

Руководство пользователя

Том **10.** Разработка собственных команд и программных модулей на основе Топоматик **Robur**

Санкт-Петербург 2015

Содержание

Обзор средств автоматизации и разработки.....3

Разработка скриптов и макрокоманд (макросов).....4

Пример 1. Первый скрипт и его регистрация в Robur.....6
Разработка макроса "Привет, Robur!".....6
Регистрация скрипта в Robur.....8
Добавление элементов управления для команд.....12
Пример 2. Макрос, вставляющий примитив в текущий чертеж проекта Robur.....14
Пример 3. Выбор примитивов текущего чертежа и чтение их свойств.....18
Пример 4. Управлением механизма отмены команд Robur (Undo) и истории команд.....19
Пример 5. Работа с поверхностями Robur (ЦММ).....21
Пример 6. Работа с подобъектами Robur.....23

Разработка программных модулей.....25

Пример 1. Модуль, выводящий сообщение.....25
Пример 2. Модуль, вставляющий примитив в текущий чертеж проекта Robur.....51
Пример 3. Выбор примитивов текущего чертежа и чтение их свойств.....54
Пример 4. Управлением механизма отмены команд Robur (Undo) и истории команд.....56
Пример 5. Работа с поверхностями Robur (ЦММ).....57
Пример 6. Работа с подобъектами Robur.....60

Обзор средств автоматизации и разработки

В 2010 году был выпущен первый программный продукт компании Топоматик - Инструменты Топоматик Robur, разработанный на новой программной платформе **Топомтик AP3**.

Платформа AP3 (Application Platform 3) разработана на базе среды **Microsoft .NET** и является новым мощным ядром Robur, позволяющим наиболее полно использовать ресурсы современных компьютеров, объекмы памяти и, соответственно, значительно быстрее решать задачи больших объемов, которые были просто "неподъемны" для предыдущей версии платформы **Robur AP2** - представлявшей собой обычное **Windows** - приложение.

Также в платформе **Топоматик AP3**, по сравнению с ее предшественницей, значительно расширена возможность автоматизации работы с программой, написания собственных модулей, расширяющих функционал программного комплекса Robur и реализующих отсутствующие в нем функции или адаптирующих его работу в соответствии со стандартами предприятия или Вашей технологией проектирования.

С помощью **AP3** Вы можете:

- Быстро вызывать необходимые команды с нужными параметрами из командной строки. По просьбам наших пользователей была реализована консоль с командной строкой.
- Создавать макросы и скрипты, позволяющие выполнять любые последовательности команд и имеющие доступ как к функциям и данным самой платформы **Robur**, так и к элементам проекта. Вызывать такие макросы можно как из командной строки, так и с помощью элементов пользовательского интерфейса (кнопок, пунктов меню и т.п.). Подробнее работа с макросами (скриптами) рассмотрена в соответствующем *разделе.*
- Создавать и распространять программные модули различной сложности, расширяющие функциональность **Robur** или адаптирующие его работу под стандарты Вашего предприятия или требования заказчиков или контролирующих организаций. Модули разрабатываются с помощью любых средств программирования, поддерживающих среду **Microsoft .Net**, и могут распространяться Вами как внутри предприятия, так и как законченные коммерческие продукты. Подробнее создание модулей рассмотрена в соответствующем *разделе*.
- Использовать практически все функции и объекты **Топоматик Robur** и его компонентов для реализации собственных задач, как в описанных выше задачах автоматизации, так и при разработке собственных программ.

Разработка скриптов и макрокоманд (макросов)

Макросами называются несложные команды, вызываемые из командной строки, написанные на поддерживаемом платформой **Robur** языке (скрипте).

Макросы позволяют выполнять последовательности команд и других макросов, использовать требуемые параметры исключая их неоднократный ввод пользователями.

Макросы имеют доступ ко всем функциям и данным как самой платформы **Robur**, так и элементам текущего проекта.

Также макросы можно вызывать, связав их элементами пользовательского интерфейса (кнопками на панелях инструментов, пунктов меню и т.п.) с помощью настроек персонализации пользовательского интерфейса **Robur**.

Сами скрипты могут быть достаточно сложными и реализовывать серьезные функции и объекты, в том числе элементы пользовательского интерфейса (окна, диалоги, кнопки).

Примечание: для разработки сложных подсистем рекомендуется использовать программные модули, а не скрипты, так как большинство современных сред разработки для **.NET Framework** обладают отладчиками, редакторами диалогов и элементов пользовательских интерфейсов, отличные справочную систему и редактор.

По умолчанию в **Robur** встроена поддержка скриптов **Iron Python**, языка программирования **Python**, поддерживающего .Net Framework.

По желанию пользователи могут установить и настроить поддержку и других скриптовых языков программирования, поддерживающих .Net, например **Lisp**. Однако, Вы должны понимать, что даже, если Вы установите поддержку **Lisp**, то перенести Ваши наработки из **AutoCAD** или других продуктов AutoDesk без переработки Вам скорее всего не удастся, так как и объектная модель **Robur** значительно отличается от **AutoCAD** и программирование для .Net в самом **Lisp** значительно отличается по синтаксиса и приемам реализации задач. Единственное удобство - это аналогичный синтаксис самого языка. Поэтому, мы рекомендуем Вам реализовывать Ваши скрипты и макросы на **Python**, так как гарантируется его поддержка.

В данном руководстве все примеры скриптов и макросов будут реализованы на Iron **Python**.

Файлы скриптов **Python** представляют собой обычные текстовые файлы, сохраненные с расширением ".py".

Для редактирования скриптов Вы можете использовать Блокнот Windows (Windows Notepad), но удобства редактирования скриптов рекомендуется скачать и использовать бесплатную программу Notepad++.

В отличие от блокнота Windows (снимок показан на Рис. 1) в **Notepad++** есть подсветка синтаксиса **Python** и крайне удобно выставляются отступы (внешний вид окна **Notepad++** показан на Рис. 2).

Рис. 1. Окно блокнота Windows

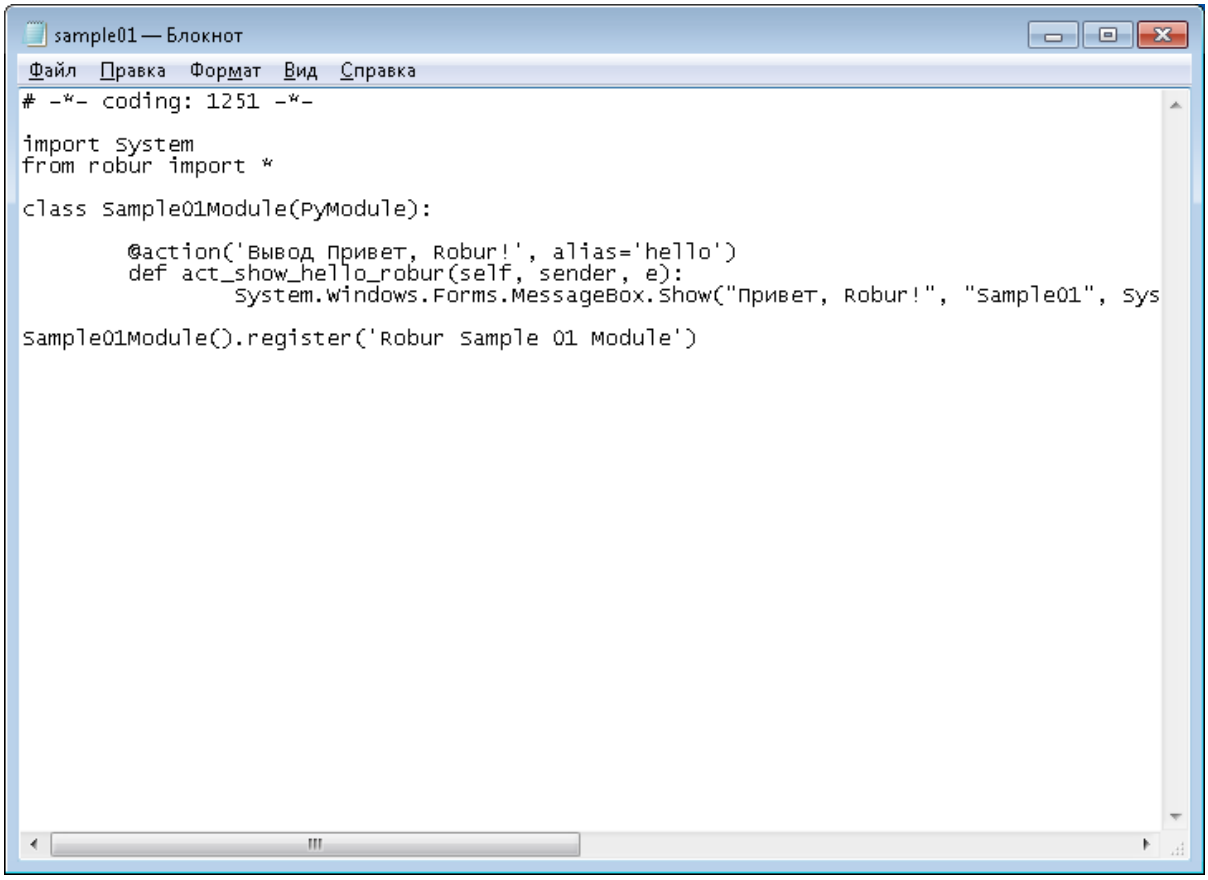
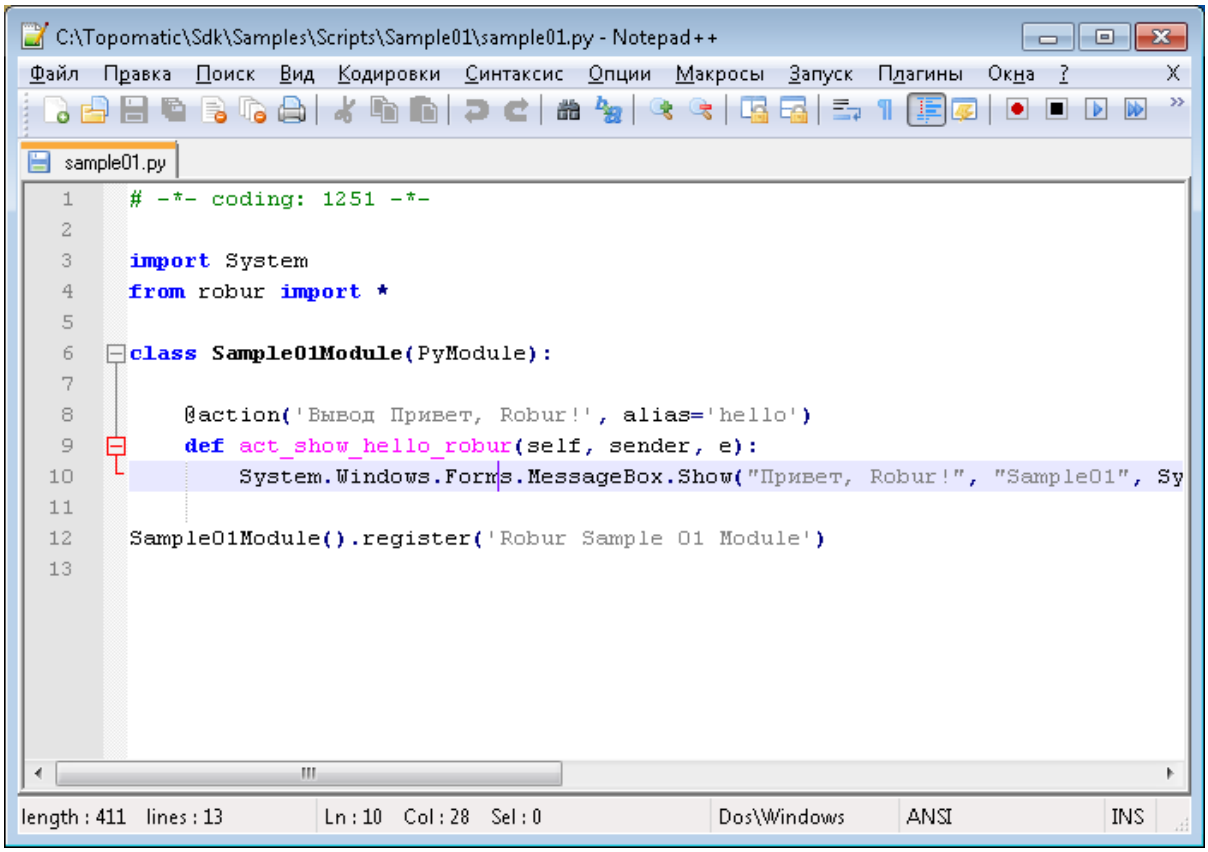
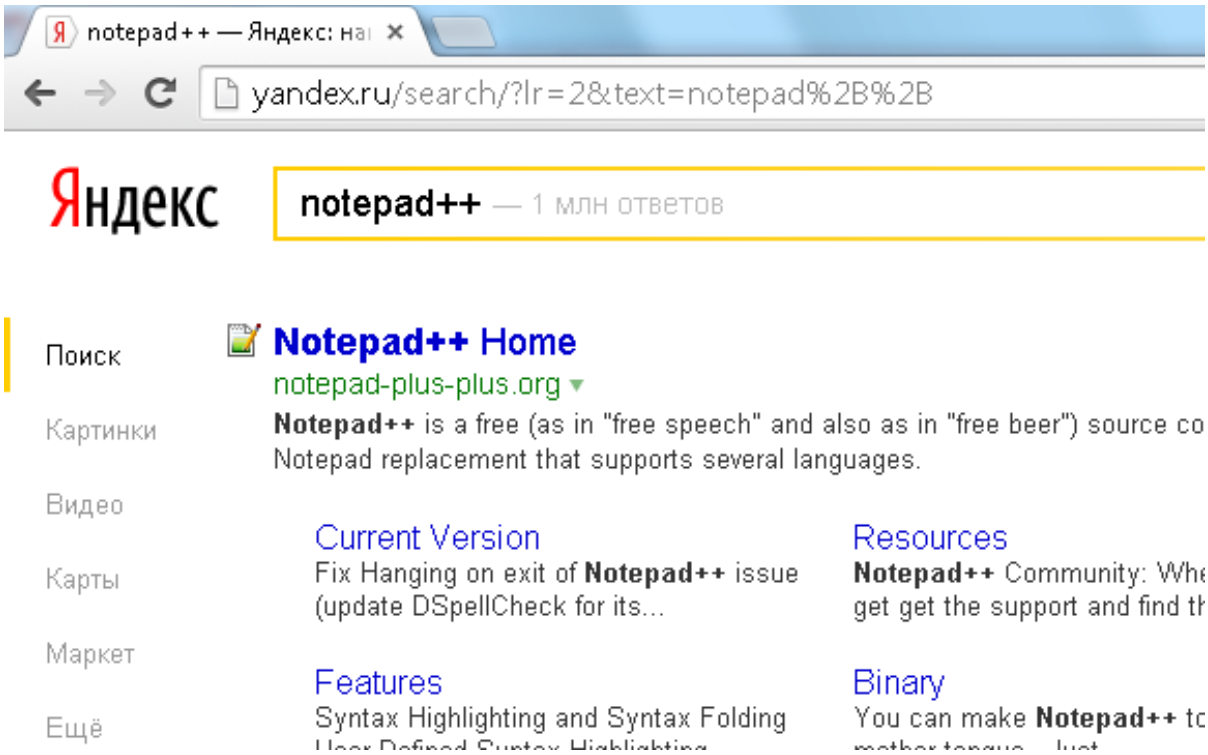


Рис. 2. Окно Notepad++



Для того, чтобы скачать **Notepad++** воспользуйтесь поисковой системой **Яндекс** (как показано на Рис. 3) или зайдите на сайт разработчиков программы (*http://notepad-plus-plus.org/download/v6.6.8.html*).

Рис. 3. Поиск программы Notepad++



Для того, чтобы подключить скрипты **Python** к **Robur**, необходимо запустить **Robur** и в окне настроек указать подключаемые модули и порядок их загрузки Это может быть необходимо, в случае, если один скрипт использует объекты другого. Также старайтесь, чтобы Ваши скрипты загружались после системных или других используемых Вами. Подробнее подключение скриптов к **Robur** рассмотрено в *примере 1*.

Пример **1.** Первый скрипт и его регистрация в **Robur**

В данном примере подробно рассматривается состав простейшего скрипта на Python, содержащего макрос, выдающий окно с сообщением "Привет, **Robur**", а затем, последовательность шагов по подключению этого скрипта к **Robur** и вызов макроса из командной строки **Robur**.

Первая часть Примера посвящена разработке скрипта.

Вторая часть Примера описывает Создание собственных пункта меню и кнопки на панели инструментов в **Robur**, вызывающей написанный нами макрос.

Постарайтесь детально разобраться в описанных здесь шагах, так как аналогичным образом подключаются все остальные скрипты, а вызов макросов и создание кнопок (других элементов управления) пригодиться Вам как для настройки среды **Robur** (персонализации для повышения удобства работы), так и при разработки программных модулей под платформу **Robur**.

Разработка макроса "Привет**, Robur!"**

Откройте файл *sample01.py* в блокноте или **Notepad++**. Детально рассмотрим его содержимое.

- Первая строка:

```
# -*- coding: 1251 -*-
```

задает кодировку для файла скрипта. В наших примерах мы использовали кодировку Windows 1251, поддерживающую кириллицу (хотя Вам мы, все-таки, рекомендуем использовать кодировку utf8).

- Следующие две строки:

```
import System
from robur import *
```

подключают внешние библиотеки Microsoft .Net и их пространство имен System к скрипту, и все элементы из пространства имен библиотеки Топоматик Robur "robur".

Если Вы не знакомы с понятием "пространство имен", то для дальнейшей работы Вам достаточно знать, что пространство имен - это список имен классов, объектов или переменных и прочего, предоставляющихся Вам из подключаемых библиотек. Поэтому, подключив пространство имен можно пользоваться всем, что оно содержит.

- После того, как нужные пространства имен подключены, Вам необходимо объявить класс Вашего модуля, подключаемого к **Robur**.

```
class Sample01Module(PyModule):
```

Строго говоря, к платформе **Robur** подключается не файл скрипта, а модуль, содержащийся в нем. Модуль это некая логическая единица, подключаемая к **Robur**. Модуль имеет свое уникальное имя (в нашем случае "**Robur Sample 01 Module**"). Имя будет задано в последней строке скрипта.

Теоретически, файл скрипта может содержать и несколько модулей, в наших примерах по одному модулю в файле объявлено просто из удобства. Мы рекомендуем Вам поступать также и при разработке собственных скриптов.

Если Вы хорошо знакомы с объектно-ориентированным программированием, то приведенная выше строка объявляет класс **Sample01Module** - наследник от **PyModule** - базовы класс модуля, разработанный фирмой Топоматик.

- Затем, с помощью директивы **def** объявляется процедура Вашего макроса внутри модуля (с именем act_show_hello_robur и тремя параметрами):

```
def act_show_hello_robur(self, sender, e):
```

а также задаются атрибуты макроса:

```
@action('Вывод Привет, Robur!', alias='hello')
```

Первым параметром **@action** задается спарвочное имя процедуры, а параметр alias фактически и назначает имя макроса для вызова процедуры, т.е. для того, чтобы вызвать макрос в командной строке **Robur** нужно будет набрать "**hello**".

- Следующая строка - это само тело процедуры - вызов диалогового окна Windows (MessageBox) со строкой "Привет, Robur!"

```
System.Windows.Forms.MessageBox.Show("Привет, Robur!",
 "Sample01", System.Windows.Forms.MessageBoxButtons.OK,
 System.Windows.Forms.MessageBoxIcon.Exclamation)
```

Так как мы не подключали всё пространство имен System.Windows.Forms, то перед каждым именем класса или параметра указывается полное имя "**System.Windows.Forms. ...**". Параметры метода MessageBox.Show можно посмотреть в справке Microsoft MSDN, где описана системное пространство имен .NET Framework System.

Обратите внимание на синтаксис языка Python, в конце строк с его операторами или другими объявлениями нет символов ";", обычно присутствующих в других языках программирования.

Также в Python нет операторных скобок (как например "{ }" в языках C и C++, "begin end" в Pascal и Delphi или "End If, End Sub и др." в Basic.

В Python блоки кода выделяются отступом от кода слева. То есть, если блок кода, например, после оператора if должен при выполнения условия выполняться весь, то все строки должны иметь отступ влево относительно самого if, а следующие строки, не относящтеся к условию должны снова иметь тот же отступ, что и if.

Для наглядности рассмотрим следующий код:

```
1  if (a == 1)
2      print '1'
3      print '2'
4      print '3'
5  print '4'
```

Здесь строки, выдающие текст 1, 2 и 3 в окно командной строки (консоль) сработают, если значение переменной a равно 1, т.е. Вы увидите 1 2 3 4. А строка, выдающая 4 выполнится всегда (т.е. если a не равно 1, то вы увидите только 4).

- И последняя строка кода:

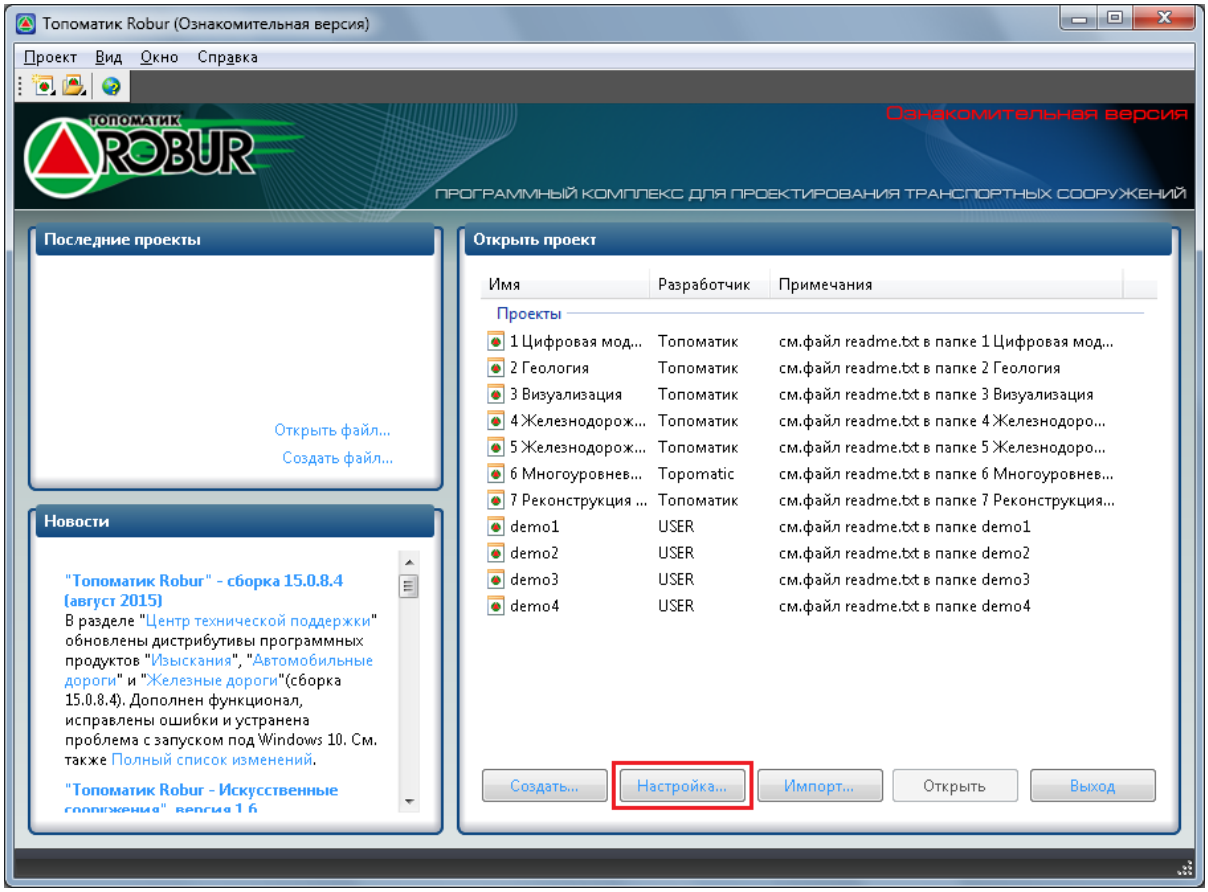
```
Sample01Module().register('Robur Sample 01 Module')
```

Создает сам объект модуля Sample01Module и регистрирует его в списке модулей палтформы **Robur** с именем **Robur Sample 01 Module**. Это имя будет отображаться в настройках в **Robur** и т.п.

