDisp::GetIDsOfNames(const IID& riid,/*[in]*/
	                                                   LPOLESTR *rgszNames,/*[in]*/
	                                                   UINT cNames, /*[in]*/
	                                                   LCID lcid,/*[in]*/
	                                                   DISPID *rgDispId)/*[out]*/

	{
		return S_OK;
	}
```

```
	HRESULT __stdcall CAddInServerIDisp::Invoke(DISPID dispIdMember,/*[in]*/
						const IID& riid,/*[in]*/
						LCID lcid,/*[in]*/
						WORD wFlags,/*[in]*/
						DISPPARAMS *pDispParams,/*[out][in]*/
						VARIANT *pVarResult,/*[out]*/
						EXCEPINFO *pExcepInfo,/*[out]*/
						UINT *puArgErr)/*[out]*/
	{
		//Инвентор запрашивает вызов MethodActivate
		if (dispIdMember == 0x03001201)
		{
			VARIANT* pfirstTime = pDispParams->rgvarg;//FirstTime
			VARIANT*  pDispAddInSite = pfirstTime + 1;//ApplicationAddInSite
			ApplicationAddInSite * pAddInSite =
			reinterpret_cast<ApplicationAddInSite*>(pDispAddInSite->pdispVal);
			this->MethodActivate(pAddInSite, pfirstTime->boolVal);
		}
		//Инвентор запрашивает вызов MethodDeactivate
		else if (dispIdMember == 0x03001202)
		{ this->MethodDeactivate(); }
		//Инвентор запрашивает вызов MethodExecuteCommand
		else if (dispIdMember == 0x03001203)
		{ this->MethodExecuteCommand(0); }//Устаревший метод (не используется)
		//Инвентор запрашивает вызов GetAutomation
		else if (dispIdMember == 0x03001204)
		{
			VariantInit(pVarResult);
			pVarResult->vt = VT_DISPATCH;
			pVarResult->pdispVal = this->GetAutomation();
		}
		return S_OK;
	}

	HRESULT __stdcall CAddInServerIDisp::MethodActivate(ApplicationAddInSite * pAddInSite,
							VARIANT_BOOL FirstTime)
	{
		HRESULT hr = pAddInSite->get_Application(&m_pInvApp);
		HRESULT Result = NOERROR;
		//Получения названия
		CComBSTR bstrCaption;
		Result = m_pInvApp->get_Caption(&bstrCaption);
		//Создание текста сообщения
		CComBSTR bstrMsgText(L"IDispatch C++: Здравствуй ");
		bstrMsgText.AppendBSTR(bstrCaption);
		//Вывод окна
		int msgboxID = MessageBoxW(__nullptr, bstrMsgText, L"Приветствие", MB_OK);
		return S_OK;
	}

	HRESULT __stdcall CAddInServerIDisp::MethodDeactivate()
	{ return S_OK; }

	HRESULT __stdcall CAddInServerIDisp::MethodExecuteCommand(long CommandID)
	{ return S_OK; }

	IDispatchPtr __stdcall  CAddInServerIDisp::GetAutomation()
	{ return __nullptr; }
}// Конец пространства имен InventorNative
```

Вызов CAddInServerIDisp::Invoke

Интерес для рассмотрения представляет реализация ***Invoke***, вынесем этот код отдельно:

```
HRESULT __stdcall CAddInServerIDisp::Invoke(DISPID dispIdMember,/*[in]*/
                                            const IID& riid,/*[in]*/
                                            LCID lcid,/*[in]*/
                                            WORD wFlags,/*[in]*/
                                            DISPPARAMS *pDispParams,/*[out][in]*/
                                            VARIANT *pVarResult,/*[out]*/
                                            EXCEPINFO *pExcepInfo,/*[out]*/
                                            UINT *puArgErr)/*[out]*/
{
        //Инвентор запрашивает вызов MethodActivate
        if (dispIdMember == 0x03001201)
        {
                VARIANT* pfirstTime = pDispParams->rgvarg;//FirstTime
                VARIANT*  pDispAddInSite = pfirstTime + 1;//ApplicationAddInSite
                ApplicationAddInSite * pAddInSite =
                reinterpret_cast<ApplicationAddInSite*>(pDispAddInSite->pdispVal);
                this->MethodActivate(pAddInSite, pfirstTime->boolVal);
        }
        //Инвентор запрашивает вызов MethodDeactivate
        else if (dispIdMember == 0x03001202)
        { this->MethodDeactivate(); }
        //Инвентор запрашивает вызов MethodExecuteCommand
        else if (dispIdMember == 0x03001203)
        { this->MethodExecuteCommand(0); }//Устаревший метод (не используется)
        //Инвентор запрашивает вызов GetAutomation
        else if (dispIdMember == 0x03001204)
        {
                VariantInit(pVarResult);
                pVarResult->vt = VT_DISPATCH;
                pVarResult->pdispVal = this->GetAutomation();
        }
        return S_OK;
}
```

Inventor присылает запрос на вызов нужной функции при помощи параметра ***dispIdMember***. Как я писал выше ***dispIdMember*** это не 128-ми битный идентификатор, а порядковый номер в disp-интерфейсе. В нашем случае для ***dispIdMember*** это будут следующие числа:

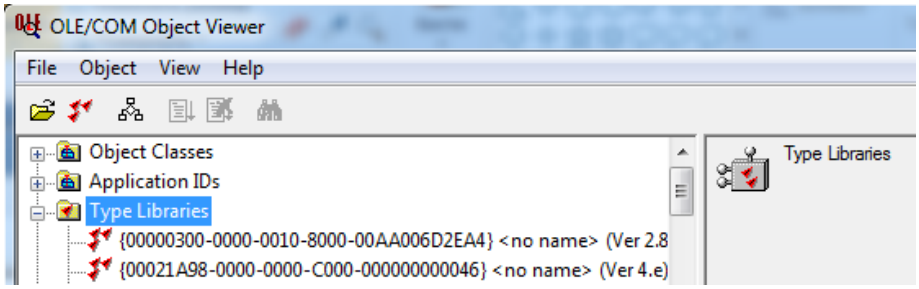
- ***MethodActivate***: 0x03001201
- ***MethodDeactivate***: 0x03001202
- ***MethodExecuteCommand***: 0x03001203
- ***GetAutomation***: 0x03001204

Утилита *oleview.exe* и IDL

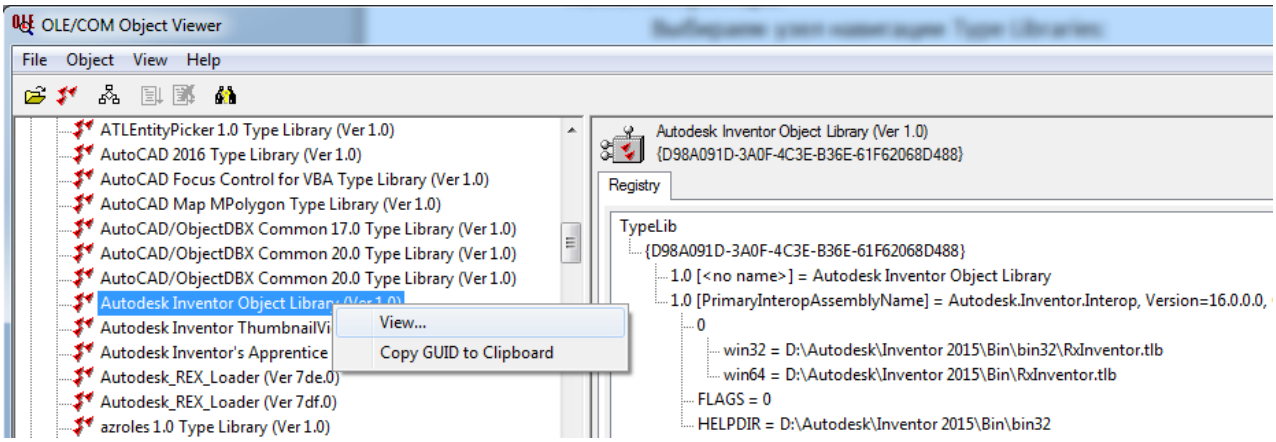
Вопрос: Откуда я узнал эти числа?
Ответ: Из IDL-файла описания интерфейсов.

Вопрос: А где взять файл IDL?
Ответ: Файл IDL умеет генерировать утилита ***oleview.exe***, которая есть в составе Visual Studio.

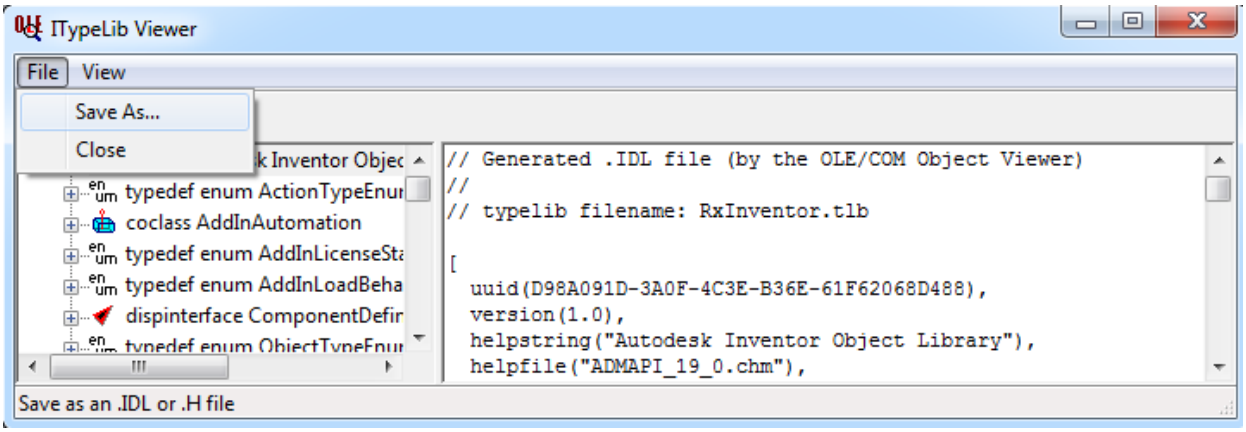
Посмотрим, как ***oleview.exe*** это делает. Открываем ***oleview.exe*** от имени ***Администратора***. Выбираем узел навигации Type Libraries:



В нем находим библиотеку типов от Inventor, и вызываем команду ***View***:



После определенных «раздумий» ***oleview.exe*** сгенерирует документ с описанием интерфейсов, сохраним его как IDL-файл с названием ***RxInventor.IDL***:



Откроем ***RxInventor.IDL*** при помощи Visual Studio и поищем упоминание о интерфейсе который мы унаследовали ***ApplicationAddInServer***. Вот результат:

```
[
      uuid(E3571293-DB40-11D2-B783-0060B0F159EF),
      helpstring("Object required to be supported by Server to qualify as a Inventor AddIn"),
      helpcontext(0x03001200)
    ]
    dispinterface ApplicationAddInServer {
        properties:
        methods:
            [id(0x03001201), helpstring("Invoked by Inventor after creating the AddIn. AddIn
should initialize within this call"), helpcontext(0x03001201)]
            void Activate(
                            [in] ApplicationAddInSite* AddInSiteObject,
                            [in] VARIANT_BOOL FirstTime);
            [id(0x03001202), helpstring("Invoked by Inventor to shutdown the AddIn. AddIn
should complete shutdown within this call"), helpcontext(0x03001202)]
            void Deactivate();
            [id(0x03001203), helpstring("Invoked by Inventor in response to user requesting
the execution of an AddIn-supplied command. AddIn must perform the command within this call"),
helpcontext(0x03001203)]
            void ExecuteCommand(long CommandID);
            [id(0x03001204), propget, helpstring("Gets the IUnknown of the object implemented
inside the AddIn that supports AddIn-specific API"), helpcontext(0x03001204)]
            IDispatch* Automation();
           };
```

Здесь как раз определены те ***id***, которые Inventor передает в функцию ***Invoke***. Вообще в этом файле много информации, которая понадобится при дальнейшей работе с Inventor через его API.

Вернемся к описанию функции ***Invoke***. Еще одним параметр, который передает Inventor, является указатель на ***pDispParams***. ***pDispParams*** – это указатель на структуру, сама структура определенна в заголовочных файлах Windows как:

```
typedef struct tagDISPPARAMS
    {
    /* [size_is] */ VARIANTARG *rgvarg;
    /* [size_is] */ DISPID *rgdispidNamedArgs;
    UINT cArgs;
    UINT cNamedArgs;
    }  DISPPARAMS;
```

Здесь нас будет интересовать указатель на массив типа ***VARIANTARG *rgvarg***. ***VARIANTARG*** – это переименованный через подстановочный макрос тип ***VARIANT***. Параметр ***cArgs*** передает количество элементов массива. Для ***MethodActivate***: ***cArgs*** =2. Т.е. в первом элементе массива ***rgvarg*** приходит указатель на булевское значение для ***FirstTime***, а во втором элементе массива ***rgvarg*** находится указатель на ***ApplicationAddInSite***. Эти данные далее передаются в вызов функции ***MethodActivate***.

Конечно, чтобы реализовать ***IDispatch*** полностью пришлось бы еще потрудится. Особенно при реализации работы с AddIn со сторонней программой через функцию ***GetAutomation***. Но для подсоединения AddIn к Inventor этого будет достаточно.

Производим окончательную компиляцию:

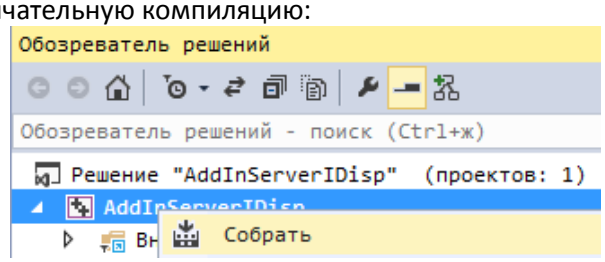
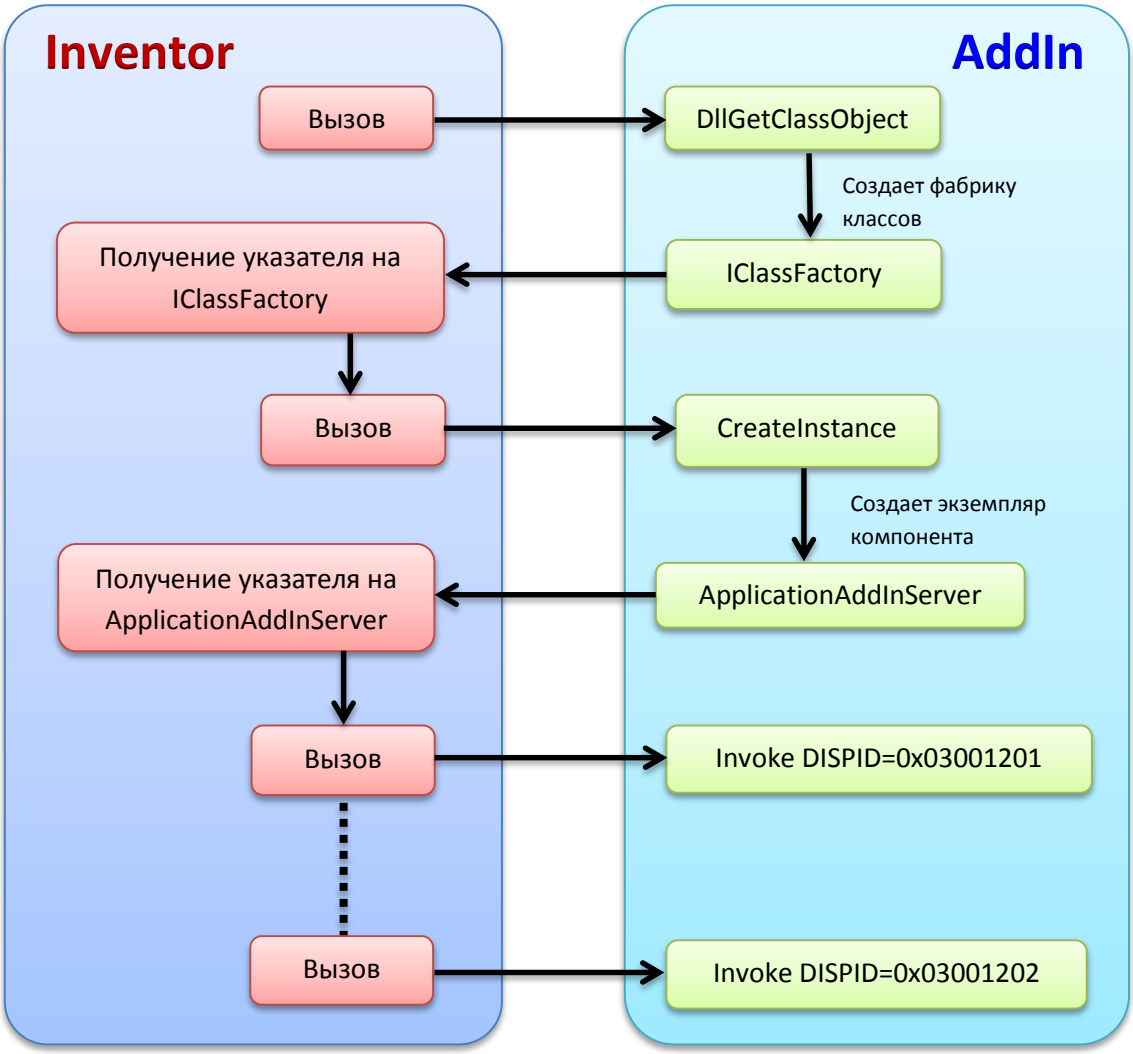


Схема вызовов

Схема вызовов для ***ApplicationAddInServer*** очень похожа на схему вызовов при реализации ***IUnknown***. Для ***IDispatch*** схема вызовов выглядит так:



Завершение и запуск

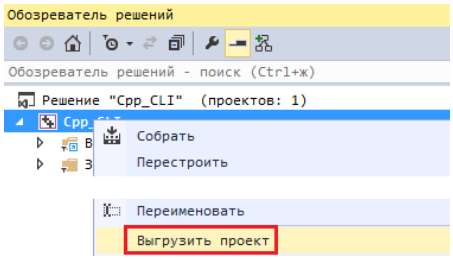
Прописываем полный путь к получившемуся DLL-файлу в ***.addin***-файле в теге ***<Assembly>***. В моем случае это так же и остается прежний путь, у вас может быть по другому:

```
<Assembly>D:\VSProjects\AddInServerIDisp\x64\Debug\AddInServerIDisp.dll</Assembly>
```

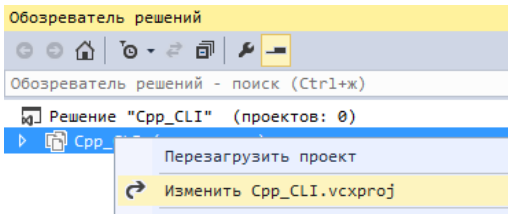
И копируем ***.addin***-файл в одну из папок путей поиска:

1. Для всех пользователей не зависимо от версии Inventor
%ALLUSERSPROFILE%\Autodesk\Inventor Addins
2. Для всех пользователей в зависимости от версии Inventor
%ALLUSERSPROFILE%\Autodesk\Inventor 2015\Addins
3. Для конкретного пользователя в зависимости от версии Inventor
%APPDATA%\Autodesk\Inventor 2015\Addins
4. Для конкретного пользователя не зависимо от версии Inventor
%APPDATA%\Autodesk\ApplicationPlugins

Проверяем назначенную версию Framework, для этого выгружаем проект, используя контекстное меню ***«Обозревателя решений»***:



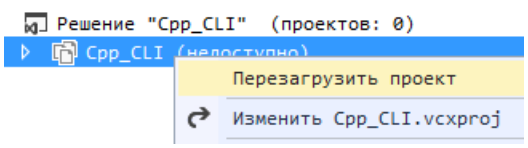
Здесь, то же через контекстное меню, загружаем файл проекта на редактирование:



Ищем тэг: ***TargetFrameworkVersion*:**