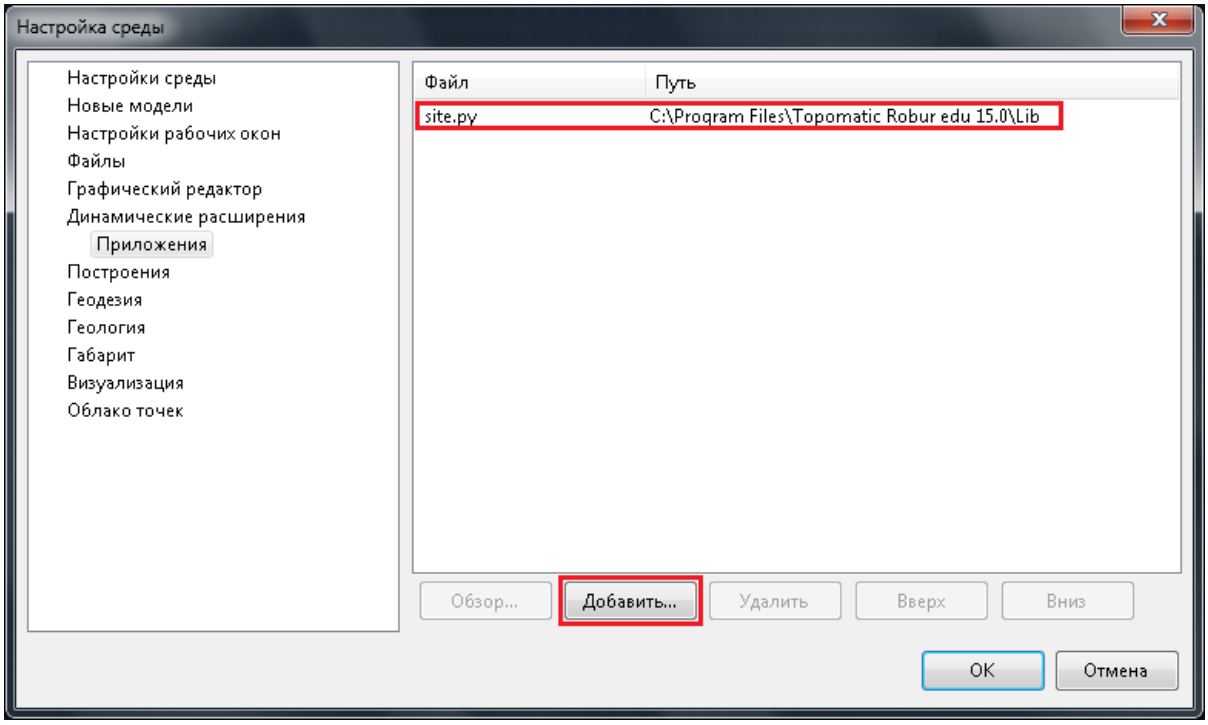
Несмотря, на то, что это - последняя строка, наверное в данном примере она - самая важная, так как фактически весь код до нее объявлял класс нашего модуля и его метод - макрос, а эта строка создает и регистрирует в **Robur** все написанное.

Регистрация скрипта в **Robur**

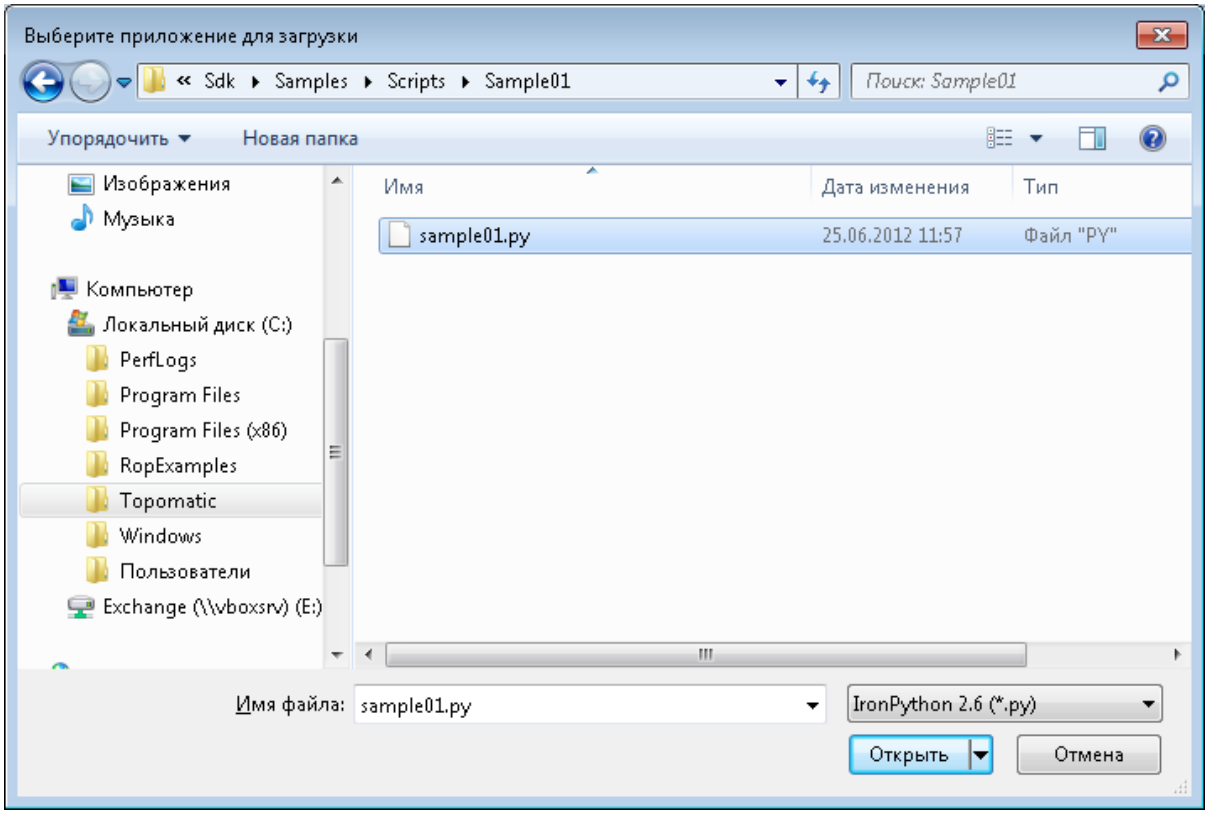
Для регистрации файлов со скриптами необходимо запустить **Robur**, и, не открывая проект, на стартовой странице (или в диалоге выбора проекта) нажать на кнопку "**Настройка...**".



В окне настроек среды выберите пункт "**Динамические расширения - Приложения**". И нажмите кнопку "**Добавить**".



В диалоге обзора необходимо найти и выбрать Ваш файл скрипта.

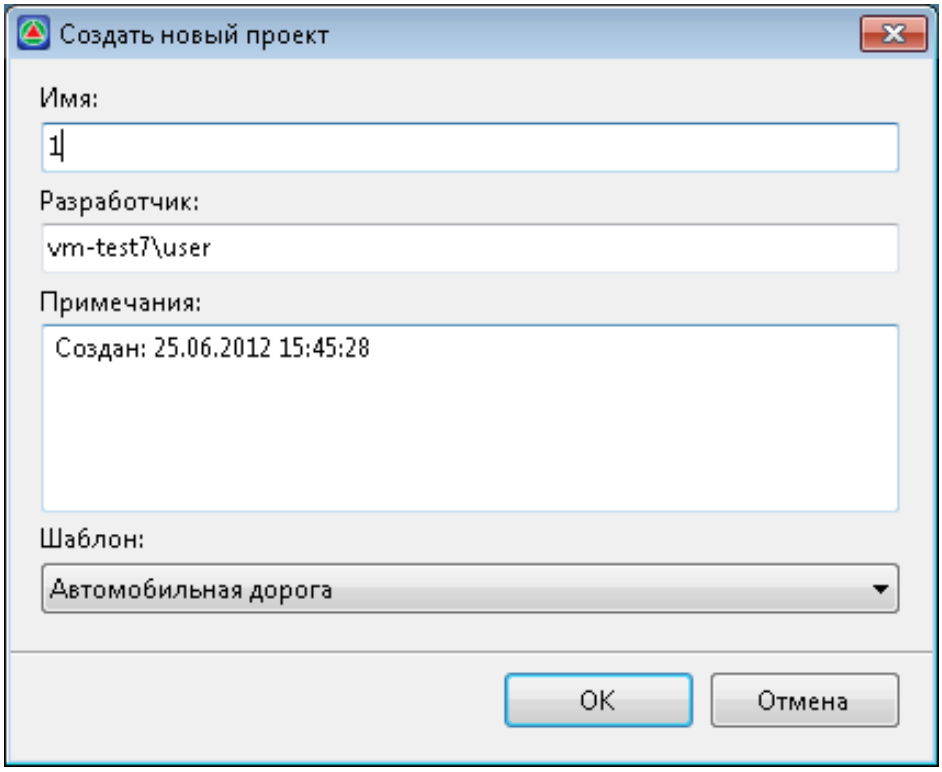


Будьте внимательны!

Для корректной работы скриптов необходимо, чтобы в настройках был также добавлен файл системного скрипта site.py (этот файл находится в подкаталоге Lib, расположенного в папке с установленным Robur).

В окне настроек этот скрипт должен быть расположен раньше пользовательских скриптов, чтобы необходимые для работы системные вызовы уже были выполнены на момент их запуска.

Затем закройте окно настроек, нажав кнопку "OK" и создайте новый проект.

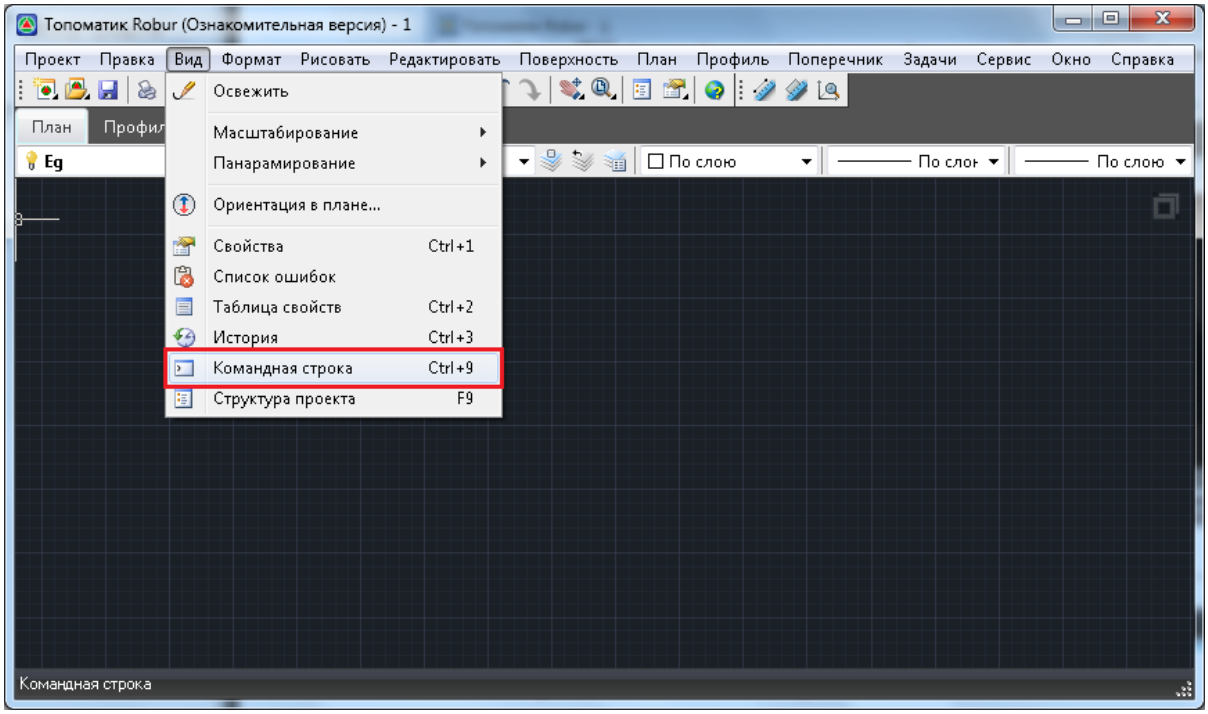


Будьте внимательны!

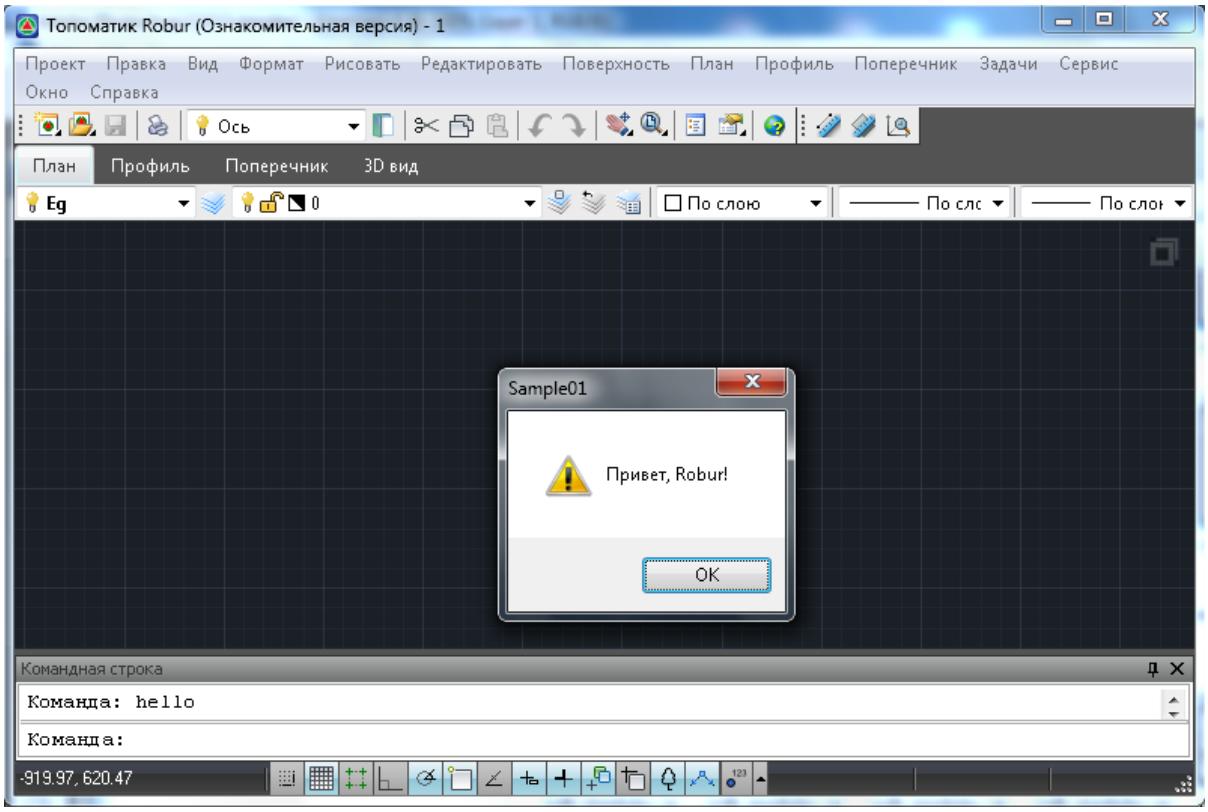
В общем случае (кроме системных команд, управляющих непосредственно средой **Robur**), макросы могут работать только когда в **Robur** открыт какой-либо проект, так как для своего выполнения им необходимы активные чертежи, модели и другие элементы проектов.

После открытия проекта вызовите командную строку и запустите в ней Ваш макрос, набрав "**hello**".

Вызов командной строки:



Вызов макроса hello в командной строке:



В результате Вы должны увидеть соответствующее диалоговое окно.

Добавление элементов управления для команд

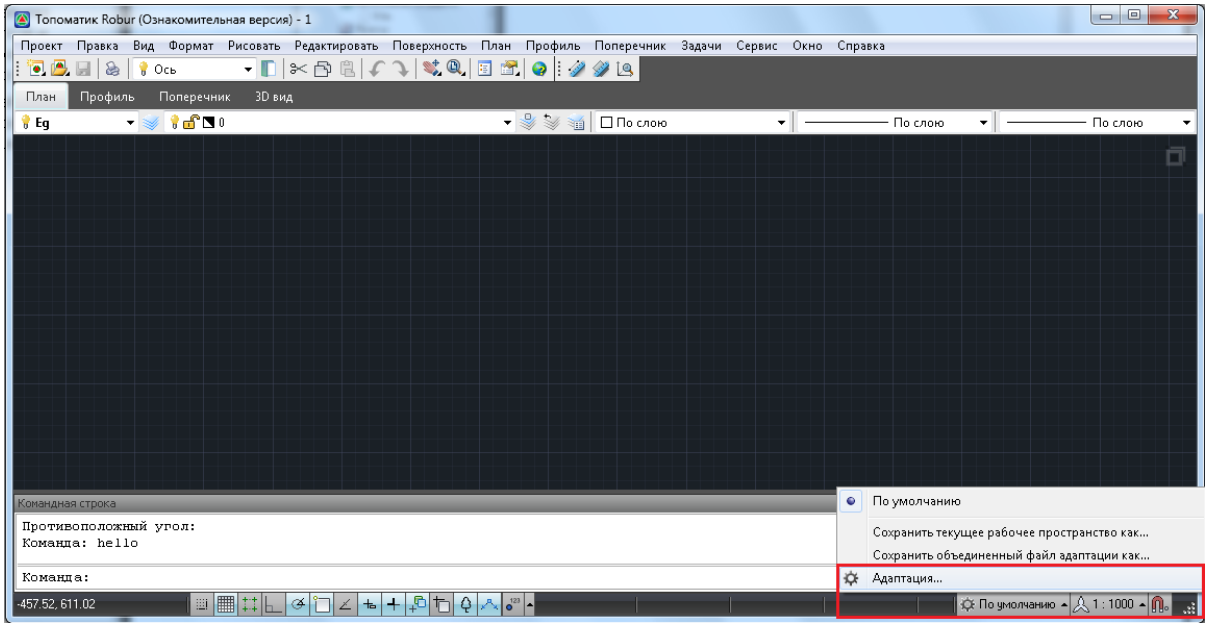
Как уже было описано выше для вызова макроса или или команды (**action**) программного модуля, подключенного к **Robur**, можно добавить пункты в меню (главное или контекстные) Robur, а также кнопки на панели инструментов.

В этом разделе мы рассмотрим как добавить пункт в меню и кнопку на панель инструментов **Robur**.

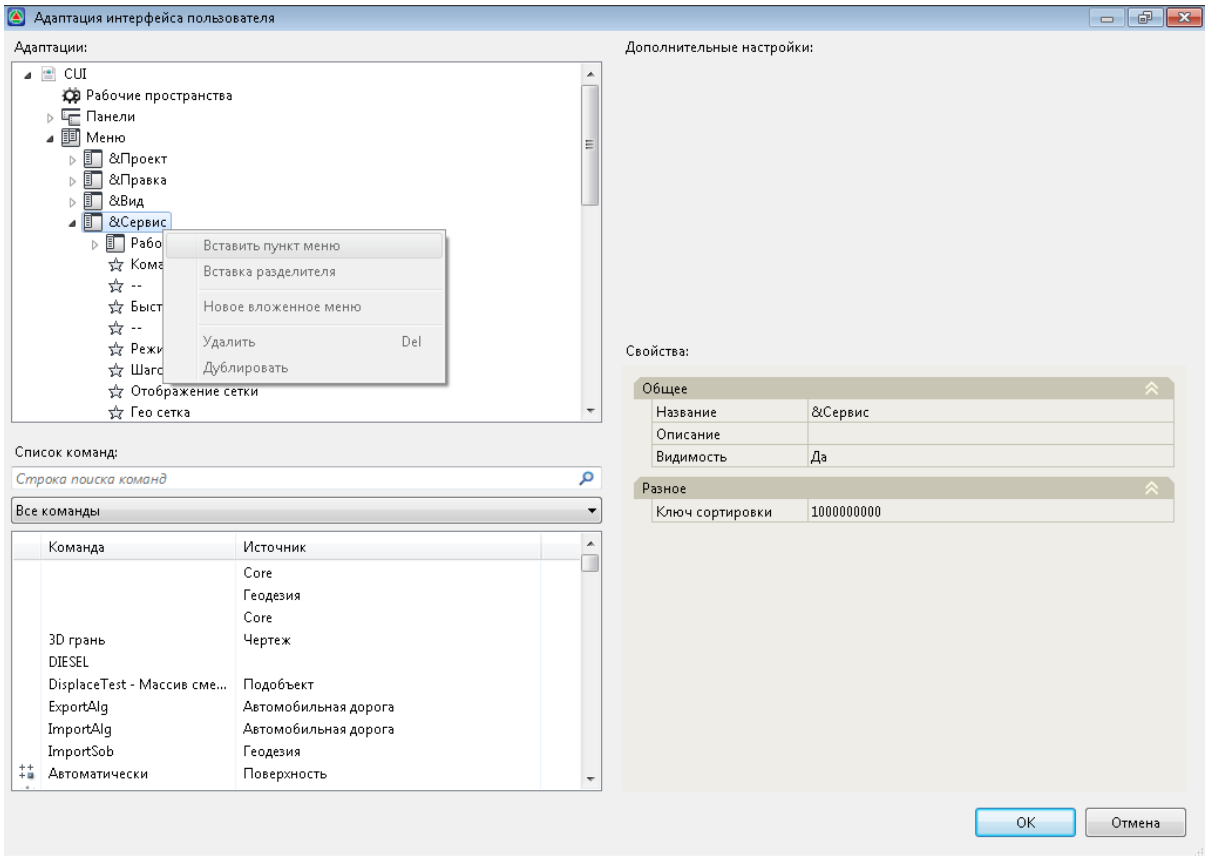
Для того, чтобы добавить собственные пункты меню или кнопки, необходимо выполнить следующие шаги:

- Запустить **Robur** и открыть любой проект
- Вызвать окно настроек персонализации (окно, в котором Вы можете настроить внешний вид среды **Robur** удобным Вам образом (скрыть или переместить кнопки и панели инструментов и пункты меню и т.п.)

  Для его вызова перейдите на окно "План" и в правом нижнем углу нажмите кнопку с названием активного профиля (на рисунке ниже - "**По умолчанию**") и выберите пункт меню "**Адаптация**".



- Затем в открывшемся окне в левом верхнем блоке выберите, в каком разделе главного меню или на какой панели инструментов Вы собираетесь создать пункты или кнопки и, щелкнув на нем правой кнопкой, в контекстном, меню выберите "**Вставить пункт меню**" (или кнопку и т.п.).

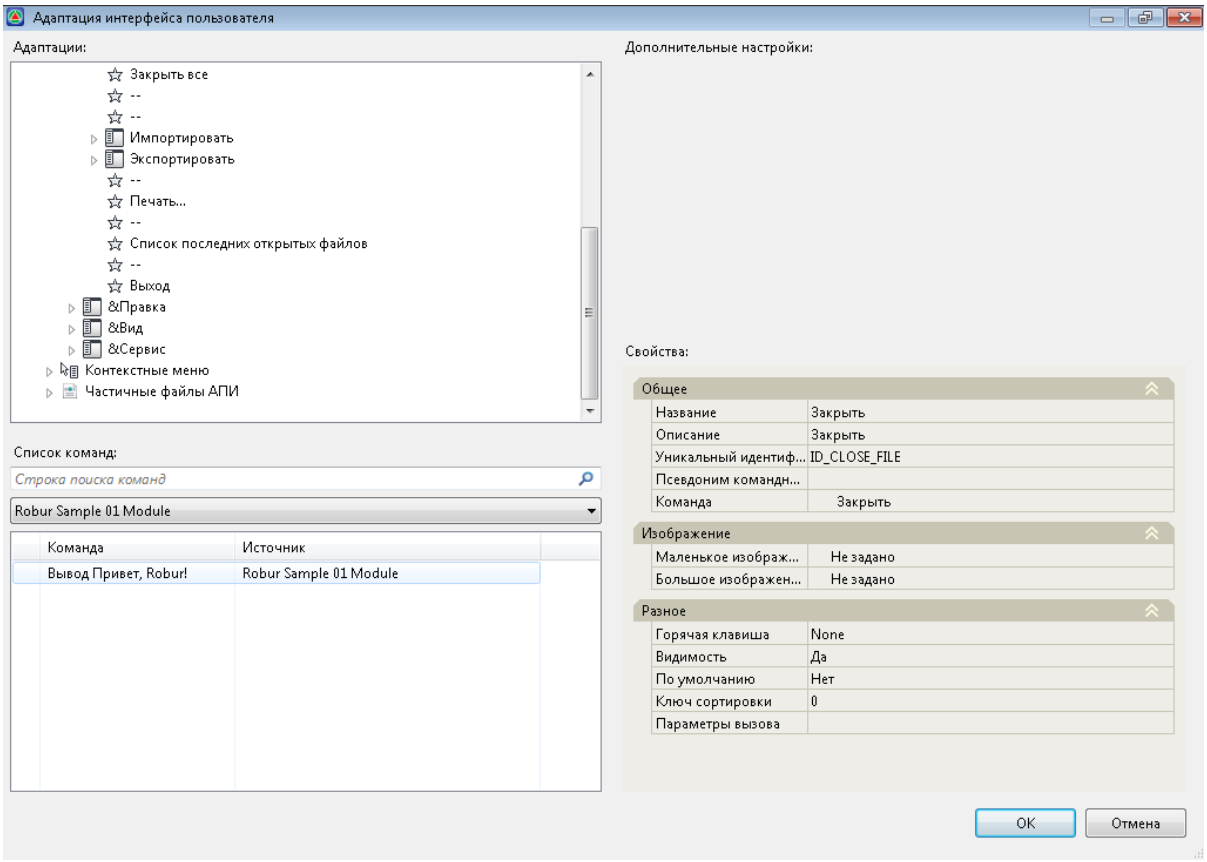


- Теперь, когда нужный элемент добавлен и выбран в левом верхнем блоке, в списке посередине левой части окна выберите тот модуль, команду или макрос из которого Вы хотите вызывать с помощью этого пункта меню или кнопки.

  В нашем случае мы выбираем **Robur Sample 01 Module** - модуль нашего первого примера (так как мы его и задали при регистрации в скрипте). Если по каким-то причинам его там нет, значит Вы допустили

ошибку и модуль не подключился к Robur. В таком случае, Вам необходимо еще раз внимательно прочитать *предыдущий раздел*.

Обратите внимание, что Вы можете также выбрать и любой другой модуль, включая компоненты самого **Robur**.



Как только Вы выбрали модуль в списке, поле под ним обновилось и стало содержать все команды из этого модуля. У нас в модуле первого примера был объявлен только один макрос "**Вывод Привет, Robur!**". Если ее там нет, значит Вы допустили ошибку пр создании *первого примера*.

Если все в порядке, перетащите мышью строчку из этого списка с командой на правое поля окна, и увидите, как оно обновится и станет содержать данные этой команды.

В принципе, в правом поле, Вы можете задать дополнительные параметры для элемента управления (название пункта меню, картинку для кнопки, подсказку и т.п.) но в нашем случае, мы просто нажмем OK и попытаемся найти и вызвать этот пункт меню (кнопку) в окне **Robur**. После перезагрузки **Robur** Ваши настройки также сохранятся.

Пример **2.** Макрос**,** вставляющий примитив в текущий чертеж проекта **Robur**

Второй пример, в отличие предшественника, уже содержит определенную функциональность и использует объекты платформы **Robur** и вносит изменения в текущий проект, открытый в **Robur**.

Пример состоит из двух модификаций скрипта: *sample02.py* и *sample02_a.py*. Первый запрашивает у пользователя координаты и радиус и вставляет в текущую модель (в ее чертеж) полилинию в форме пятиконечной звезды, вписанной в окружность заданного радиуса. Второй скрипт проще - он запрашивает координаты точки в чертеже и вставляет по ним текст "TEST". Вы можете построчно сравнить код обоих скриптов и увидеть, как используются функции чертежа **Robur** и некоторых других объектов платформы

(таких, как **CadView** - поле для редактирования модели в окне Плана, профиля и некоторых других окон **Robur**.

Рассмотрим код скрипта:

```
# -*- coding: 1251 -*-

import System
import Topomatic
import math
from robur import *

class Sample02Module(PyModule):

@action('Вставка примитива', alias='drawp')
def act_draw_p(self, sender, e):
```

Также как и в предыдущем примере, мы задаем кодировку самого файла Python, подключаем необходимые пространства имен библиотек и создаем модуль. Обратите внимание, что мы добавили пространство имен math, так как для расчета координат звезды нам понадобятся математические функции Python, кроме того изменилось название класса модуля на **Sample02Module** - так как пример второй.

Следующим шагом, также объявлены процедура и описание макроса, с новыми именами. Теперь, для вызова в командной строке нужно будет набрать **drawp**.

Далее мы получаем **CadView** активного окна в **Robur** в момент выполнения макроса. Если на активном окне не оказывается **CadView**, то макрос завершает свою работу, строки:

```
cadview = self.CadView
if (cadview == None):
    return
```

Следует подробно остановиться на **CadView**, редакторе видов моделей в **Robur**.

Как Вы знаете, **Robur** - это система, в основе которой лежит единая пространственная модель, а не набор каким-то образом связанных чертежей и дополнительных параметров.

Это дает колоссальные преимущества при проектировании объектов и в общем-то и является основной изюминкой **Robur**. Так как все виды (проекции) этой модели отображают одну и ту же модель, то независимо от того, в каком из видов Вы редактируете модель - меняются и все остальные. Например, Вы меняете отметки на профиле дороги, они меняются на соответствующих поперечниках. Меняете положение трассы в плане, меняются профили и поперечники, так как трасса уже проходит по другим отметкам земли и т. п.

CadView - это компонент визуального редактирования или просто отображения модели. С помощью него осуществляется редактирование плана, профиля, поперечников в соответствующих окнах, также некоторых других видов.

С помощью **CadView** и его дополнительного инструментария (курсоры, выборки, диалоговые средства, средства пересчета координат и др.) происходит отображение слоев модели, выбор объектов, задание координат и другие операции с моделью и ее составляющими.

В нашем примере мы используем **CadView** для того чтобы снять выделение со всех объектов, которые были выбраны на момент вызова макроса, а также используем курсор **CadView** для визуального задания координат и запроса радиуса.

Так как мы используем метод самой платформы, то естественно при выборе координат будут работать привязки, включенные в **Robur**, и все другие удобства, присутствующие при работе функций самого комплекса **Robur**. Это очень удобно, так как однотипное поведение одинаковых режимов работы будет понятно и ожидаемо пользователями Ваших разработок.

Следующим шагом мы получаем текущий чертеж, а точнее, сперва активный слой чертежей в **CadView**, а потом уже и сам чертеж:

```
drawing_layer = self.ActiveDrawingLayer
```

Здесь, пользователям **AutoCAD** надо иметь ввиду что под "активным слоем чертежей" имеется ввиду не слой чертежа, на котором располагаются примитивы (как слои в **AutoCAD**), а слой модели **Robur**, на котором располагается весь чертеж целиком, так как **Robur** в одном проекте может содержать несколько чертежей. Также на слоях других типов будут располагаться и другие элементы проекта: геодезические съемки, поверхности, подобъекты и другие.

Конечно, в чертежах **Robur** примитивы также располагаются на различных слоях, как и в **AutoCAD**, только эти слои становятся дочерними, по отношению к слою всего чертежа. Кстати, другие элементы проекта также имеют подслои, например, слои пикетажа, плана в подобъектах или точек и горизонталей в поверхности. И этими слоями можно централизованно управлять, в том числе и из скриптов и модулей, разрабатываемых Вами.

Наш скрипт, проверяет, удалось ли получить активный слой чертежа и сам чертеж, в противном случае, с помощью команды Print выдается сообщение в командную строку **Robur** и макрос завершает работу. Вполне возможно, что в данный момент в **Robur** активно окно, не имеющее **CadView** или в этой проекции нет чертежей, поэтому такая ситуация вполне может произойти и выводить аварийных сообщений не стоит, также не стоит и продолжать выполнение, чтобы не произошло исключений (программных ошибок при обращении к несуществующим объектам).

Следующими строками идет подготовка переменных для принятия координат от **CadView** (такой конструкцией в Python подготавливается переменная, передаваемая в функцию .NET по ссылке):

```
rp = clr.Reference[Topomatic.Cad.Foundation.Vector3D]()
```

Если Вы обратитесь к документации функции GetPoint курсора **CadView**, то увидите что параметр точки с типом Vector3D передается по ссылке. Вам часто прийдется использовать такой прием и объявление ссылок, так как очень много функций платформы **Robur** и системных функций .NET Framework используют такой способ возврата значений.

акже объявляется и ссылка на переменную типа double - для возврата значения радиуса:

```
rr = clr.Reference[float]()
```

Далее идет строка сброса выделения в **CadView**:

```
cadview.SelectionSet.Clear()
```

Снимать выделение рекомендуется перед каждой командой, требующей от пользователя выбора чего либо на **CadView**, чтобы не вводить его в заблуждение, так как некоторые команды могут работать с уже выбранными объектами на **CadView** или с некоторыми из них (например, удовлетворяющими каким-либо условиям).

Затем, вызывается команда, запрашивающая у пользователя координаты на экране:

```
while (Topomatic.Cad.View.Hints.CadCursors.GetPoint(cadview, rp,
 'Укажите точку')):
```

Обратите внимание, что функция запроса взята в качестве условия срабатывания цикла while, что означает, что наша команда по вставке звезды будет повторяться до тех пор, пока пользователь не откажется от задания координат (нажмет правую кнопку мыши или Escape на клавиатуре).

Далее строки, запрашивающие радиус (по умолчанию радиус задается равным 50м):

```
rr.Value = 50.0
from Topomatic.Cad.View import GetPointResult
```

```
gpr = Topomatic.Cad.View.Hints.CadCursors.GetDoubleWithDefault(cadview, rr,
 'Укажите радиус')
if (gpr != GetPointResult.Accept):
    return
```

Обратите внимание, что при обращении к инструменту **CadView** классу CadCursors мы указываем полное имя, в то время как тип GetPointResult объявлен один раз заранее, с помощью строки:

```
from Topomatic.Cad.View import GetPointResult
```

а потом используется уже без полного имени.

Оставшееся тело нашего скрипта - это вставка в чертеж полилинии в виде звезды, вписанной в указанный радиус. Собственно весь большой блок и посвящен созданию объектов - узлов этой полилинии (вершин звезды), расчету их координат и добавление в список узлов полилинии. Для того, чтобы не пугать Вас большими блоками кода, мы также составили второй вариант примера, код которого здесь. Он полностью аналогичен рассматриваемому, но вместо звезды вставляется текст и никаких узлов и расчетов в нем нет. Сравнив коды обоих примеров Вы четче поймете, какая строка за что отвечает.

Далее в скрипте задается положение первой вершины - ее угол относительно центра звезды (точки вставки, заданной пользователем), а по углу - смещения координат:

```
angle = 0.0
dx = math.cos(angle) * rr.Value
dy = math.sin(angle) * rr.Value
```

И сами координаты точки:

```
pos = Topomatic.Cad.Foundation.Vector2D(rp.Value.X + dx, rp.Value.Y + dy)
```

Создается объект полилинии:

```
polyline = Topomatic.Dwg.Entities.DwgPolyline()
```

И в нее добавляется первый узел:

```
polyline.Add(Topomatic.Cad.Foundation.BugleVector2D(pos, 0))
```

Узел полилинии в **Robur** - это не просто двухмерная или трехмерная точка (Vector2D или Vector3D), а специальный объект типа BugleVector2D, который и создается на базе точки pos.

Затем запускается цикл, в котором рассчитываются и добавляются остальные 5 вершин (последняя вершина будет совпадать с первой, так как звезда - замкнутая фигура).

```
for i in xrange(1, 6):
    angle += math.pi / 1.25
    dx = math.cos(angle) * rr.Value
    dy = math.sin(angle) * rr.Value
    pos = Topomatic.Cad.Foundation.Vector2D(rp.Value.X + dx, rp.Value.Y +
 dy)
    polyline.Add(Topomatic.Cad.Foundation.BugleVector2D(pos, 0))
```

И последним шагом полилиния добавляется в чертеж:

```
drawing.ActiveSpace.Add(polyline)
```

Также, как и в предыдущих примерах в конце файла скрипта мы создаем модуль и регистрируем его в **Robur**:

```
Sample02Module().register('Robur Sample 02 Module')
```

Пример **3.** Выбор примитивов текущего чертежа и чтение их свойств

В принципе, в предыдущих примерах, Вы подучили все необходимые сведения для разработки полноценных скриптов и макрокоманд для **Robur**. Дальнейшие примеры будут только приемам работы с определенными и наиболее часто используемыми компонентами проекта (такими как чертежи, поверхности, подобъекты и др) или самой платформы.

Исчерпывающую информацию о свойствах и функциях всех этих объектов, а также о составе интересующих Вас подсистем Вы сможете найти в документации для разработчика платформы **Robur** (**Robur Platform API**). И все эти знания могут Вами использоваться независимо от того, будете ли Вы разрабатывать скрипты или полноценные модули на Вашем любимом языке программирования для .NET.

Этот пример продемонстрирует Вам как выбирать существующие примитивы чертежей и получать их свойства. Макрос этого примера попросит пользователя выбрать примитив типа "отрезок" (**DwgLine**), получит свойство этот отрезка - длину и вставит в активный чертеж примитив "текст" (**DwgText**) со значением длины примерно в центре отрезка (как подпись).

Рассмотрим скрипт (sample03.py):

Импорт пространств имен, объявление модуля и процедуры макроса аналогично предыдущему примеру:

```
# -*- coding: 1251 -*-

import System
import Topomatic
from robur import *

class Sample03Module(PyModule):

    @action('Вставка подписи к отрезку', alias='draws')
    def act_draw_s(self, sender, e):
```

Также как и в предыдущих примерах получаем **CadView** и слой активного чертежа для вставки подписи:

```
cadview = self.CadView
if (cadview == None):
    return
drawing_layer = self.ActiveDrawingLayer
if (drawing_layer == None):
    print 'Активный чертеж не найден'
    return
drawing = drawing_layer.Drawing
```

Сбрасываем выбранные объекты в **CadView**. Кстати, можно и не сбрасывать выделенные объекты, а проверить, нет ли среди них отрезков, а в случае если они найдены, то не запрашивать у пользователя выбрать отрезки, а подписать их длину и выйти из макроса. Тогда, вместо **SelectionSet.Clear** и цикла с выбором примитивов, надо добавить код:

```
cadview.SelectionSet.FilterSelected(filter)
if (cadview.SelectionSet.Count == 0):
    print 'Не выбрано ни одного отрезка'
    return
for line in cadview.SelectionSet:
```

Вариант примера с таким поведением рассмотрен в скрипте *sample03_a.py*.

Обратите внимание, что в случае, если среди выбранных объектов нет отрезков, скрипт выдает соответствующее сообщение в окно командной строки и завершает работу. Также заметьте, что фильтр для выбора отрезков (`filter = lambda obj: isinstance(obj,`

`Topomatic.Dwg.Entities.DwgLine)`) объявлен раньше, чем в основном примере и используется, как аргумент, фильтрующий список выбранных элементов, оставляя в нем только отрезки.

Чаще всего, оба метода комбинируются и, если,

А в нашем примере, далее следует код, запрашивающий у пользователя отрезок. Здесь также используется **SelectionSet**:

```
cadview.SelectionSet.Clear()
while (True):
    filter = lambda obj: isinstance(obj, Topomatic.Dwg.Entities.DwgLine)
    #line = cadview.SelectionSet.PickOneObjectAtScreen(filter,
 'Выберите отрезок')
    line = drawing_layer.SelectionSet.PickOneObjectAtScreen(filter,
 'Выберите отрезок')
    if (line == None):
        return
```

Обратите внимание, что код содержит две похожих инструкции выбора отрезка, но одна закоментирована и не выполняется.

Отличие этих строк в том, что одна использует для выбора **SelectinSet** всего **CadView**, то есть пользователь может выбрать отрезок, расположенный не только в активном чертеже, а в любом из чертежей, которые отображаются в данный момент на **CadView**, тогда как наша подпись будет вставлена в активный чертеж, а не обязательно в тот, где находился отрезок.

Однако объект **SelectionSet** есть и у каждого слоя **Robur**, в том числе и у слоя активного чертежа. Вторая строка, как раз, запросит отрезок именно у слоя с активным чертежом, а другие отрезки, даже, если они отображаются будут проигнорированы.

Вы можете поменять строки, закоментировав ненужную и посмотреть на поведение скрипта.

Также обратите внимание, как задается процедура фильтра (**lambda**), проверяющая, что объект является наследником от **DwgLine**.

Следующие строки посвящены получению нужных свойств выбранного отрезка и добавление текста в чертеж:

```
text = Topomatic.Dwg.Entities.DwgText()
from System import Math
text.Content = Math.Round(line.Length, 2).ToString()
text.Rotation = line.Rotation
text.Position = Topomatic.Cad.Foundation.Vector3D(line.StartPoint.X
 + (line.EndPoint.X - line.StartPoint.X) / 2, line.StartPoint.Y
 + (line.EndPoint.Y - line.StartPoint.Y) / 2, line.StartPoint.Z +
 (line.EndPoint.Z - line.StartPoint.Z) / 2)
drawing.ActiveSpace.Add(text)
```

Также как и в предыдущем примере работа заключена в цикл **while**, пока пользователь не отменит выбор отрезков.

Пример **4.** Управлением механизма отмены команд **Robur (Undo)** и истории команд

Следующий наш пример посвящен такому механизму **Robur**, как **Undo** или **Отмена**. Как известно все современные программы хранят журнал изменения состояния объектов и по требованию пользователя могут их отменить.

Также пользователь видит список последних изменений, где каждое изменение имеет какое - то понятное название (имя команды и т.п.).

Если Вы отобразите этот список выполненных команд (историю) в Robur,

Запустите предыдущий пример и подпишите за одну команду несколько отрезков (так как пример содержит цикл **while**, пока пользователь не прекратит выбирать отрезки), то Вы увидите, что история отобразит каждую подпись как отдельную команду. И каждую подпись можно по отдельности отменить.

В нашем случае ситуация не драматична, так как в примере мы сначала меняем все свойства примитива

```
text.Content = Math.Round(line.Lenght, 2).ToString()
text.Rotation = line.Rotation
text.Position = Topomatic.Cad.Foundation.Vector3D(line.StartPoint.X
 + (line.EndPoint.X - line.StartPoint.X) / 2, line.StartPoint.Y
 + (line.EndPoint.Y - line.StartPoint.Y) / 2, line.StartPoint.Z +
 (line.EndPoint.Z - line.StartPoint.Z) / 2)
```

а уже потом добавляем его в модель

```
drawing.ActiveSpace.Add(text)
```

Однако, если поменять в одной команде несколько свойств у существующего примитива, уже имеющегося в модели, то каждое изменение будет зафиксировано в истории как отдельный шаг. Это может быть недлпустимо, так как отмена только части свойств из одной команды может нарушить целостность чертежа или всей модели.

Для таких целей в платформе **Robur** предусмотрен механизм транзакций. Транзакция представляет собой логическую группировку нескольких изменений в виде единой записи в истории, откат которой может быть произведен только полностью.

В ходе выполнения кода Вы можете сами определить, какая часть операций входит в транзакцию, а также задать ей имя (для окна истории). Кроме того, если в момент выполнения команд транзакции произошла какая-то ошибка, то можно не применять транзакцию, и все уже начатые изменения применены не будут.

Перейдем к примеру (*sample04.py*). В качестве основы был взят предыдущий пример, *sample03.py* и дополнен необходимым кодом.

Перед началом цикла **while** запускаем транзакцию (`drawing.BeginUpdate('Добавление подписей к отрезкам')`), а после выхода, применяем ее (`drawing.EndUpdate()`).

Теперь, если мы запустим команду и подпишем несколько примитивов, то в окне истории все равно отобразится только одна операция.

Если бы нам было необходимо отменить транзакцию, то перед строкой `drawing.EndUpdate()` нужно вызвать

```
drawing.TransactionManager.CurrentTransaction.Undo()
```

Строго, говоря не у каждого объекта есть **TransactionManager**, позволяющий управлять транзакциями, а также не каждый объект разрешает отмену, поэтому в Ваших разработках, если операции отмены являются критичными, то нужно сначала проверять назначен ли **TransactionManager**, а уже потом начинать работать с транзакциями (вызывать **BeginUpdate**).

Также надо иметь ввиду, что если транзакция началась корректно (корректно отработал **BeginUpdate**), что у **TransactionManager** будет установлено свойство **CurrentTransaction**.

Также у **CurrentTransaction** есть булево свойство **CanUndo**, показывающее возможна ли отмена данной транзакции. Некоторые редкие операции не допускают отмены.

Благодаря этому примеру Вы научились применять механизм транзакций в своих командах и делить выполняемые изменения на логические части.

Пример **5.** Работа с поверхностями **Robur (**ЦММ**)**

В предыдущих примерах мы обращались к активному чертежу проекта: вносили новые примитивы или выбирали отрезки и читали их параметры. Следующие два примера посвящены двум другим типам элементов проекта - поверхностям (**ЦММ**) и подобъектам (изыскательским трассам, автомобильным и железным дорогам), конечно, каждому типу можно посвятить очень много примеров, так как функционал этих подсистем очень богатый. Однако, все объекты и функции подобъектов и поверхностей подробно описаны в документации **Robur API**, а наша цель лишь рассмотреть способы их использования из скриптов **Python**.

Первый пример посвящен поверхности. В нем мы разместим две команды (макроса).

Первый запрашивает у пользователя несколько координат и вставляет в них точки поверхности, а потом спрашивает, желает ли пользователь создать структурную линию проходящую через эти точки. При положительном ответе мы создаем структурную линию.

Точки и структурные линии - это основные составляющие поверхности, поэтому научившись оперировать с ними, Вы сможете решать большинство собственных задач. Второй макрос скрипта, запрашивает у пользователя структурные линии и поднимает их отметки на 20 метров (отметка каждой точки структурной линии поднимается на 20 м.).

Рассмотрим код скрипта:

Заголовок модуля, объявление используемых пространств имен и функций макросов рассмотрены в *предыдущих примерах* и аналогичны им.

Перейдем сразу к получению активной поверхности:

```
surface = self.ActiveSurface
if (surface == None):
    print 'Активная ЦММ не найдена'
    return
```

Заметьте, что в этом примере мы также используем транзакции, чтобы все операции внутри нашего макроса были единой командой в окне истории и отменялись как одно целое. Подробнее транзакции были рассмотрены в *предыдущем примере*.

```
surface.BeginUpdate('Вставка точек и структурных линий')
```

Затем мы создаем структурную линию, несмотря на то, что структурную линию мы будем добавлять в модель только по согласию пользователя, мы создаем ее заранее и добавляем в нее точки, как в контейнер.

```
line = Topomatic.Sfc.StructureLine()
```

Затем, также как во втором примере, запрашиваем координаты для точек. При вводе координат учитывайте, что структурная линия будет проведена через эти точки в порядке их ввода.

```
while (Topomatic.Cad.View.Hints.CadCursors.GetPoint(cadview, rp,
 'Укажите точку')):
```

Следующие строки создают и добавляют точки в поверхность:

```
r = surface.GetElevation(rp.Valuse.Pos)
if (r == None):
    r = 100.0
rp.Value.Z = r
pt = Topomatic.Sfc.SurfacePoint(rp.Value)
index = Topomatic.Sfc.PointEditor(surface).Add(pt)
```

И сразу же сохраняют номера (индексы) точек в качестве узлов в структурную линию:

```
line.Add(index)
```

Отметка каждой рассчитывается по **ЦММ** с помощью функции поверхности **GetElevation**, если возможно (если в указанных пользователем координатах есть поверхность), в противном случае ставится 100м.

```
r = surface.GetElevation(rp.Valuse.Pos)
if (r == None):
    r = 100.0
rp.Value.Z = r
```

Далее запрашивем, нужно ли создавать структурную линию по введенным точкам:

```
ref = clr.Reference[bool]()
if (not Topomatic.Cad.View.Hints.CadCursors.GetBoolean(cadview, ref, None,
 'Создать структурную линию?:')):
    surface.EndUpdate()
    return
if (not ref.Value):
    surface.EndUpdate()
    return
```

Если пользователь отменяет выбор или отвечает отрицательно, то завершаем транзакцию и выходим. Также, в таких случаях, чтобы трижды не писать **EndUpdate()**, лучше использовать блок **try .. finally ..** . Если Вы имеете опыт программирования, то данный прием должен быть Вам знаком.

Если пользователь согласен, то добавляем структурную линию в модель, ну и конечно, **EndUpdate()**:

```
surface.StructureLines.Add(line)
surface.EndUpdate()
```

Первый макрос завершен. Если выполнить его, то мы получим описанный выше результат.

Второй макрос запрашивает у пользователя выбрать структурную линию.

Также как и первый он получает активную **ЦММ** проекта, а затем объявляется функция-предикат, которая будет передана в качестве фильтра в запрос структурной линии:

```
surface = self.ActiveSurface
if (surface == None):
    print 'Активная ЦММ не найдена'
    return
cadview.SelectionSet.Clear()
surface.BeginUpdate('Поднять структурную линию')
filter = lambda obj: isinstance(obj, Topomatic.Sfc.StructureLine)
```

Изначально у **SelectionSet** есть метод выбора объекта, который позволяет выбирать любой объект на **CadView**, но в качестве параметра ему можно передать функцию-предикат, уточняющую выбор, необходимый Вам. В нашем случае эта функция очень простая, она просто разрешает выбор, если объект является структурной линией. Однако в своих разработках Вы можете проверять и очень сложные условия, зависящие от многих факторов. Главное описать в предикате подходит ли передаваемый ему объект и вернуть соответствующий результат.

После того как предикат объявлен далее вызывается сам запрос выбора структурной линии:

```
line = drawing_layer.SelectionSet.PickOneObjectAtScreen(filter,
 'Выберите структурную линию')
```

Если структурная линия выбрана, то проходим по всем ее узлам и увеличиваем их отметку.

```
for i in xrange(0, line.Count):
    node = line[i]
    newNode = Topomatic.Sfc.StructureLineNode(node.Index, node.Elevation +
 20)
    line[i] = newNode
```

После подключения скрипта к **Robur** попробуйте вызвать обе команды, чтобы посмотреть результат.