```
<TargetFrameworkVersion>v4.5</TargetFrameworkVersion>
```

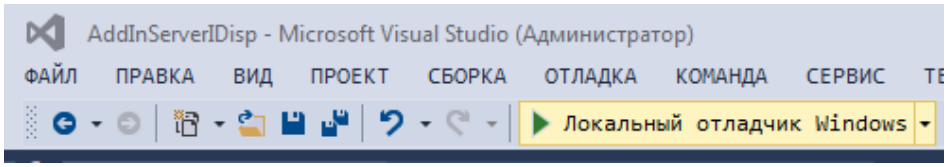
В нашем случае все нормально – версия Framework проекта совпадает с версией Framework в файле ***Inventor.exe.config***, о котором было написано выше. Закрываем файл проекта и загружаем проект снова в «Обозревателе решения» через контекстное меню:



Устанавливаем точку останова в функции ***MethodActivate***:

```
96    HRESULT __stdcall CAddInServerIDisp::MethodActivate(ApplicationAddInSite * pAddInSite, VARIANT_BOOL FirstTime)
97    {
98        HRESULT hr = pAddInSite->get_Application(&m_pInvApp);
99        HRESULT Result = NOERROR;
100       //Получения названия
101       CComBSTR bstrCaption;
102       Result = m_pInvApp->get_Caption(&bstrCaption);
103       //Создание текста сообщения
104       CComBSTR bstrMsgText(L"IDispatch C++: Здравствуй ");
105       bstrMsgText.AppendBSTR(bstrCaption);
106       //Вывод окна
107       int msgboxID = MessageBoxW(__nullptr, bstrMsgText, L"Приветствие", MB_OK);
108       return S_OK;
109   }
```

И запускаем AddIn на отладку:



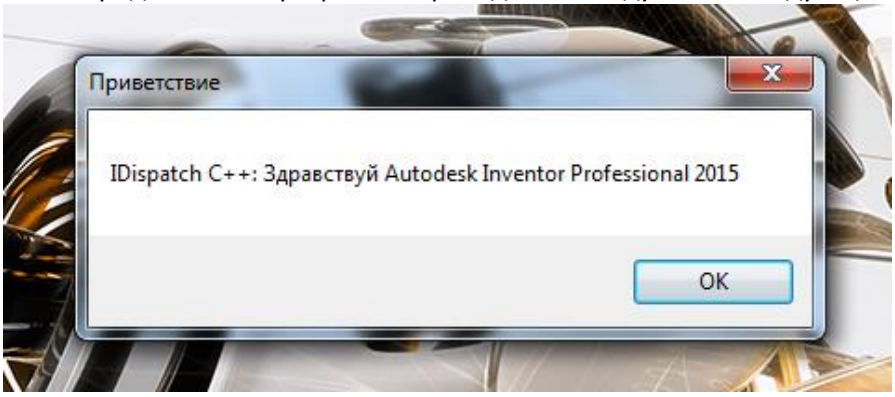
Программа останавливается в точке останова, где мы можем производить отладку:

```
105       bstrMsgText.AppendBSTR(bstrCaption);
106       //Вывод окна
107       int msgboxID = MessageBoxW(__nullptr, bstrMsgText, L"Приветствие", MB_OK);
108       return S_OK;
109   }
110
```

Локальные

Имя	Значение
this	0x0000000010864480 {m_cRef=3 m_pInvApp=0x00000000c7e05a0 <Информация недоступна, си
pAddInSite	0x00000000107e63f8 <Информация недоступна, символы для RxAddIns.dll не загружены>
FirstTime	-1
bstrCaption	L"Autodesk Inventor Professional 2015"
hr	S_OK
bstrMsgText	L"IDispatch C++: Здравствуй Autodesk Inventor Professional 2015"
Result	S_OK
msgboxID	-858993460

Дальнейшее продолжение программы приведет к выводу окна с следующим сообщением:



Готовый пример можно скачать здесь:

И в заключение хочу сказать пару слов, по поводу переносимости скомпилированных файлов (DLL и EXE) «нативного» C++, на другие компьютеры. Может возникнуть ситуация, когда на

компьютере, где установлен Visual Studio все запускается без проблем, а на другом компьютере программа будет отказываться работать. Это связано с тем, что на другом компьютере могут отсутствовать некоторые DLL-библиотеки, например, библиотеки типов CRT:

msvcr<номер версии>.dll

Что бы решить эту проблему необходимо либо на другом компьютере установить актуальную версию:

Visual C++ Redistributable Packages

либо включить необходимые библиотеки в статическую линковку, что бы полученный DLL-файл содержал в себе все необходимое.

М. Казаков
14.06.15