В каталоге наших примеров имеется также скрипт Yamal05. В этом примере также реализована функция, поднимающая отметку структурной линии, однако, в отличие от описанного примера оно происходит за счет редактирования отметок самих точек поверхности, лежащих в узлах структурной линии. Рекомендуем Вам ознакомиться с данным примером, чтобы понимать, как редактировать свойства точек поверхности.

Пример **6.** Работа с подобъектами **Robur**

Этот пример рассматривает работу с подобъектами, в нем мы определим пикетажное положение указанной пользователем точки относительно текущего подобъекта и вставим в текущий чертеж соответствующую подпись.

Сначала, как в предыдущих примерах получаем текущий чертеж:

```
cadview = self.CadView
if (cadview == None):
    return
drawing_layer = self.ActiveDrawingLayer
if (drawing_layer == None):
    print 'Активный чертеж не найден'
    return
drawing = drawing_layer.Drawing
```

Затем, получаем текущий подобъект. Обратите внимание, что получение активного подобъекта отличается от способа получения активных повержености или чертежа, так как в **Robur** нет функции **get_active_alignment()**. Возможно, что такая функция появится в дальнейшем, однако логика ее реализации сейчас имеет несколько противоречий, так как в проекте **Robur** могу присутствовать модели, содержащие в себе несколько подобъектов. Кроме того, надо иметь в виду, что в некоторых сборка **Robur** (изыскательские версии), функционал, связанный с работой подобъектов может быть значительно ограничен.

Для получения активного подобъекта в текущей версии **Robur** необходимо использовать класс **ModelWatcher**, а точнее, его статическое свойство Current, содержащее Singleton-объект типа ModelWatcher. Такой объект создается при открытиии проекта, содержащего подобъекты, а его свойство **ActiveModel** при выделении в структуре проекта подобъекта, содержит ссылку на его модель.

Обратите внимание, что для работы с **ModelWatcher** в начале скрипта добавлены необходимые объявления и подключения соответствующей сборки **Robur**:

```
clr.AddReferenceByPartialName("Topomatic.Alg.Runtime")
...
from Topomatic.Alg.Runtime.ServiceClasses import ModelWatcher
```

Сам код получения активного подобъекта выглядит так:

```
model = ModelWatcher.Current.ActiveModel
if (model == None):
    print 'Активный подобъект не найден'
    return
alignment = model.Alignment
if (alignment == None):
    print 'Подобъект не выбран'
```

```
    return
```

Подобъект - одна из самых основных частей проекта **Robur**. Именно подобъект реализует, как общий функционал пространственной кривой - трассы: план трассы, профили (черные и проектные), струны смещений профилей, пикетаж, и поперечники... Также подобъект реализует и частные задачи автомобильных и железных дорог или геодезических трасс (расчеты верха земполотна автодорог, конструирование поперечных профилей железных дорог и т.п.).

Если Вы обладаете навыками программирования для **Microsoft .NET**, то настоятельно рекомендуем Вам ознакомиться с составом сборок **Topomatic.Alg.dll**, начиная с класса **Alignment** и его функций и свойств.

Специфические функции железнодорожных подобъектов находятся в сборке **Topomatic.Alg.Rail.dll**, а автодорожных - в **Topomatic.Alg.Road.dll**.

Также основные функции подобъектов и их параметры подробно рассмотрены в руководстве **API** платформы **Топоматик АР3**.

Затем, как в примере 2, в цикле запрашиваем у пользователя координаты точек:

```
while (Topomatic.Cad.View.Hints.CadCursors.GetPoint(cadview, rp,
 'Укажите точку')):
```

Далее мы обращаемся к плану подобъекта и вычисляем смещение в метрах относительно начала трассы для указанной точки с помошью функции **PosToStaOffs** (эта функция также определяет смещение по касательной в сторону от подобъекта).

PosToStaOffset - это основная функция при работе с планом линейного объекта, так как 90% задач тем или иным способом сводятся к расчету координат точки по смещению от началу трассы (sta) и смещению от оси (offset) и наоборот. Обратная функция называется **StaOffsetToPos**.

Блок кода, использующий функцию и добавляющий примитив (текст **DwgText**) с подписью пикета:

```
if (alignment.Plan.CompoundLine.PosToStaOffset(rp.Value.Pos, sta, offs)):
    text = Topomatic.Dwg.Entities.DwgText()
    text.Content = alignment.Stationing.StationToString(sta.Value)
    text.Position = rp.Value
    drawing.ActiveSpace.Add(text)
```

Обратите внимание!

Для получения имени пикета мы используем свойство подобъекта **Stationing** - реализующее таблицу пикетажа и все его функции. Для получения текста с именем пикета из смещения от начала трассы в метрах мы вызываем его функцию **StationToString.**

Для получения сведений об орстальных свойствах пожалуйста обратитесь к руководству **API** платформы **Топоматик АР3**.

Разработка программных модулей

Кроме, подключения *скриптов*, **Robur** позволяет подключать программные модули - сборки, написанных на различных языках программирования с использованием **Microsoft .Net Framework** для операционных систем **Windows**.

Конечно, для решения небольших задач и макросов предпочтительно использовать скрипты, однако для серьезных разработок, требующих скорости вычисления, максимального использования ресурсов компьютера или для создания богатого пользовательского интерфейса, предпочтительно использовать модули.

Модули могут выполнять практически любые задачи, от самых простых – до целых подсистем и использовать все возможности **Robur** и операционной системы, а также управлять всеми ресурсами компьютера.

Кстати, подсистемы самого комплекса **Robur** (такие как «ЦММ и Топоплан», «Подобъекты», «Геодезия» и др.) – представляют собой такие же модули, просто разрабатываемые и поддерживаемые самой компанией **Топоматик**.

С помощью этой части руководства и описанных здесь занятий Вы сможете получить исчерпывающие сведения о том, как создавать и подключать модули к **Robur** и использовать наиболее основные подсистемы **Robur**.

Для разработки мы будем использовать наиболее удобную и популярную среду разработки приложений для **.Net Framework** под операционной системой **Windows - Microsoft Visual Studio**.

Для работы Вам понадобится компьютер с **Microsoft Windows** (нами использовался **Windows 7 Профессиональная Service Pack 1** (сборка 7601) 64-разрядная) с установленными на нем **Robur** (нами были использованы «Топоматик Robur - Ознакомительная версия» 15.0.2.4) и Microsoft Visual Studio (нами использовалась версия **Microsoft Visual Studio Community 2015** английская версия).

Все создаваемые в данном руководстве примеры реализованы на языке программирования **C#** для **.Net Framework 3.5**, как самом популярном языке для **Microsoft .Net Framework**, однако в будущем Вы можете использовать любой нравящийся Вам язык программирования для этой платформы.

Для того, чтобы избежать каких-либо сложностей, связанных с ограничениями политики безопасности **Windows** или некорректной настройкой путей поиска файлов, мы рекомендуем выполнять все занятия в учетной записи с правами администратора. Демонстрационная версия **Robur** должна быть установлена (устанавливается по умолчанию) в папку «**C:\Program Files\Topomatic Robur edu 15.0**».

Также имейте ввиду, что созданные Вами модули при будут предназначены для определенной версии **Robur** (для той версии, с которой производится сборка модуля), т.е. в нашем случае это **Robur 15.0**. При загрузке модуля контролируется его соответсвие версии основной программы и при его отсутствии модули не запускаются. На различных сборках одной и той же версии Robur, естественно обновлять модули не требуется, все будет выполняться корректно.

В отличие от модулей, скрипты на макроязыках (python), описанные в предыдущих частях нашего руководства, не производят контроль версии в момент запуска и будут выполняться в любом случае. Однако при наличии в них кода, обращающегося к несуществующим или обновившимся объектам платформы **Robur**, будут завершаться аварийно.

Эту особенность обязательно надо учитывать при разработке модулей, и особенно, при поставке их Вашим пользователям.

Пример **1.** Модуль**,** выводящий сообщение

В отличие от скрипта, рассмотренного в *первом примере раздела скриптов*, мы создадим два примера - первый вариант выводит сообщение сразу после загрузки **Robur**, а уже вторая модификация выводит сообщение только, когда вызывается соответствующая команда из командной строки (или с помощью кнопки или пункта меню).

Часть 1. Вывод сообщения "Привет, Robur!" при загрузке программы

В этом занятии мы создадим простейший плагин (модуль), который после загрузки **Robur** выдаст диалоговое окно с сообщением **"Привет, Robur!"**.

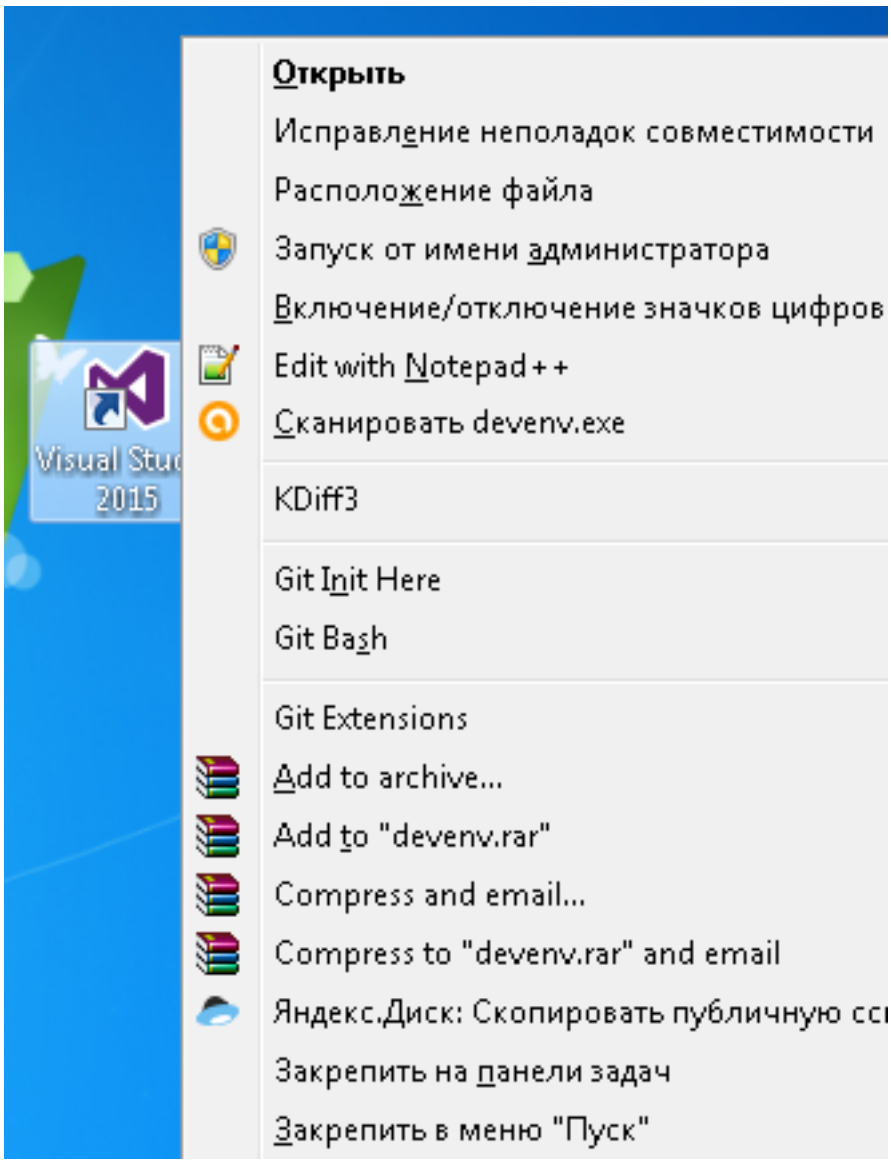
Конечно, создание такого плагина сложнее, чем аналогичного скрипта, рассмотренного в *первом примере* разработки скриптов и требует больше шагов и знаний, но если Вы собираетесь разрабатывать серьезные функции с богатым пользовательским интерфейсом (или разрабатывать коммерческие продукты), то Вам необходимо осваивать именно этот способ.

Исходный код данного примера (проект **Visaul Studio**) расположен в папке **"Sdk\samples\Modules \Sample01"**.

Приступим к разработке примера:

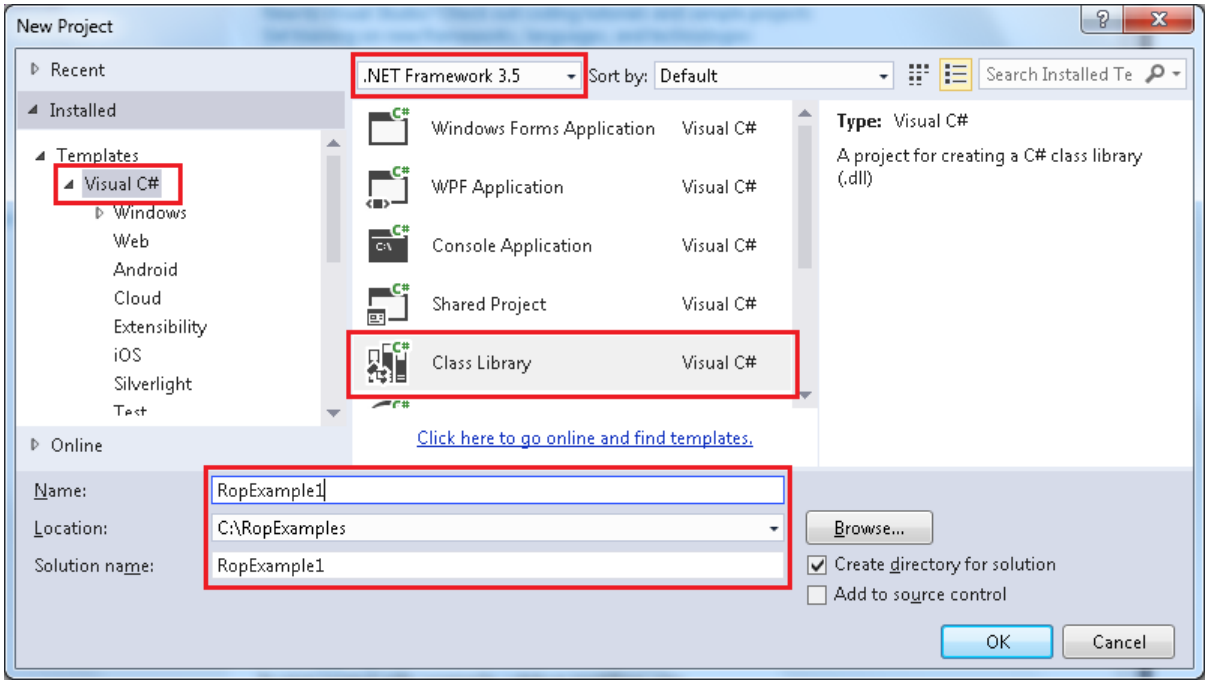
- Запустите **Microsoft Visual Studio**. Обяхательно сделайте это от имени администратора. Это необходимо для того, чтобы при компиляции Ваш модуль корректно создался в той же папке, где расположены программные компоненты Robur - то есть, чаще всего, в папке **"C:\Program Files\Topomatic..."**. Обычно Windows запрещает доступ на запись к этой папке даже для локальных пользователей, поэтому, чтобы избежать проблем со сборкой этот шаг очень важен!

  Запуск Visual Studio от имени алдминистратора с помощью ярлыка **Visual Studio** на рабочем столе показан на рисунке ниже:



Если у Вас нет пункта "Запуск от имени администратора...", то возможно, у Вас уже есть привилегии администратора локального компьютера и все будет работать. Но если в процессе сборки модуля в будущем у Вас возникнут проблемы с невозможность записать файл **Topomatic.RopExample1.dll** - это значит, что нужно вернуться к этому шагу и добиться запуска студии с соответствующими привилегиями.

- Создайте новый проект для **.Net Framework 3.5** типа **"Visual C# / Windows / Class Library"** как показано на рисунке. Для этого в главном меню выберите **"File / New / Project…"**.



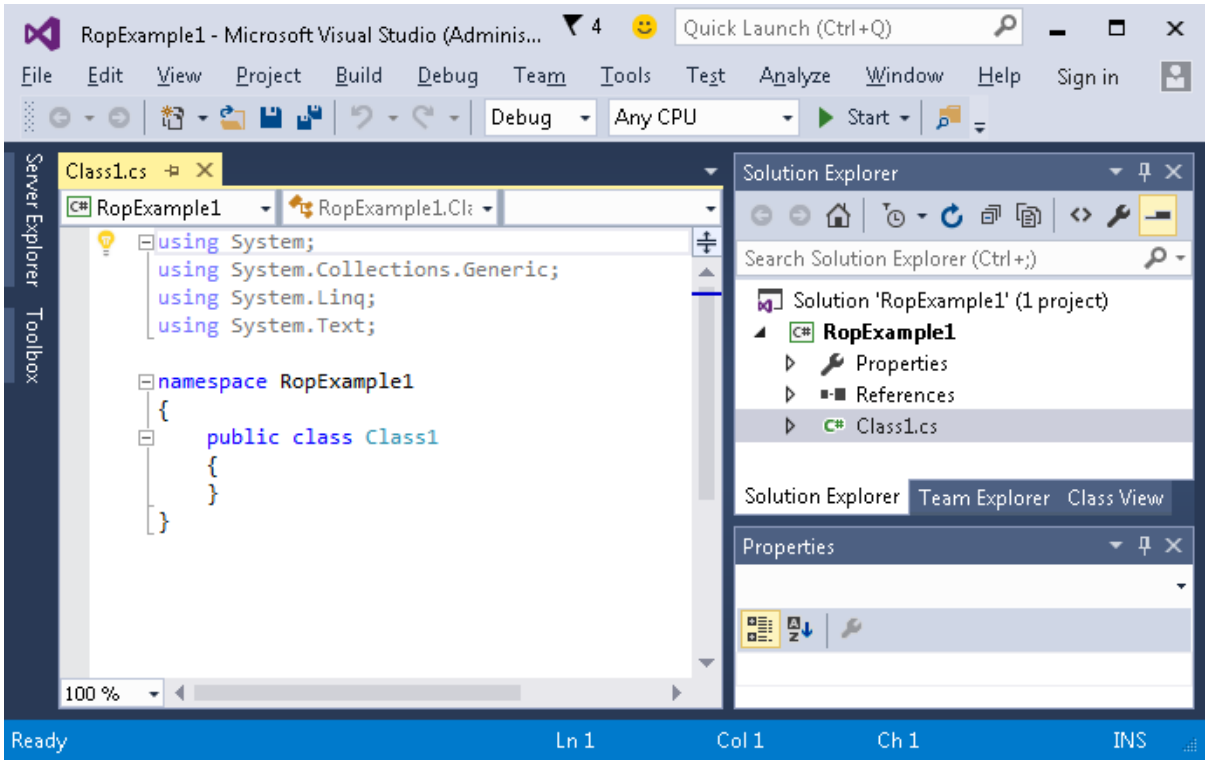
Окно **Visual Studio** с открывшимся новым проектом будет выглядеть примерно так как показано выше.

Создайте новый проект с именем **RopExample1** и решение (solution) с таким же названием.

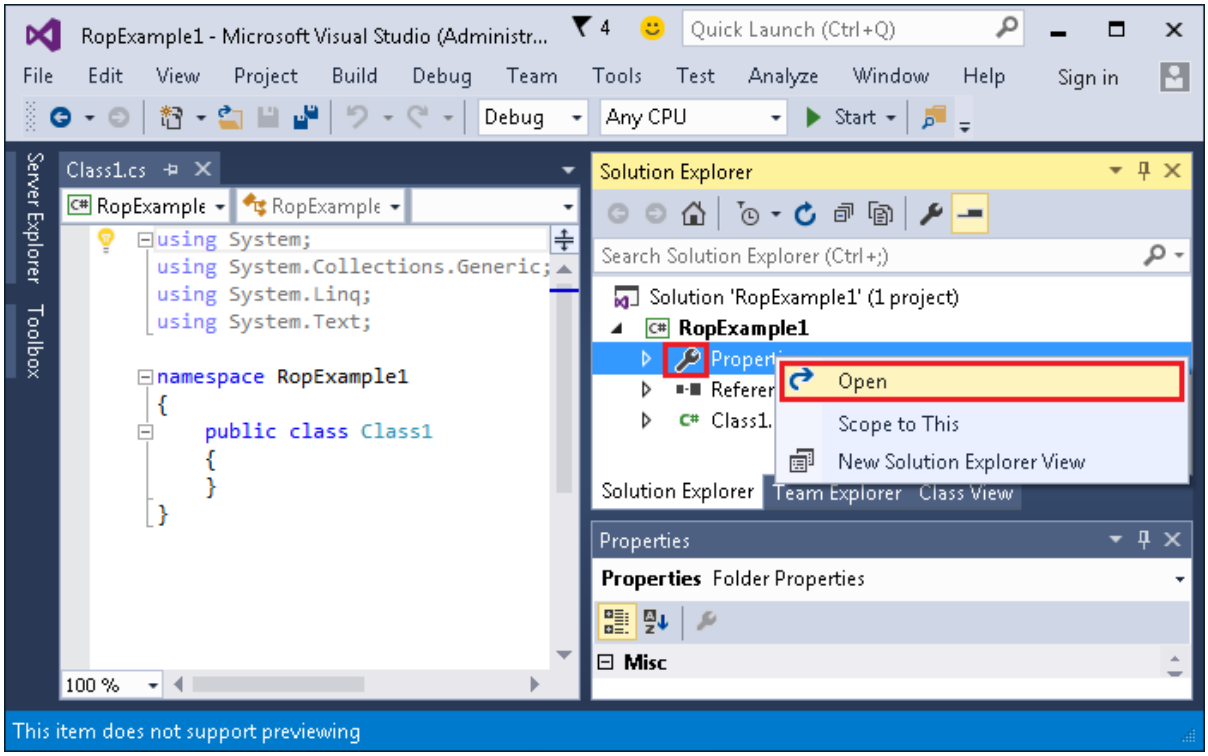
Путь по которому Вы сохраните Ваш проект может быть любым (на рисунках указана папка **"C:\RopExamples"**).

Готовый исходный код примера располагается в папке **"C:\Topomatic\Sdk\samples\Modules\Sample01"**.

- Если все было задано корректно, то окно **Visual Studio** будет выглядеть примерно следующим образом:



- После создания необходимо настроить свойства проекта. Для этого в панели **"Solution Explorer"** выберите проект **"RopExample1"** и, вызвав контекстное меню (правая кнопка мыши), выберите в нем пункт **"Properties…"** как показано на рисунке.



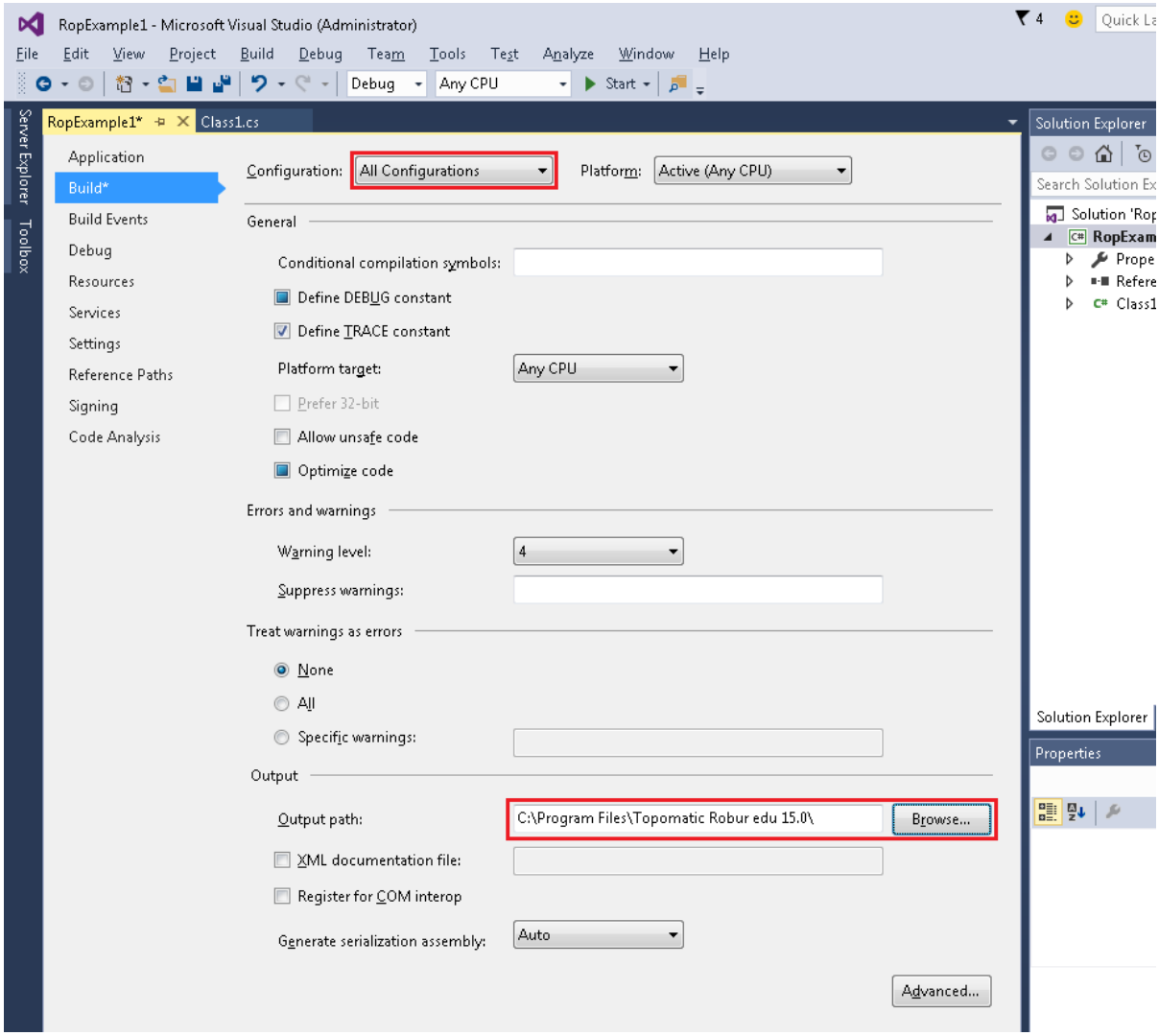
- На вкладке **"Build"** необходимо для всех собираемых конфигураций (список **"Configuration"** устанавливается в положение **"All Configurations"**) необходимо задать в качестве пути для создаваемых сборок папку, в которую установлен **Robur** (в нашем случае, это – **"C:\Program Files\Topomatic Robur edu 15.0\"**).

  БУДЬТЕ ВНИМАТЕЛЬНЫ!

  Для того, чтобы сборка прошла корректно Ваш пользователь (пользователь, от имени которого запущена **Visual Studio**) должен иметь права на запись в эту папку.

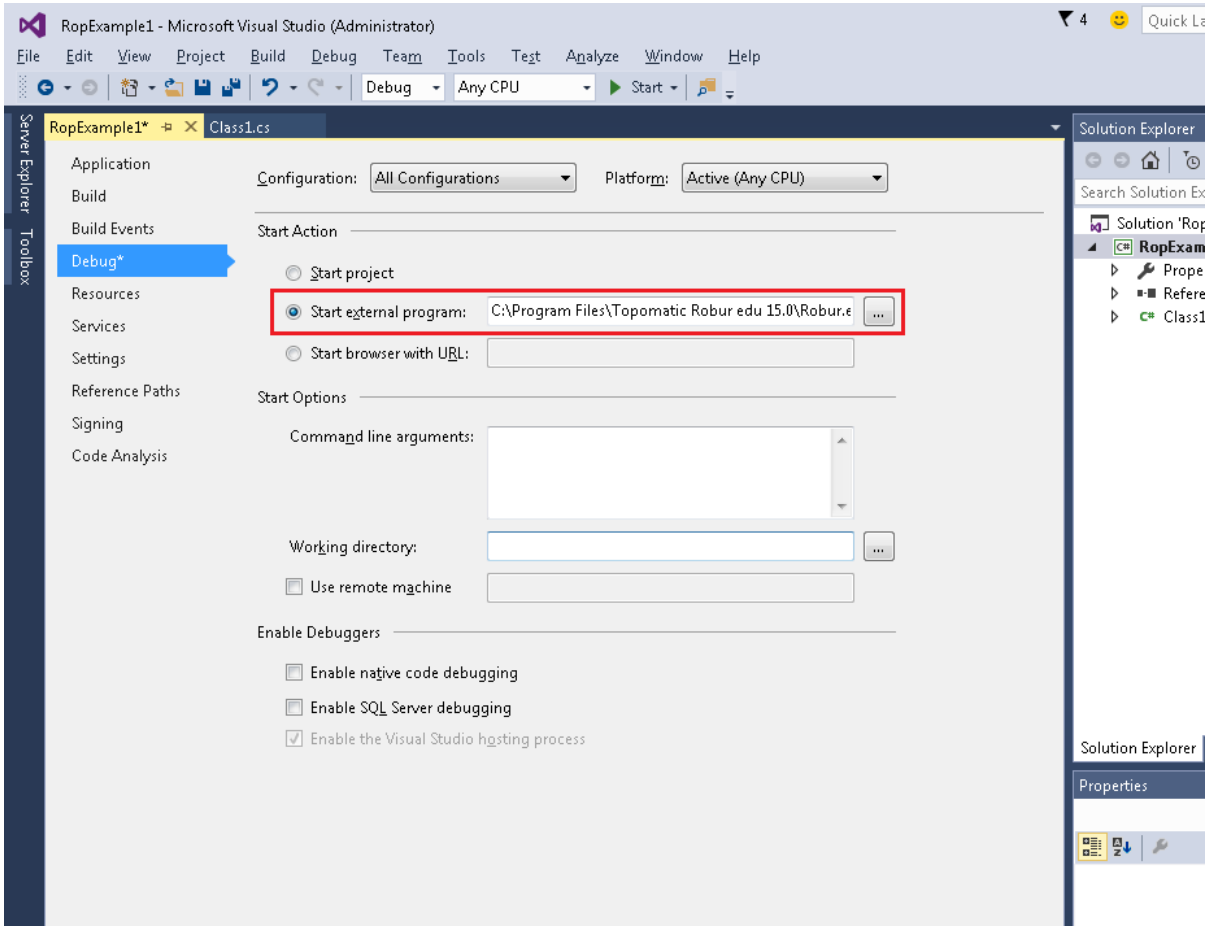
  Чаще всего, доступ на запись обычным пользователям в папку **Program Files** запрещен, так что прийдется его открыть или запускать **Visual Studio** от имени администратора.

  Если этого не сделать, то при сборке модуля нашего примера мы получим соответствующее сообщение об ошибке.

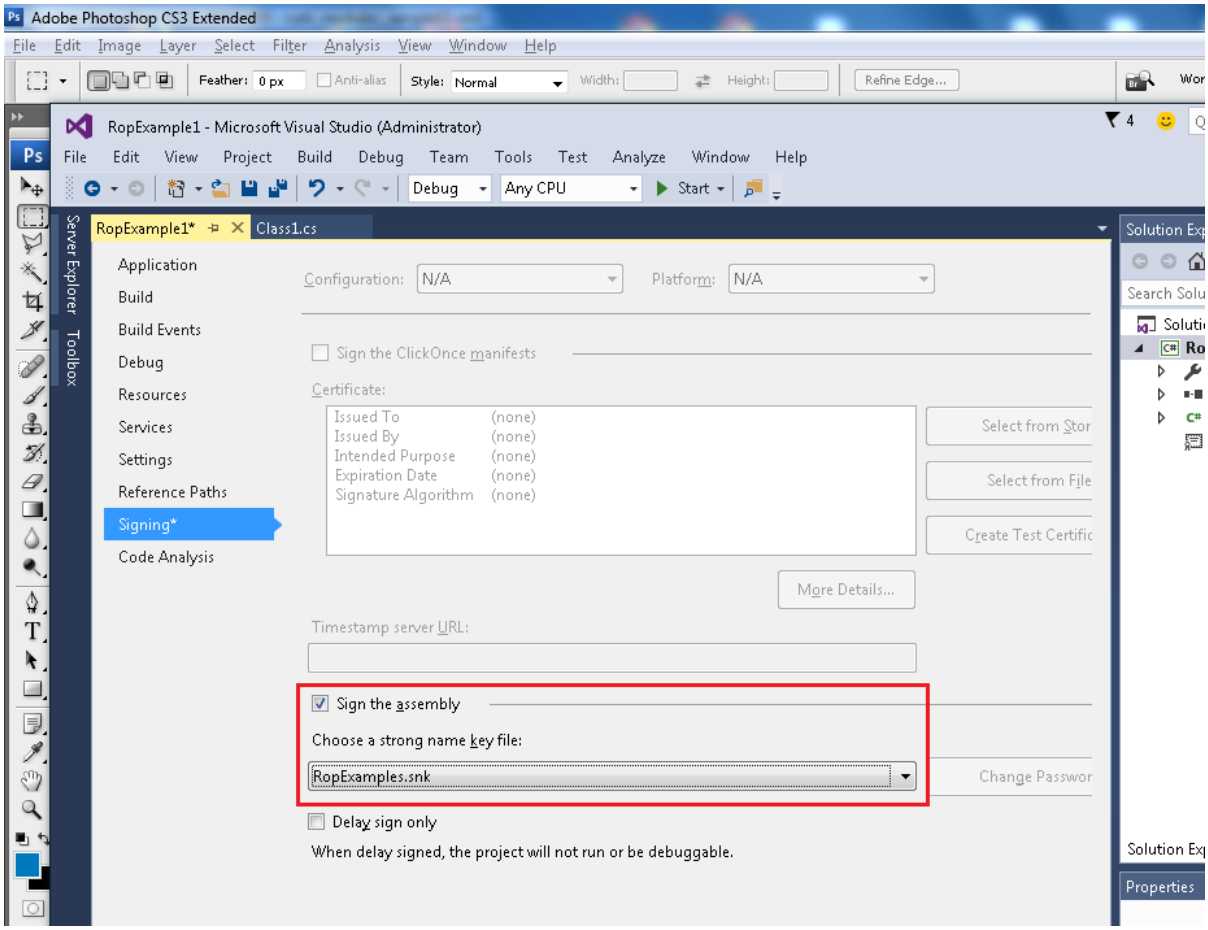


- Далее необходимо перейти на вкладку **"Debug"** и установить в качестве программы для запуска исполняемый файл **Robur** (в нашем случае это – **"C:\Program Files\Topomatic Robur edu 15.0\Robur.exe"**).

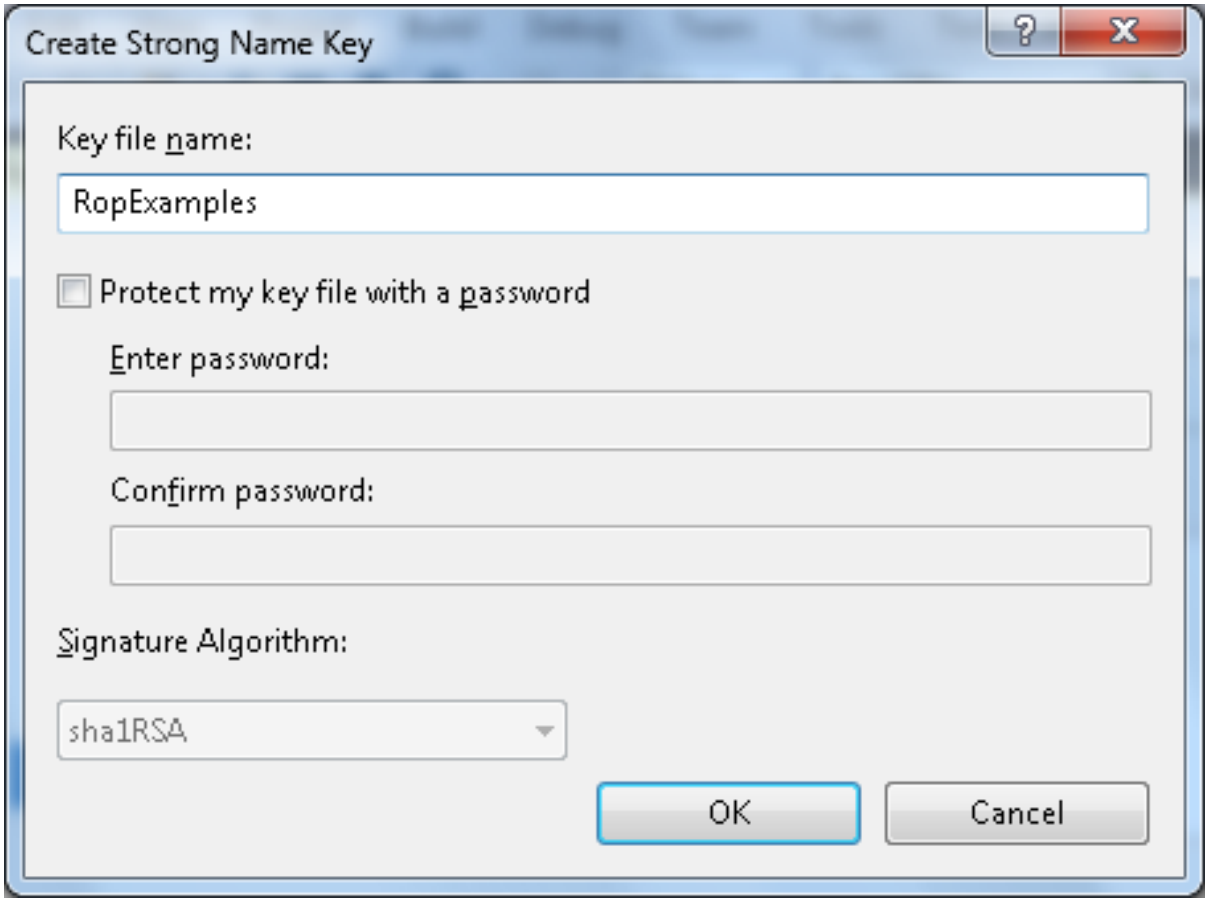
  Это необходимо для того, чтобы при запуске отладчика **Visual Studio** запускался **Robur**, который затем бы уже подключал и загружал ваш модуль.



- Еще одним важным условием для подключения модуля к **Robur** - это цифровая подпись. Поэтому, необходимо зайти на вкладку **"Signing"** и установить галочку **"Sign the assembly"**, выбрав в списке Ваш файл ключа для цифровых подписей (или создав новый, как это было сделано в нашем примере - для этого мы в списке выбирали "**New...**" и задали имя файла ключа **"RopExamples.snk"**.). Смотрите этот шаг на рисунках ниже:



Окно создания файла ключа цифровой подписи:

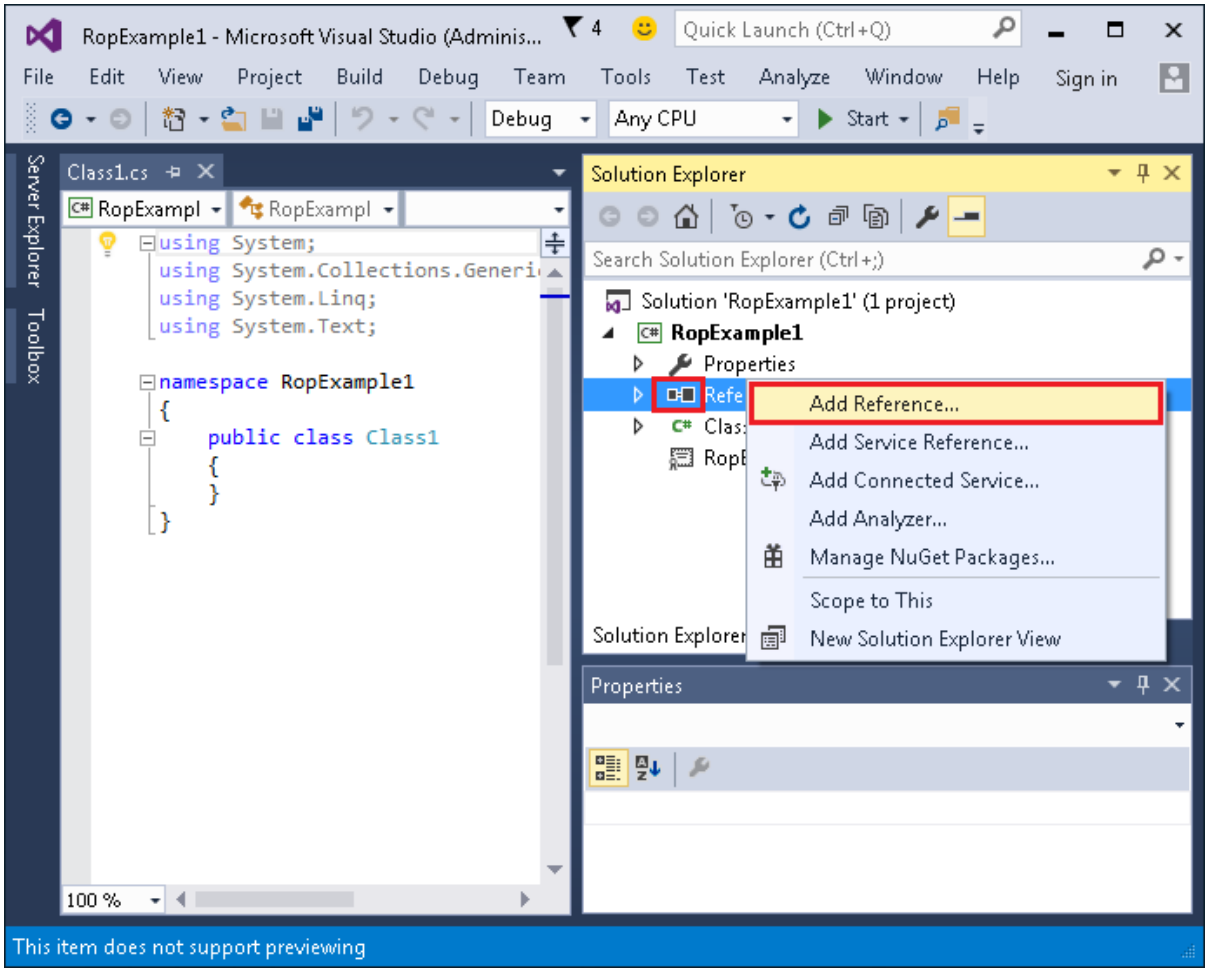


- После того, как все необходимые манипуляции со свойствами проекта выполнены, необходимо добавить ссылки на необходимые для функционирования Вашего модуля сборки. Часть сборок автоматически подключаются при создании проекта **Visual Studio**, остальные необходимо подключать в процессе разработки при необходимости использования их функционала.

  Так как наш пример использует класс **MessageBox** - диалоговое окно с сообщением из сборки **Microsoft System.Windows.Forms.dll**, то необходимо подключить ее.

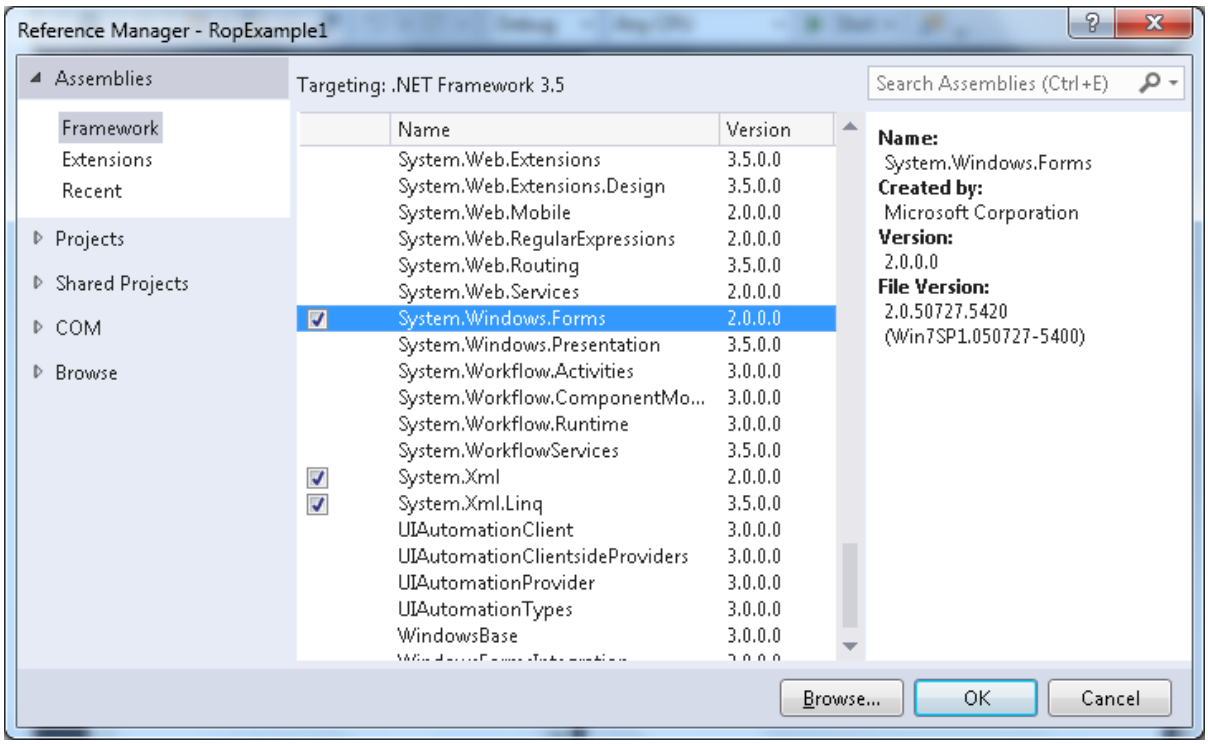
  Также для функционирования модулей Robur необходимо подключать сборки **Topomatic.Stg.dll** и **Topomatic.ApplicationPlatform.dll**.

  Для этого вызовите контекстное меню для элемента **"References"** в **Solution Explorer Visual Studio** и выберите в нем пункт **"Add Reference..."**, как показано на рисунке ниже:

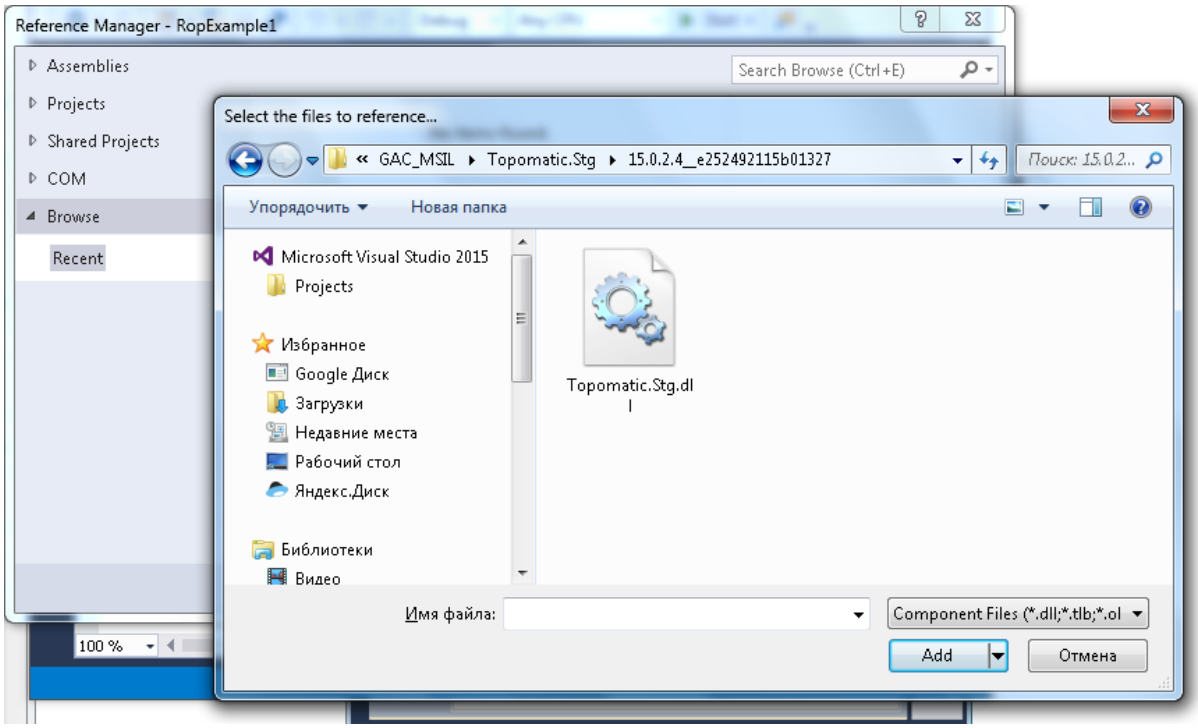


- В открывшемся окне **"Add Reference"** выбираем нужные сборки. Следует иметь ввиду, что зарегистрированные системные сборки **.NET** выбираются на разделе **"Assemblies - Framework"**, а незарегистрированные, необходимо указывать как файлы вручную на вкладке **"Browse"**.

  Итак: добавляем **System.Windows.Forms**:



- Добавляем **Topomatic.Stg.dll** и **Topomatic.ApplicationPlatform.dll** из папок **C:\Windows\assembly\GAC_MSIL\Topomatic.Stg\...** и директории, в которую установлен **Robur** (в нашем случае - это **"C:\Program Files\Topomatic Robur edu 15.0"**) соответственно:

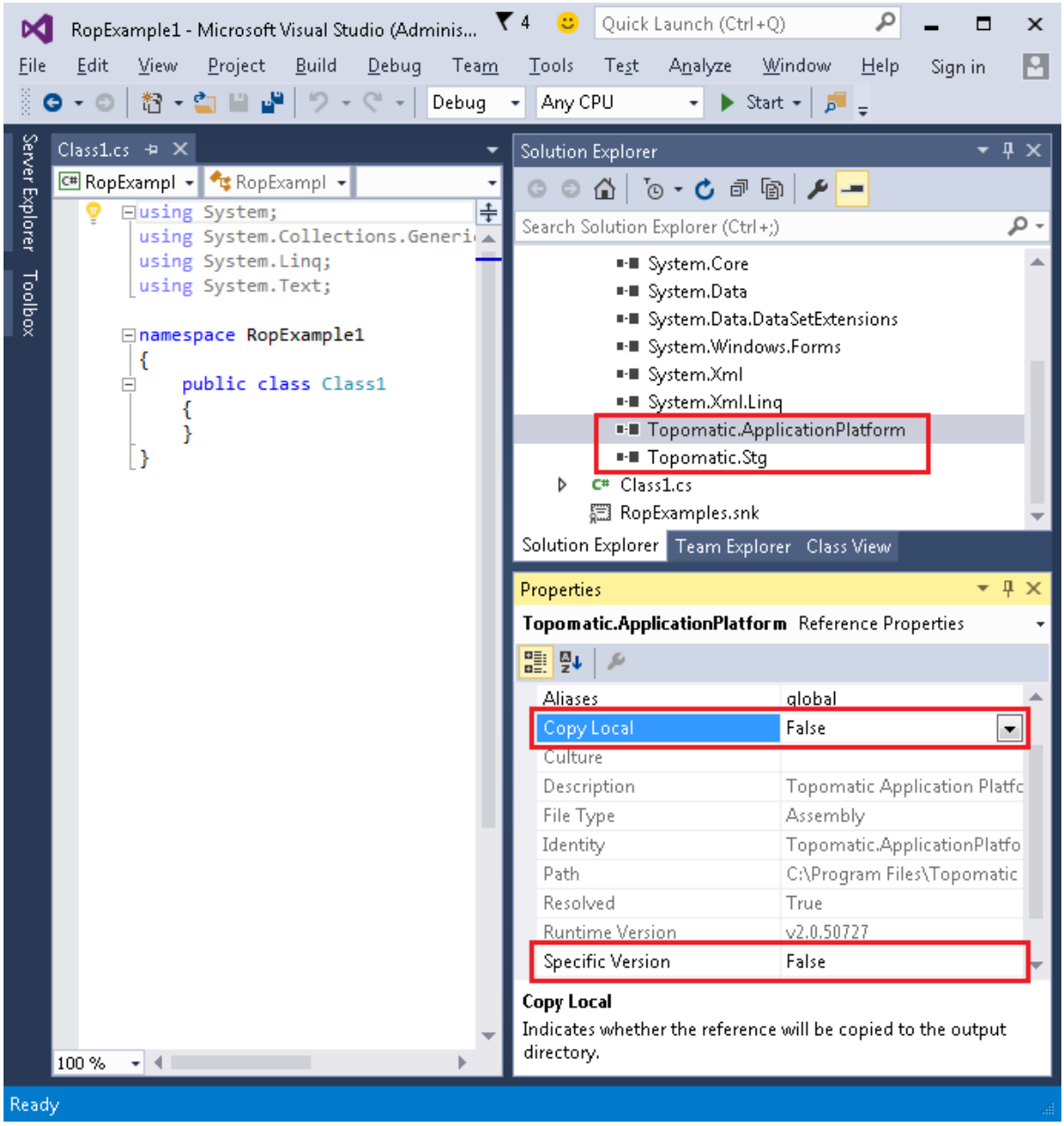


- После добавления сборок в проект, у сборок подключаемых как файл надо отключить атрибуты **"Copy Local"** и **"Specific Version"**.

  Первый флаг при сборке версии предполагает, что подключаемая библиотека будет копироваться в папку, в которую будет выгружена Ваш собираемый плагин. Это неудобно, так как появятся некоторые лишние копии.

Второй флаг означает, что для работы Вашего модуля нужна именно конкретная версия подключаемой библиотеки. Если версия окажется другой, то Ваш модуль не заработает, хотя, как правило, другая версия библиотеки будет подходящей и такие жесткие требования не нужны.

Отключение этих флагов для выбранных двух библиотек показано ниже:



- После того, как все свойства проекта подготовлены, необходимо создать два класса: класс хоста плагинов (Plugin Host) и класс модуля, для последующего подключения к **Robur**.

  Строго говоря, к платформе **Robur** подключается не файл сборки dll со всем ее содержимым, а модули. Однако, для того, чтобы сообщить **Robur**, какие именно модули нужно подключать к **Robur**, необходим вспомогательный объект - хост плагина (**PluginHost**).

  Если рассматривать вариант подключения *скриптов (макрокоманд)* к **Robur**, то можно увидеть, что сущность хоста плагина в нем отсутствует (плагином фактически является сам файл скрипта), а сущности модулей в обоих случаях аналогичны.

Итак, создаем классы хоста и модуля. Для класса хоста можно использовать создаваемый по умолчанию в проекте **Visual Studio** класс **Class1**, переименовав его в **RopExample1PluginHost**.

Для этого щелкаем в **Solution Explorer** на пункте **Class1.cs** правой кнопкой мыши и, вызвав контекстное меню, выбираем **"Rename"**.

В качестве нового имени задаем **"RopExample1PluginHost.cs"** и соглашаемся с предложением переименовать все ссылки на этот класс.

- Далее, дважды щелкаем на **"RopExample1PluginHost.cs"** и переходим к его редактированию.

Необходимо изменить код класса следующим образом:

```
public class RopExample1PluginHost : PluginHost
{
    protected override Type[] GetModules()
    {
        return new Type[] { typeof(RopExample1Module) };
    }

    public override string PluginName
    {
        get { return "RopExample1"; }
    }
}
```

Первая строка кода - это объявление класса для Вашего плагина:

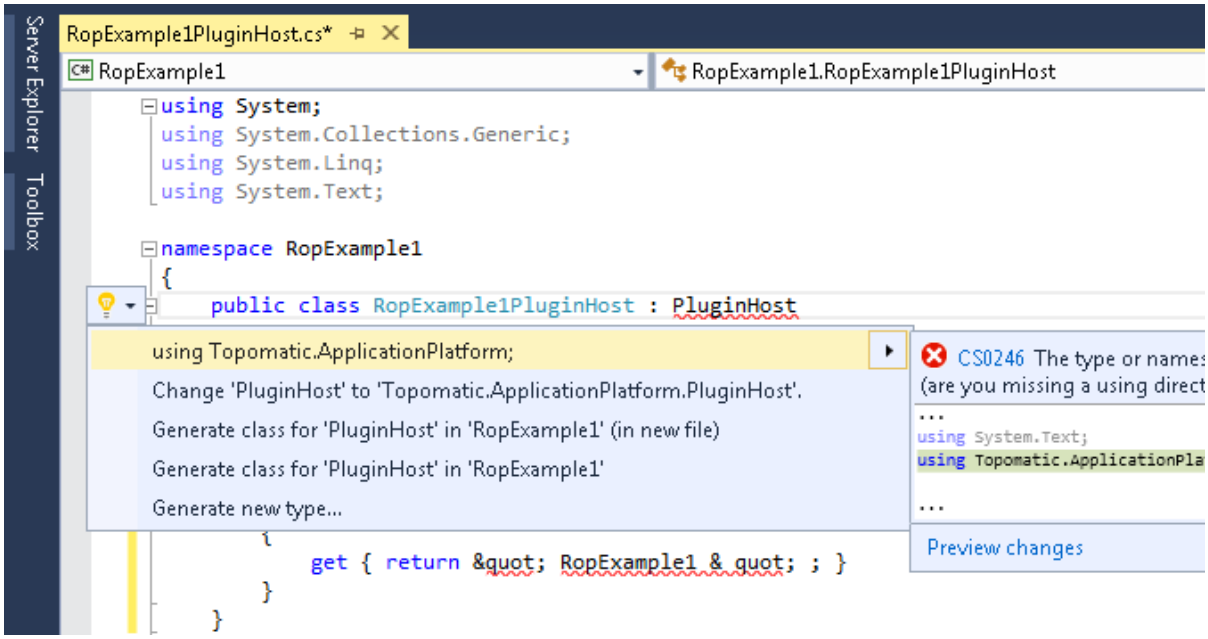
```
public class RopExample1PluginHost : PluginHost
```

Класс Вашего плагина наследуется от класса **PluginHost** платформы **Robur** (**PluginHost** объявлен в сборке **Topomatic.ApplicationPlatform.dll**).

Далее код, переопределяет функцию базового класса **PluginHost GetModules**, которая возвращает массив типов модулей, которые должен подключать к Robur Ваш плагин. В нашем случае - это один единственный класс модуля **RopExample1Module**. Затем переопределятся свойство **PluginName**, возвращающее имя Вашего плагина - **RopExample1**.

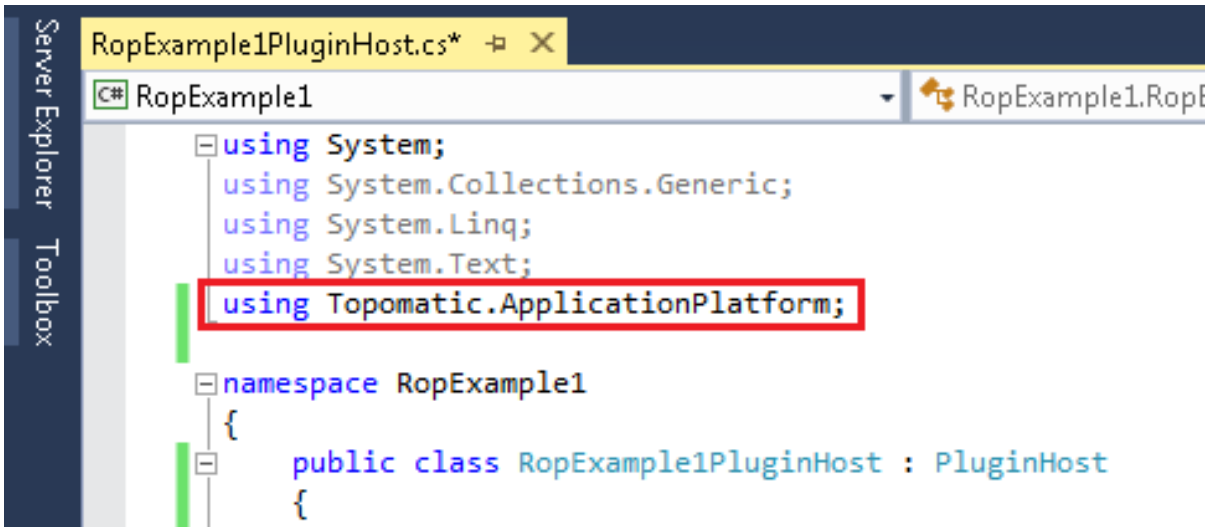
Чтобы код корректно работал, необходимо подключить пространство имен **Topomatic.ApplicationPlatform** (написать в группе строк с инструкциями using в самом верху файла **RopExample1PluginHost** строку **"using Topomatic.ApplicationPlatform"**).

Новые версии **Visual Studio** (в том числе и используемая нами **Visual Studio 2015**) позволяют автоматически подключать пространства имен для используемых типов. Для этого установите курсор на имя используемого типа (в нашем примере это **PluginHost** в строке **class RopExample1PluginHost : PluginHost**) и нажмите **"Ctrl + ."** (клавиша с точкойи буквой "Ю" на клавиатуре) и в контекстном меню выберите **"using Topomatic.ApplicationPlatform"**, как показано на рисунке ниже:

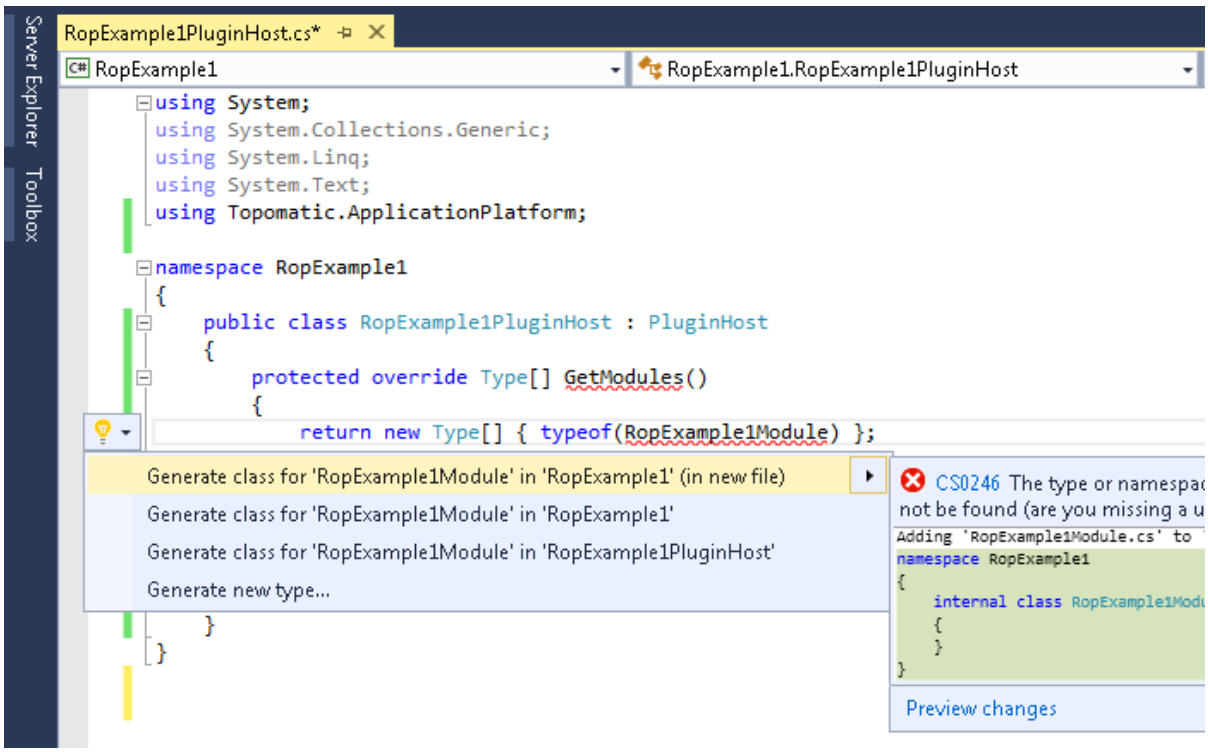


Строка с **using** будет добавлена автоматически. Это крайне удобная возможность, так как с помощью комбинации клавиш **"Ctrl + ." Visual Studio** позволяет автоматически выполнять и другие формальные операции с кодом (создавать функции и классы, добавлять полные пути типов из других пространств имен, добавлять директивы **using** и многое другое.

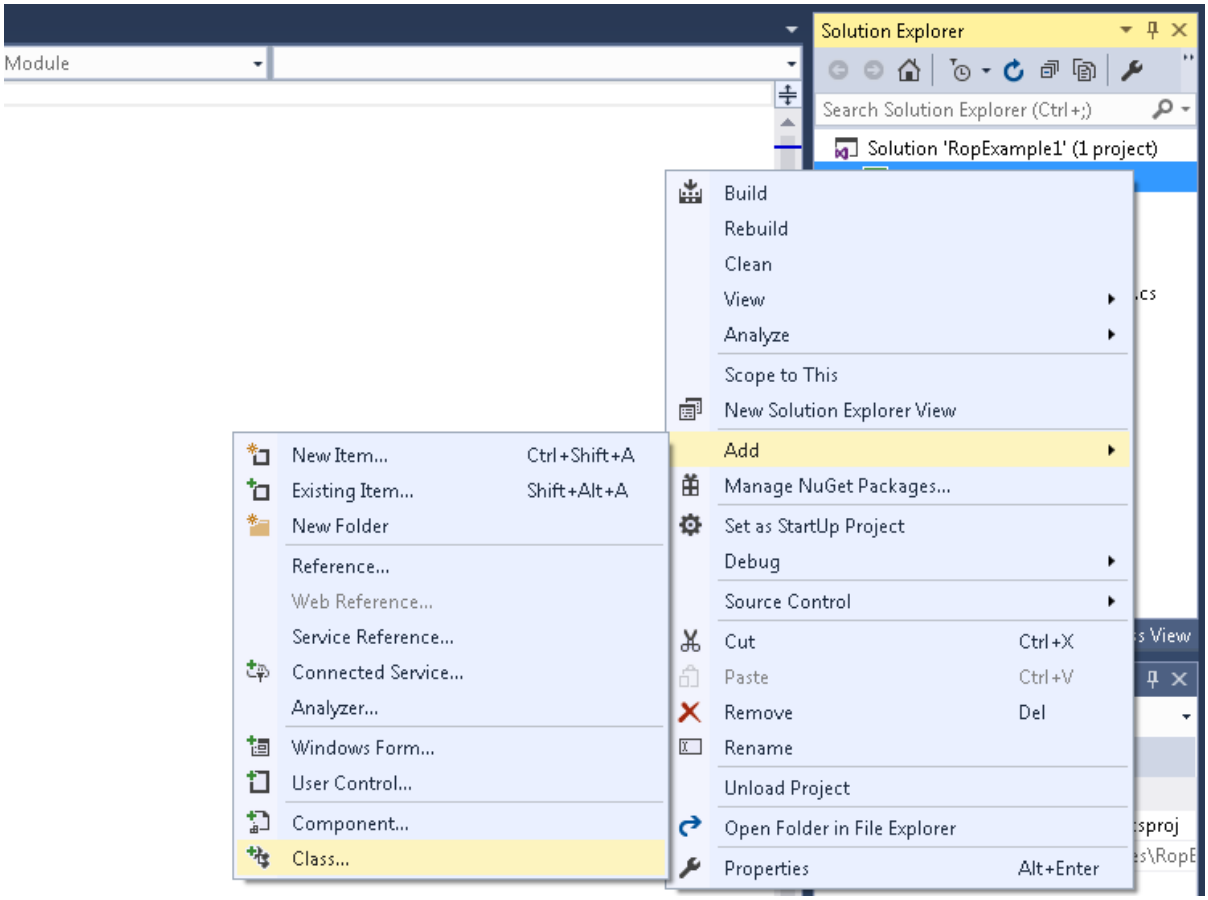
Окно кода с корректно заданным **using** приведена на рисунке ниже:



Также в коде **Visual Studio** выделит, как некорректный тип и **RopExample1Module** в блоке кода **typeof(RopExample1Module)**, потому что мы еще не создали класс нашего модуля. Кстати говоря, можно снова воспользоваться комбинацией **"Ctrl + ."**, установив курсор на **RopExample1Module** и автоматически создать файл с нужным классом.

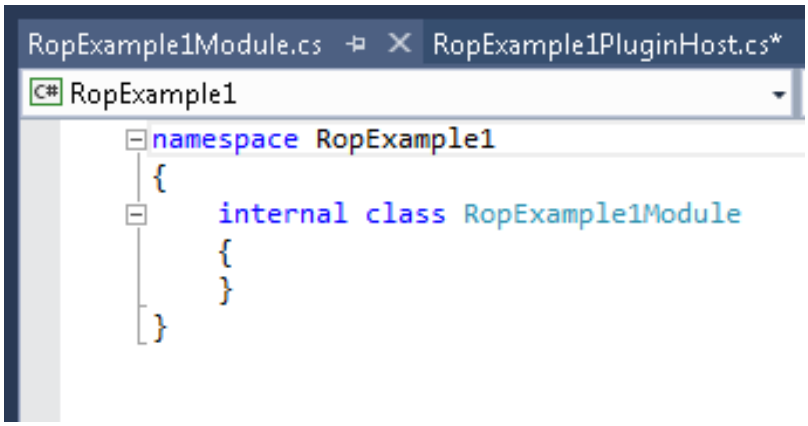


Однако рассмотрим также и создание класса модуля вручную. Для создания класса модуля в дереве **Solution Explorer** выберите элемент проекта **RopExample1** и вызовите его контекстное меню. В нем выберите пункт **"Add"**, как показано на рисунке ниже:



В открывшемся окне оставьте выбранным тип создаваемого элемента **Class** и укажите имя файла **RopExample1Module.cs**.

После открытия созданного файла с классом, окно кода должно выглядеть следующим образом:



Необходимо модифицировать его код так:

```
namespace RopExample1
{
    public partial class RopExample1Module : Module
    {
        public RopExample1Module()
        {
            InitializeComponent();
        }

        public RopExample1Module(IContainer container)
        {
            container.Add(this);
            InitializeComponent();
        }

        protected override void OnApplicationInitialized(object sender,
 ApplicationInitializedEventArgs e)
        {
            base.OnApplicationInitialized(sender, e);
            MessageBox.Show("Привет, Robur!");
        }
    }
}
```

И сразу же добавить uses:

```
using Topomatic.ApplicationPlatform;
using System.ComponentModel;
using System.Windows.Forms;
```

Например, способом с "Ctrl + .", описанным выше.

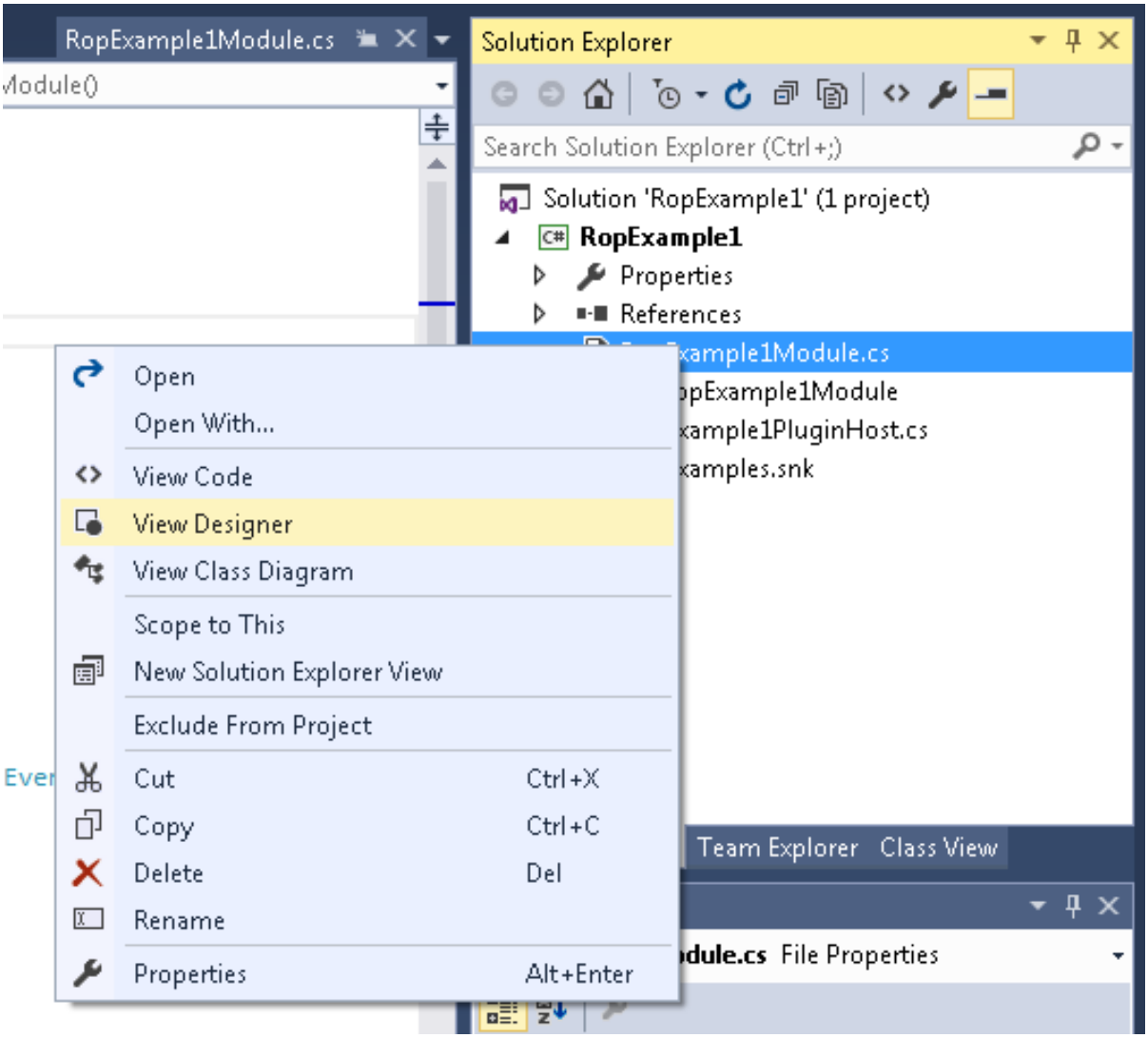
Комментарии к коду:

- Строка `public partial class RopExample1Module : Module` объявляет класс Вашего модуля как наследника от класса **Module** платформы **Robur**.

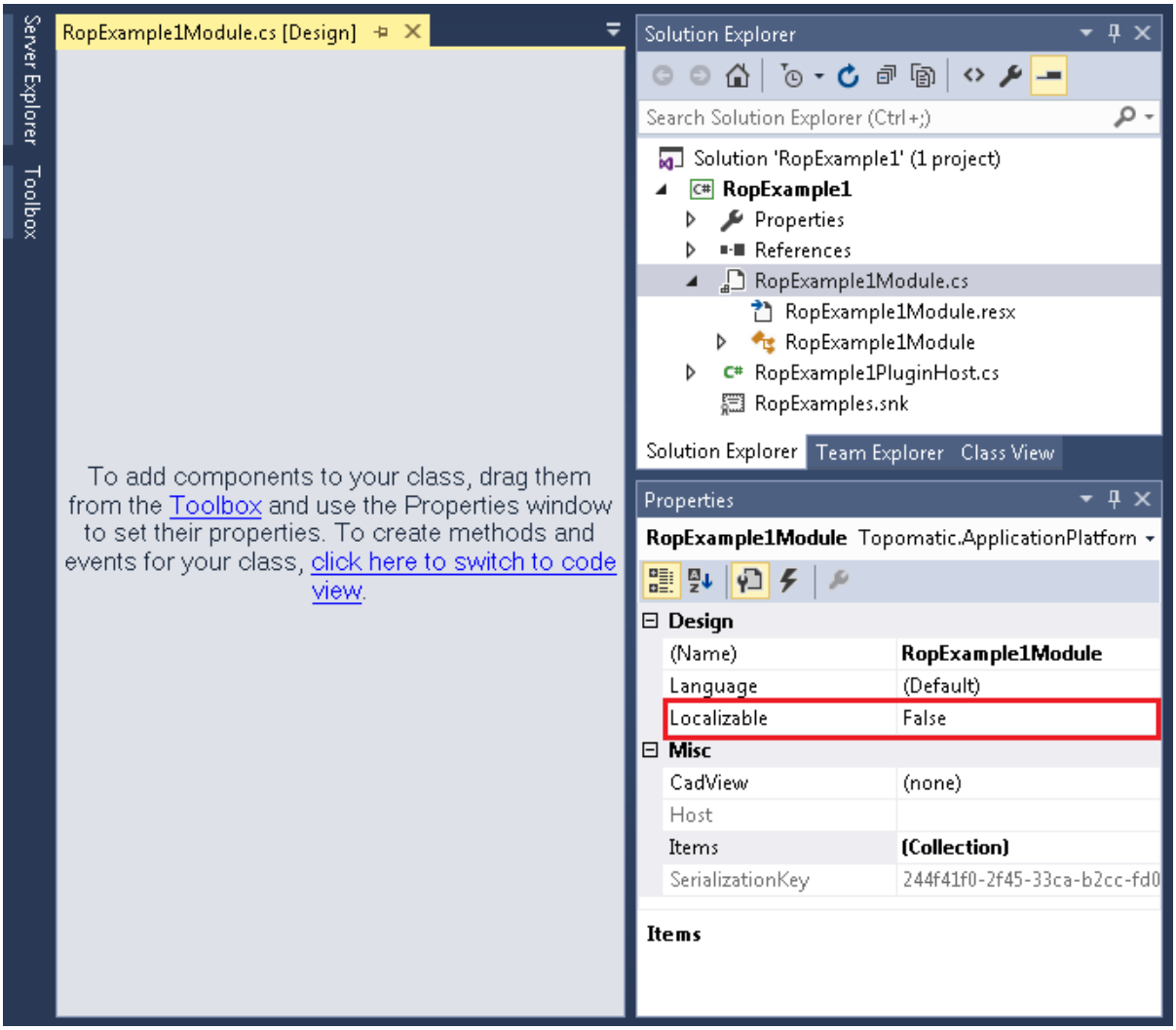
  Инструкция **partial** указывает на то, что класс разбит на несколько файлов с исходным кодом. Дело в том, что класс **Module** - это компонент, и при сборке для него будут созданы файлы ресурсов и кода, реализующего создания компонента по этим ресурсам.

Компонент - это такой класс, свойства и элементы которого могут редактироваться с помощью визуального редактора **Visual Studio**. Как редактировать эти совйства будет рассмотрено позже при добавлении **action'ов.**

Так как мы не будем редактировать никакие свойства модуля, но нам нужно, чтобы **Visual Studio** все-таки сгенерировала файлы ресурсов, нам прийдется открыть модуль на редактирование в режиме дизайнера (щелкнуть правой кнопкой мыши на элементе RopExample1Module и в контекстном меню выбрать View Designer).



В этом режиме, в дереве Solution Explorer и в совйствах модуля поменять "туда и обратно" с False на True и обратно свойство, например **Localizable**. Как это показано на рисунке:



Такие своеобразные манипуляции "вынудят" Visual Studio создать для модуля пустой файл ресурсов и сгенерировать соответствующий код.

- **Будьте внимательны!**

  Если этого не сделать, то проект не соберется, так как функции InitializeComponent() не будут найдены.
- Далее следует два конструктора **RopExample1Module(...)** для класса нашего модуля.
- И последняя функция **OnApplicationInitialized(...)** - как раз и есть то самое место, где наш модуль выводит сообщение.

```
protected override void OnApplicationInitialized(object sender,
 ApplicationInitializedEventArgs e)
{
    base.OnApplicationInitialized(sender, e);
    MessageBox.Show("Привет, Robur!");
}
```

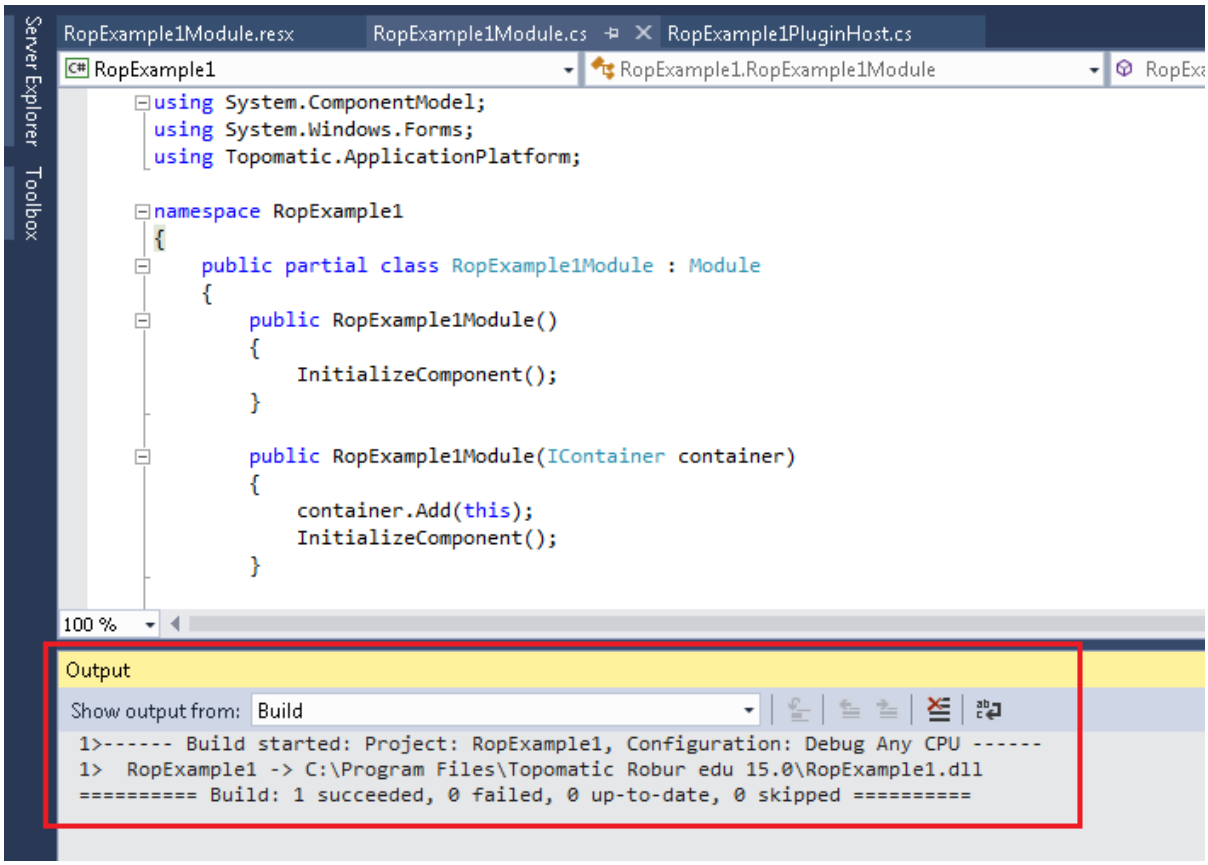
Функция перекрывает одноименную функуию базового класса **Module** и сначала вызывает ее - `base.OnApplicationInitialized(sender, e);`, а потом выводит сообщение.

OnApplicationInitialized(...) вызывается платформой у всех модулей в момент, когда **Robur** открылся и все его модули и компоненты загрузились.

Платформа **Robur** вызывает много других функций, объявленных в классе Module при наступлении соответствующих условий (при загрузке и сохранении настроек для Вашего плагина, при создании, загрузке или закрытии проектов и др.). Перекрывая эти функции в Ваших модулях, Вы можете выполнять различные операции, необходимые Вам. Кроме того в **Module**, имеется функция **OnHandleEvent()**, в которую приходят еще множество событий от платформы и других ее модулей, в том числе сторонних разработчиков. Отфильтровав нужные Вам, Ваш модуль также может выполнять определенные действия.

- Наш модуль готов, можно приступить к его сборке. Выбираем в главном меню **Visual Studio "Build - Build Solution"** или нажимаем **"Ctrl + Shift + B"**.

  Если все было сделано корректно, то вы получите соответствующее сообщение в окне сообщений **Visual Studio**:



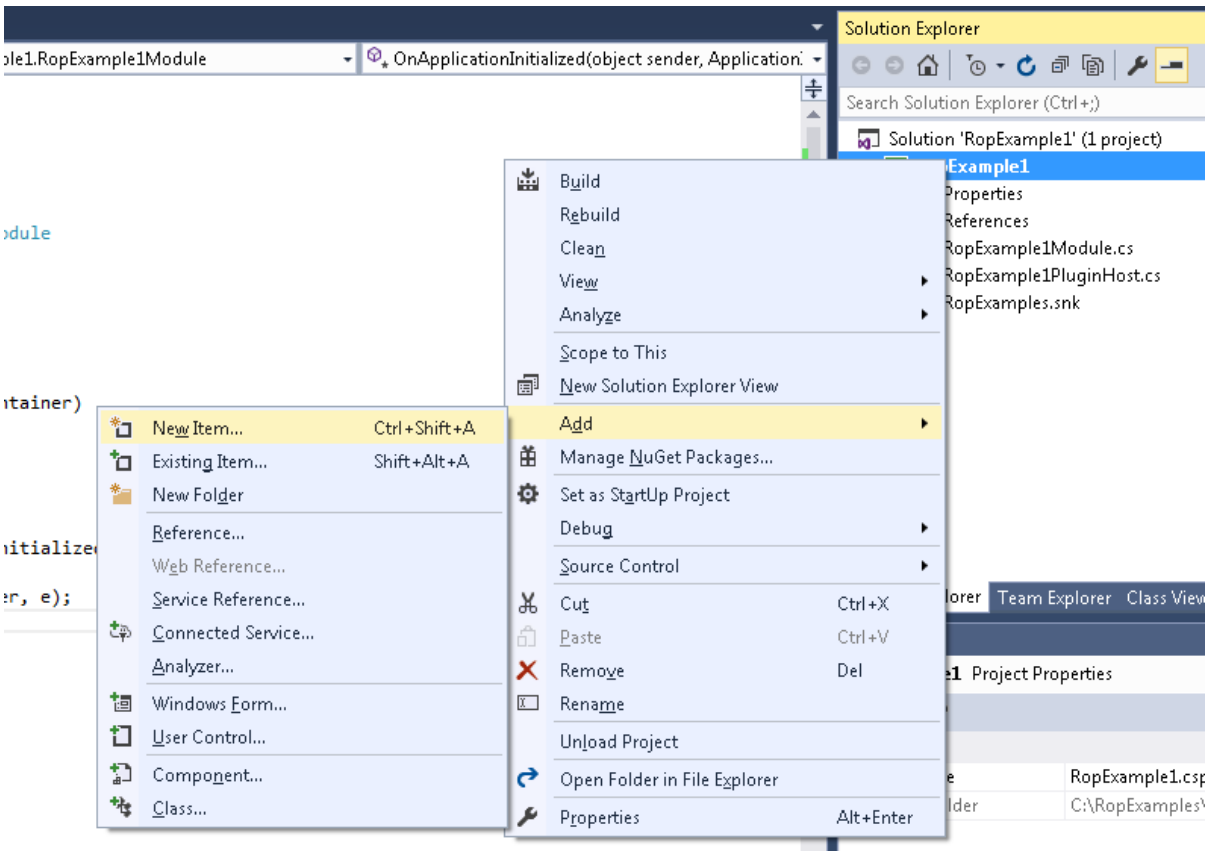
  Если где-то ошибки в коде, то Вы увидите их. В случае ошибок доступа, проверьте, не забыли ли Вы запустить **Visual Studio** от имени администратора, как это было описано в начале раздела.

- Заключительным шагом нашего примера является создание файла **"AddIn"**. Это- текстовый файл (XML), который помещается в папку с исполняемым файлом **Robur**, в котором указывается, что надо загрузить Ваш плагин при загрузке **Robur**.

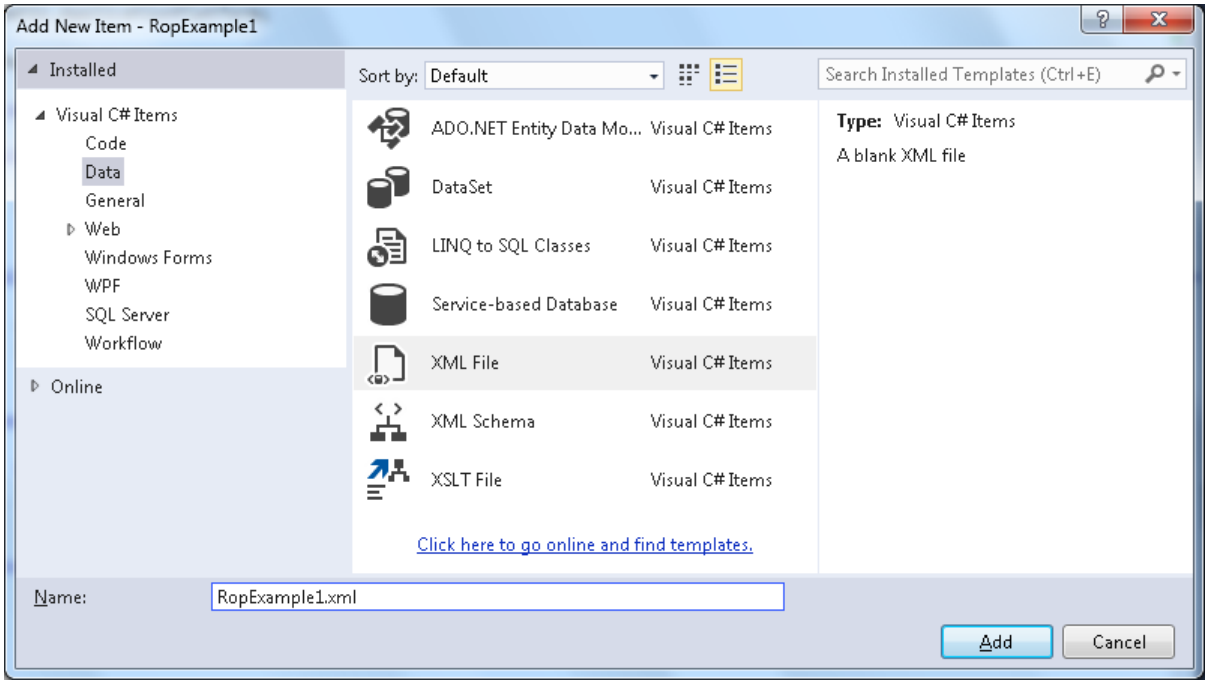
  Фактически, при запуске **Robur**, платформа сканирует папку и считывает все файлы с расширением **"AddIn"**. Считав их, платформа, сортирует их в указанном порядке и пытается загрузить и присоединить к **Robur** все заданные модули.

  Удобнее всего создать файл **"AddIn"**, также в **Visual Studio**, указав, чтобы при сборке он копировался в папку с **Robur**.

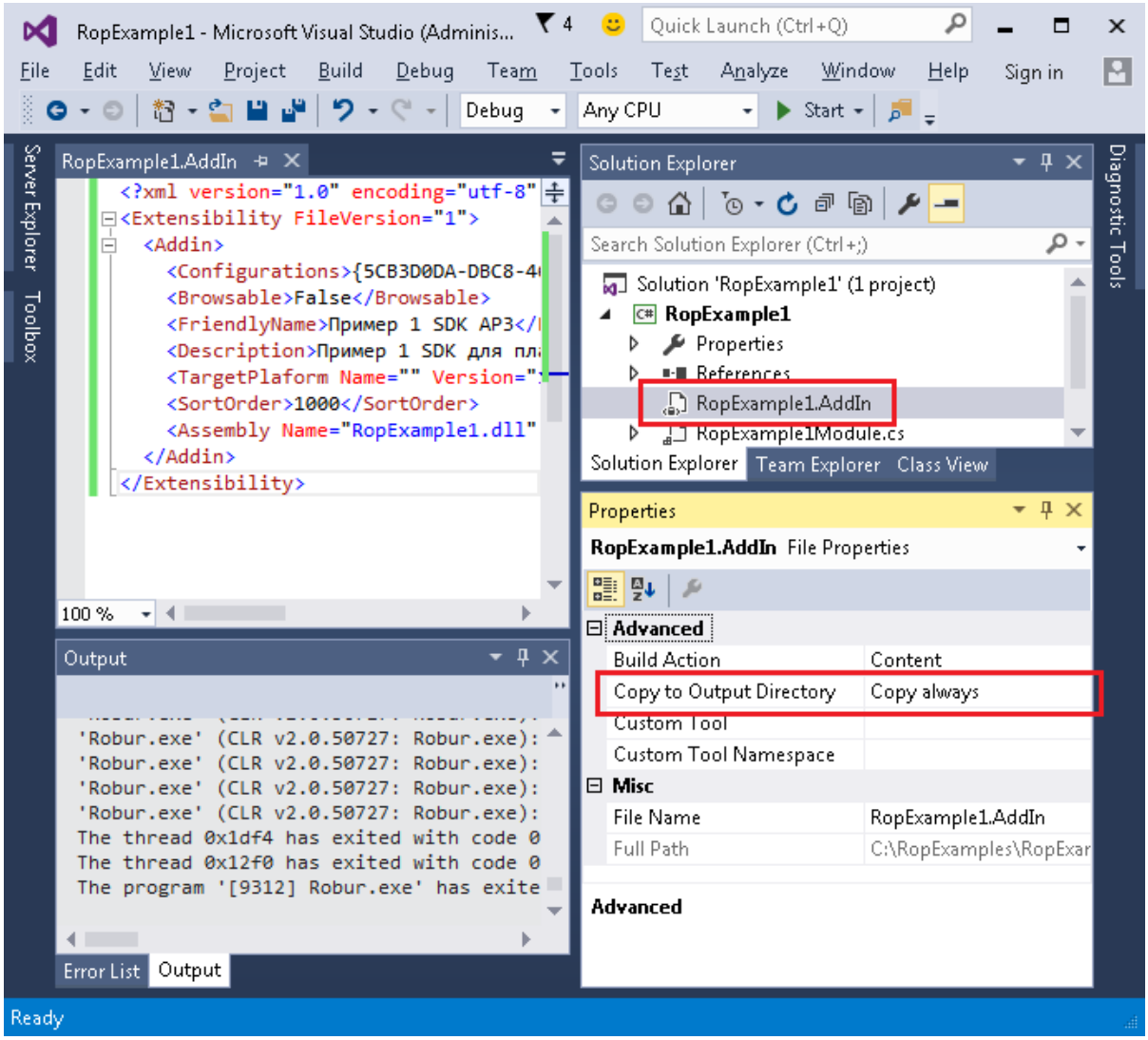
  Для этого вызовите контекстное меню, щелкнув на элементе проекта **RopExample1Module** в **Solution Explorer** и выбрав пункт **"Add - New Item".**



А в открывшемся окне выбрать тип элемента **"Data"** и **"XML File"**, а имя файла **"RopExample1Module.xml"**.

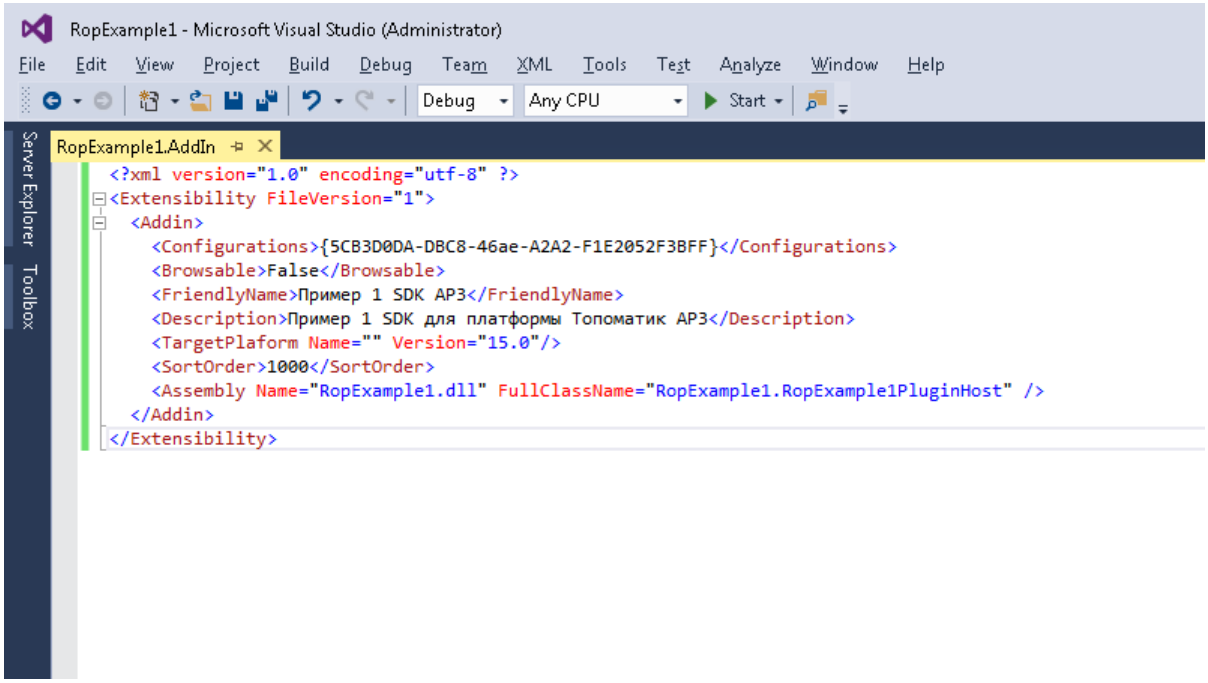


К сожалению, сразу нельзя задать расширение **"AddIn"**, поэтому переименуем файл, щелкнув по нему в **Solution Explorer**, а заодно укажем в свойствах, чтобы файл копировался при сборке в папку **Robur**.



В редактор XML вставляем содержимое:

```
<?xml version="1.0" encoding="utf-8" ?>
<Extensibility FileVersion="1">
  <Addin>
    <Configurations>{5CB3D0DA-DBC8-46ae-A2A2-F1E2052F3BFF}</
Configurations>
    <Browsable>False</Browsable>
    <FriendlyName>Пример 1 SDK AP3</FriendlyName>
    <Description>Пример 1 SDK для платформы Топоматик AP3</Description>
    <TargetPlaform Name="" Version="15.0"/>
    <SortOrder>1000</SortOrder>
    <Assembly Name="RopExample1.dll"
 FullClassName="RopExample1.RopExample1PluginHost" />
  </Addin>
</Extensibility>
```

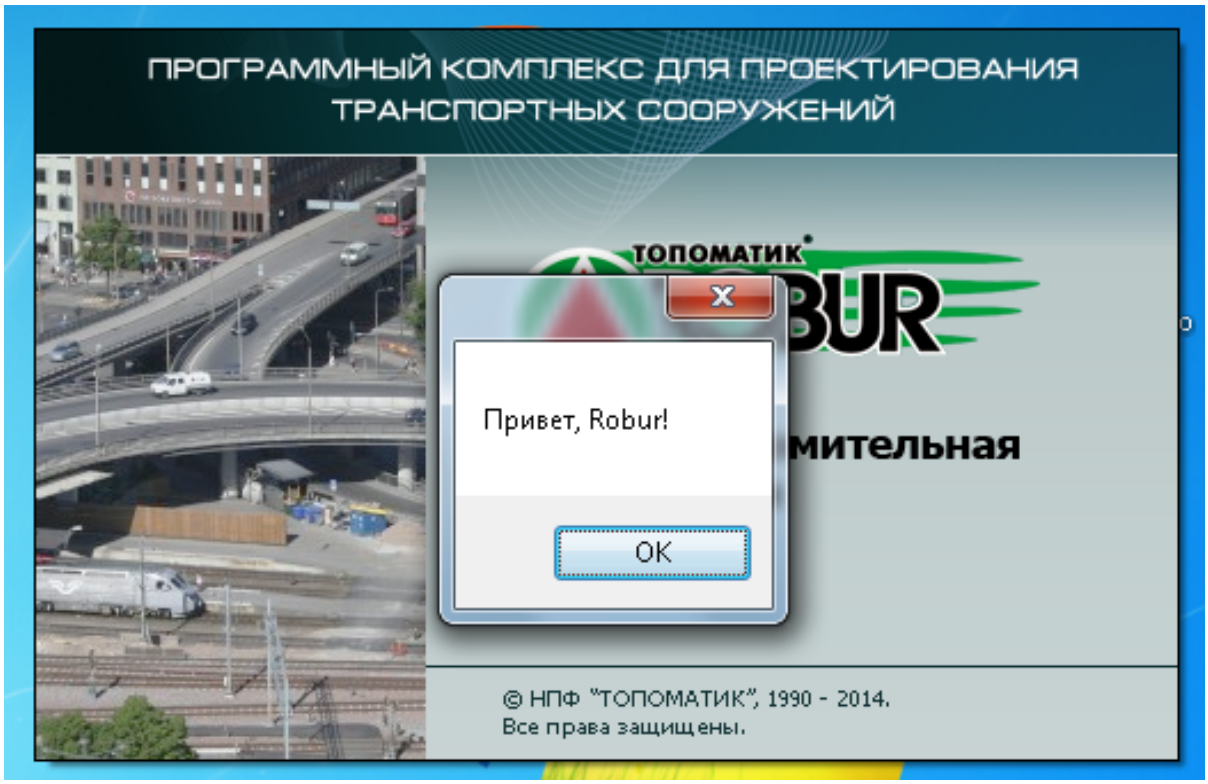


Служебная информация в этом файле определяет, для какой именно конфигурации Robur будет закгржен модуль, а также, будет ли он отображаться в списке, иметь собственную пиктограмму, как будет подписан и т.п.

Кроме того, стоит обратить внимание на номер версии, для которой предназначен этот модуль - он также жестко прописан в файле AddIn.

Теперь необходимо пересобрать еще раз проект и запускать **Robur**!

Если все было выполнено корректно, результат должен быть примерно следующим:



Часть 2. Вывод сообщения при вызове команды из командной строки или с помощью элементов управления (кнопок или пунктов меню)

В этой части примера мы модифицируем модуль из предыдущего раздела таким образом, чтобы диалоговое окно **"Привет, Robur!"** отображалось не сразу при загрузке программы, а при вызове команды mhello из командной строки **Robur**, также как это происходило *в примере 1 руководства по написанию макросов*. Как вызывать команды из командной строки рассмотрено в конце *этого раздела*.

Исходный код данного примера (проект **Visaul Studio**) расположен в папке **"Sdk\samples\Modules \Sample01a"**.

Кроме того, Вы можете добавить пункт меню или кнопку на панели управления **Robur** точно таким же образом, как и при вызове макросов. Для добавления кнопок или других элементов управления смотрите *этот раздел*.

Для того, чтобы вызвать какую - то функцию Вашего модуля из командной строки **Robur** или с помощью кнопок и пунктов меню в модуле необходимо добавить action (по-русски "экшен" - команда или действие). Фактически, **action** - это объект, имеющий некоторый уникальный идентификатор (строковый параметр) и связанную с ним функцию модуля, которая выполняется при вызове **action'а**. К одному **action'у** может быть подключено несколько кнопок и пунктов меню, если нужно, чтобы все они выполняли одну и ту же функцию.

Кроме уникального идентификатора необходимого платформе **Robur**, у **action'а** имееются и другие свойства:

- имя команды для вызова из командной строки
- его пользовательское название для меню (или для всплывающей подсказки при наведении мышки на кнопку панели инструментов)
- свойства видимости и активности
- прочие системные дополнительные атрибуты

Кроме функции вызова команды у **action'а** есть и другие полезные функции, например , которая вызывается перед функцией выполнения команды, и выполняет какие-либо проверки, а затем, разрешает или запрещает дальнейшее выполнение action'а.

С помощью нее Вы можете проверить, имеется ли в проекте активный подобъект, чертеж или поверхность, если они необходимы для выполнения команды.

Удобство в том, что одна такая функция проверки может использоваться несколькими **action'ами**. Для пользователя это также крайне удобно ввиду однотипности его действий с программой.

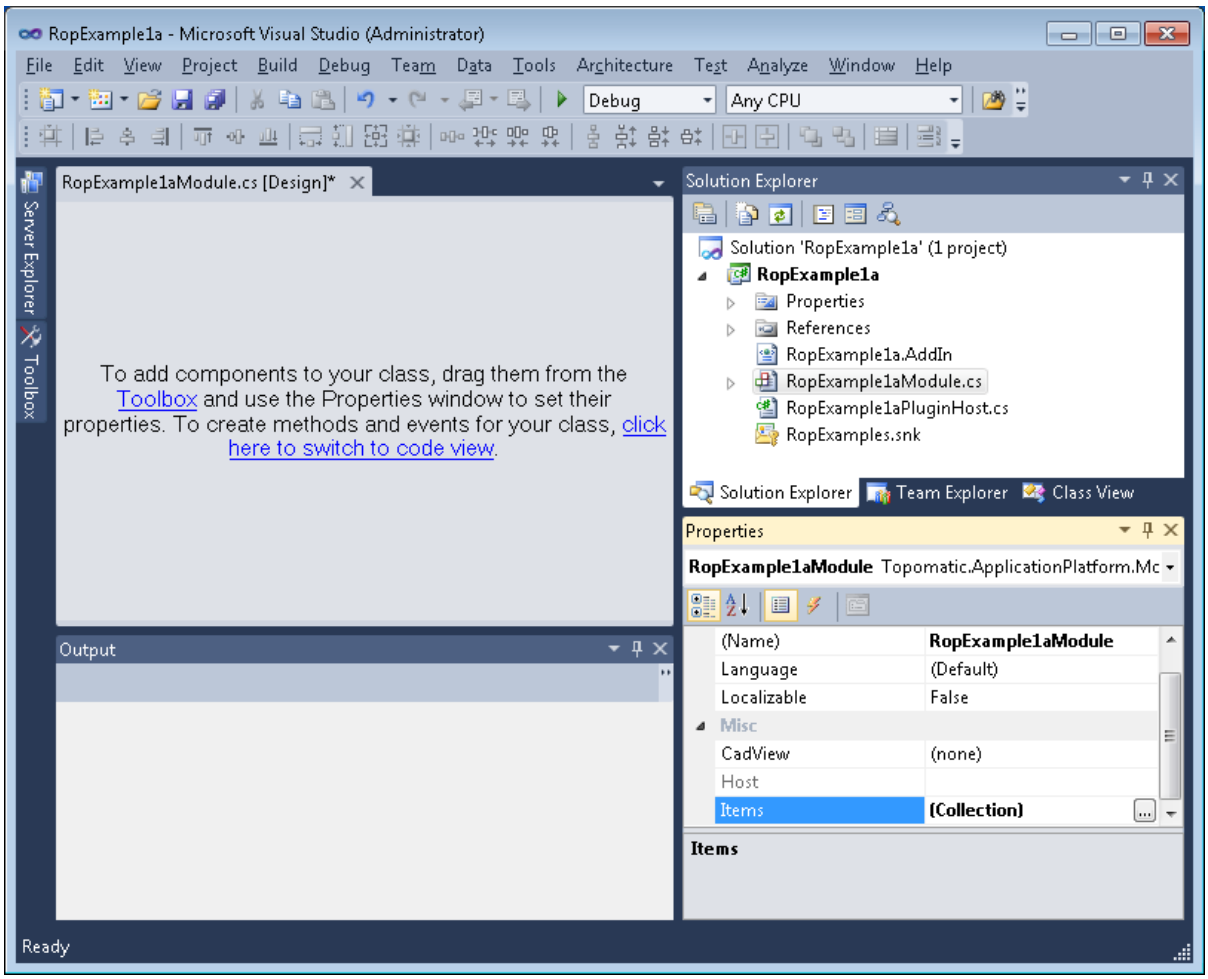
Например, все **action'ы** подсистемы работы с поверхностью в самом **Robur**, в этой функции проверяют наличие активной поверхности в структуре проекта и предлагают активировать (или при отсутствии поверхностей в проекте вообще даже создать новую) поверхность. И только если пользователь отказывается, то действия **action'ов** отменяется.

Также платформа **Топоматик АРЗ** поддерживает различные типа **action'ов**, однако их использования выходит за пределы данного руководства, так как такоие **action'ы** нужны только для специализированных задач, связанных с пользовательских интерфейсом (например создание выпадающих списков в меню или список последних открытых файлов и т.п.). В нашем руководстве под action'ом мы подразумеваем объекты класса **CallAction**.

Итак, перейдем к созданию **action'а** в нашем модуле, выводящего сообщение.

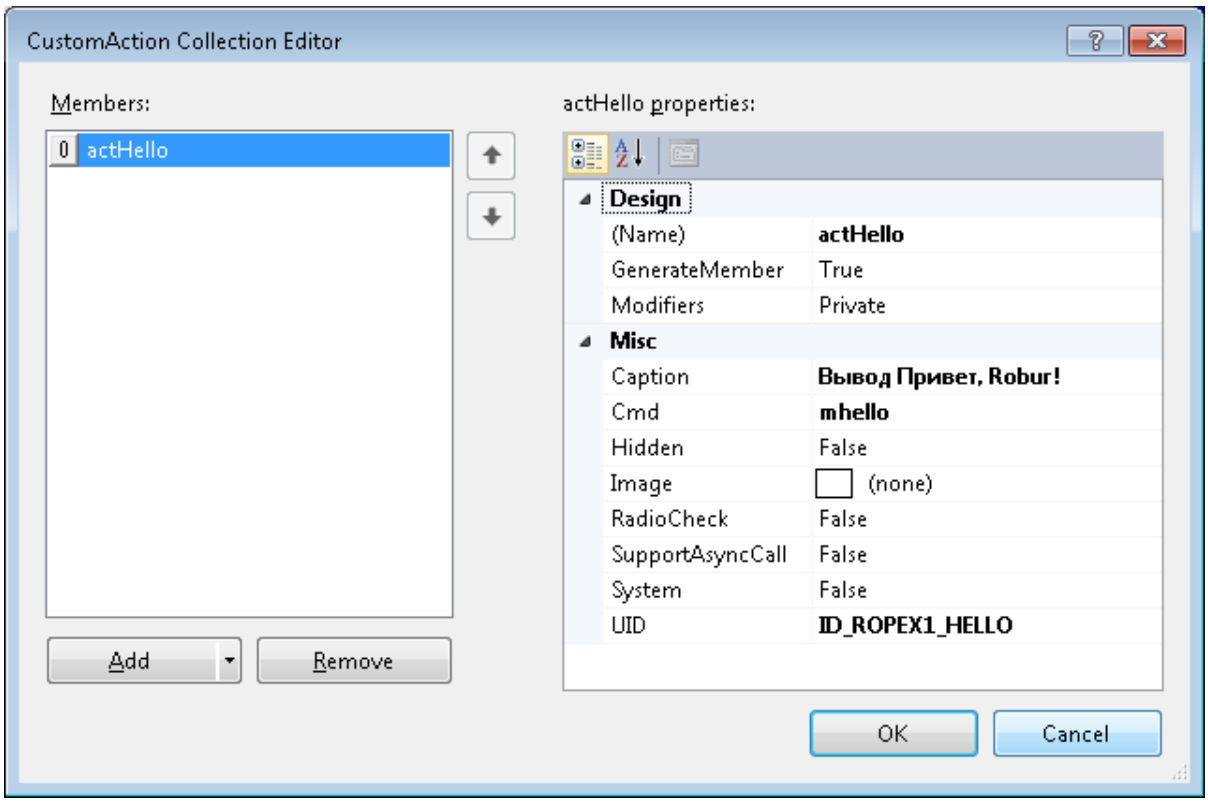
Для этого нужно открыть на редактирование **RopExample1Module** в визуальном редакторе **Visual Studio** (дважды щелкнуть на нем в **Solution Explorer**).

После того, как редактор откроется в окне свойств нужно выбрать **"Items"**

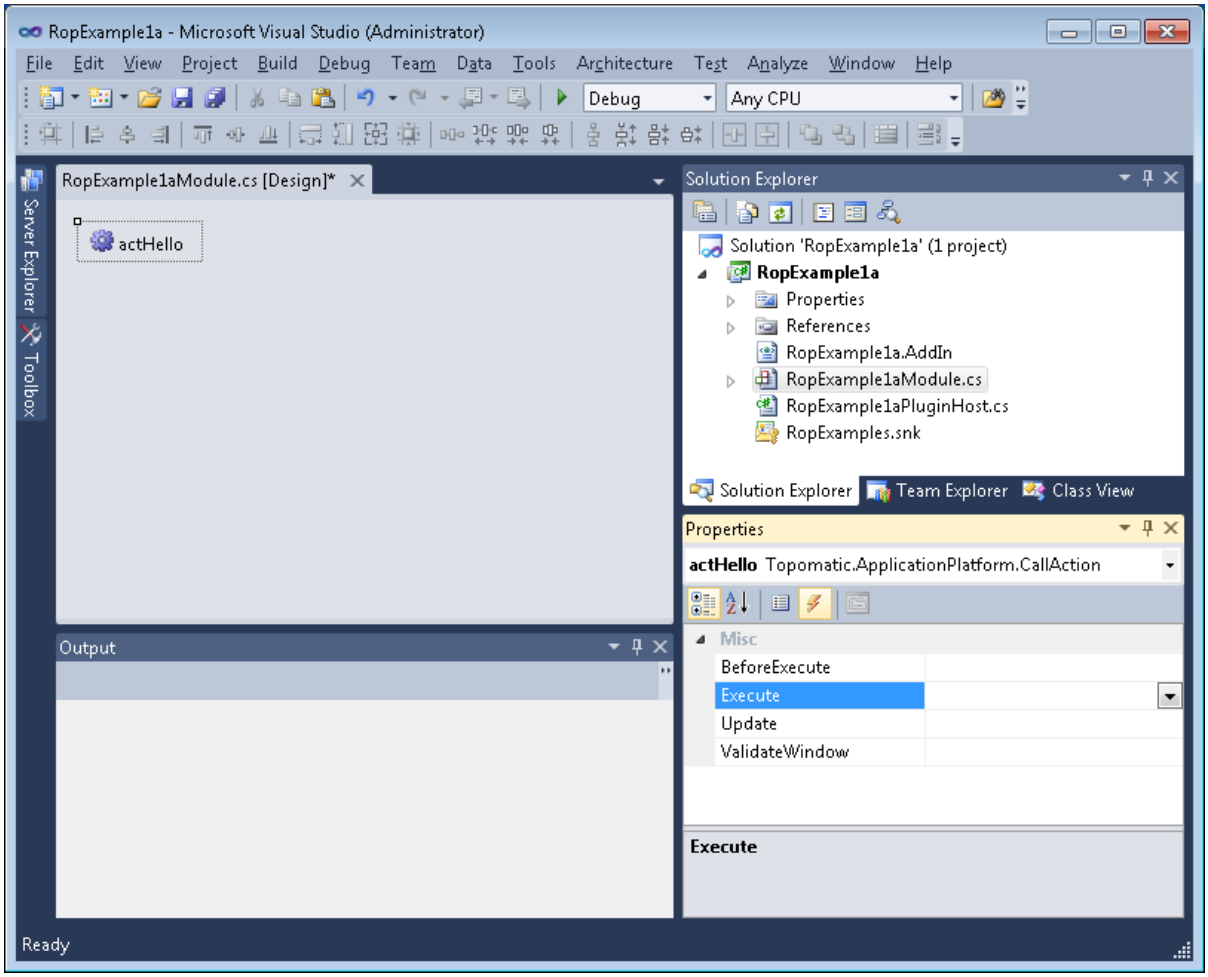


И в открывшемся окне редактирования списка **action'ов** надать слева внизу кнопку **"Add"** и выбрать тип **"CallAction"**.

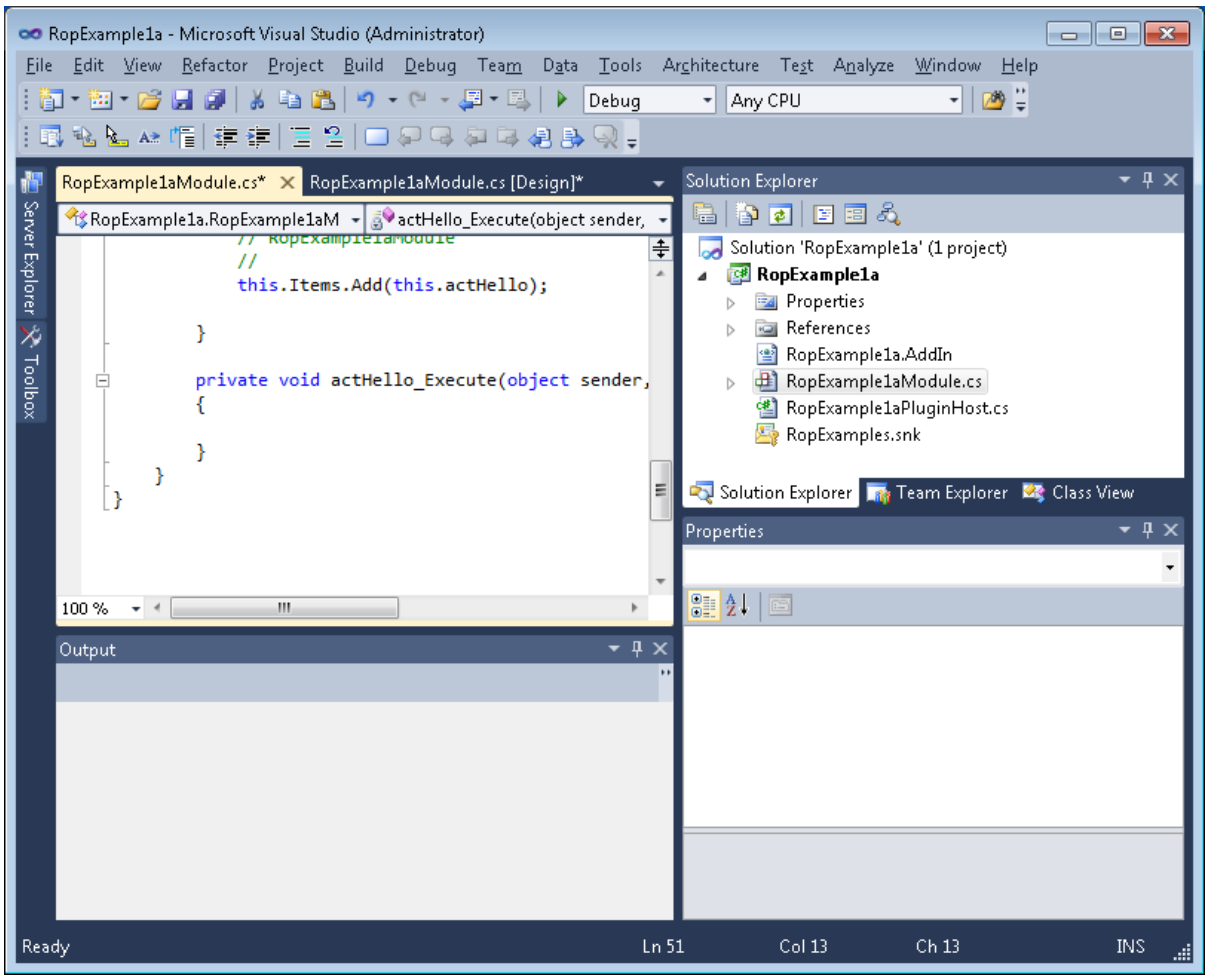
Теперь необходимо заполнить поля: **Name** (внутреннее имя **action'a**), **Caption** (текст для меню), **cmd** (Команда для командной строки) и **UID** (уникальный идентификатор **action'a**).



После задания свойств, нажмите OK, выберите action в окне дизайнера **Visual Studio**, а окно свойств переключите в режим отображения функций-обработчиков:



Дважды щелкните на строке обработчика **Execute**, при этом произойдет создание функции обработчика и откроется редактор кода



В котором надо вписать вызов окна **"Привет, Robur!"**.

```
MessageBox.Show("Привет, Robur!");
```

Добавить необходимые для сборки **using** и собрать проект. Также, если Вы модифицируете предыдущий пример, рекомендуем удалить вывод сообщения при загрузке **Robur**. А в случае отдельного примера отключить предыдущий плагин от **Robur** (например, удалив файл **".AddIn"** из папки **Robur**).

После запустите **Robur**, откройте какой-нибудь проект и вызовите в командной строке **"mhello"**.

Также можно добавить пункт меню или кнопки для вызова **action'a** также как это делалось для макроса в *примере*.

Пример **2.** Модуль**,** вставляющий примитив в текущий чертеж проекта **Robur**

Второй пример, в отличие предшественника, уже содержит определенную функциональность и использует объекты платформы **Robur** и вносит изменения в текущий проект, открытый в **Robur**.

Первое, что нужно сделать - это подключить необходимые сборки к проекту. Кроме **Topomatic.ApplicationPlatform.dll** и Topomatic.Stg.dll (которые нужны любому плагину) нам понадобится также сборки:

- подсистемы чертежей Robur - **Topomatic.Dwg.dll**
- сборка с компонентом CadView - **Topomatic.Cad.View.dll**

- сборка для работы с слоями чертежей - **Topomatic.Dwg.Layer.dll**
- сборка с описаниями базовых типов для геометрических построений - **Topomatic.Cad.Foundation.dll**
- библиотека базовых классов и контейнеров Топоматик - **Topomatic.FoundationClasses.dll**

Внимание! При подключении сборок не забывайте о флагах **"Copy Local"** и **"Specific Version"** (описано в *первом примере*).

Также, как и во втором разделе *предыдущего примера* нам необходимо добавить в наш модуль, action. Однако ему надо задать другой идентификатор (), имя пунктов меню () и имя команды для командной строки **mdrawp**.

Полный код примера прилагается в папке **"\Sdk\samples\Modules\Sample02"**.

В процедуре обработчика выполнения action'а мы получаем **CadView** активного окна в **Robur** в момент выполнения функции. Если на активном окне не оказывается **CadView**, то функция завершает свою работу:

```
var cadView = this.CadView;
if (cadView == null)
{
    return;
}
```

Следует подробно остановиться на **CadView**, редакторе видов моделей в **Robur**.

Как Вы знаете, **Robur** - это система, в основе которой лежит единая пространственная модель, а не набор каким-то образом связанных чертежей и дополнительных параметров.

Это дает колоссальные преимущества при проектировании объектов и в общем-то и является основной изюминкой **Robur**. Так как все виды (проекции) этой модели отображают одну и ту же модель, то независимо от того, в каком из видов Вы редактируете модель - меняются и все остальные. Например, Вы меняете отметки на профиле дороги, они меняются на соответствующих поперечниках. Меняете положение трассы в плане, меняются профили и поперечники, так как трасса уже проходит по другим отметкам земли и т. п.

CadView - это компонент визуального редактирования или просто отображения модели. С помощью него осуществляется редактирование плана, профиля, поперечников в соответствующих окнах, также некоторых других видов.

С помощью **CadView** и его дополнительного инструментария (курсоры, выборки, диалоговые средства, средства пересчета координат и др.) происходит отображение слоев модели, выбор объектов, задание координат и другие операции с моделью и ее составляющими.

В нашем примере мы используем **CadView** для того чтобы снять выделение со всех объектов, которые были выбраны на момент вызова макроса, а также используем курсор **CadView** для визуального задания координат и запроса радиуса.

Так как мы используем метод самой платформы, то естественно при выборе координат будут работать привязки, включенные в **Robur**, и все другие удобства, присутствующие при работе функций самого комплекса **Robur**. Это очень удобно, так как однотипное поведение одинаковых режимов работы будет понятно и ожидаемо пользователями Ваших разработок.

Следующим шагом мы получаем текущий чертеж, а точнее, сперва активный слой чертежей в **CadView**, а потом уже и сам чертеж:

```
var drawingLayer = DrawingLayer.GetDrawingLayer(cadView);
if (drawingLayer == null)
{
    return;
}
var drawing = drawingLayer.Drawing;
if (drawing == null)
{
    return;
}
```

Здесь, пользователям **AutoCAD** надо иметь ввиду что под "активным слоем чертежей" имеется ввиду не слой чертежа, на котором располагаются примитивы (как слои в **AutoCAD**), а слой модели **Robur**, на котором располагается весь чертеж целиком, так как **Robur** в одном проекте может содержать несколько чертежей. Также на слоях других типов будут располагаться и другие элементы проекта: геодезические съемки, поверхности, подобъекты и другие.

Конечно, в чертежах **Robur** примитивы также располагаются на различных слоях, как и в **AutoCAD**, только эти слои становятся дочерними, по отношению к слою всего чертежа. Кстати, другие элементы проекта также имеют подслои, например, слои пикетажа, плана в подобъектах или точек и горизонталей в поверхности. И этими слоями можно централизованно управлять, в том числе и из скриптов и модулей, разрабатываемых Вами.

Наш модуль, проверяет, удалось ли получить активный слой чертежа и сам чертеж, в противном случае, ыдается сообщение и action завершает работу. Вполне возможно, что в данный момент в **Robur** активно окно, не имеющее **CadView** или в этой проекции нет чертежей, поэтому такая ситуация вполне может произойти и выводить аварийных сообщений не стоит, также не стоит и продолжать выполнение, чтобы не произошло исключений (программных ошибок при обращении к несуществующим объектам).

Далее идет строка сброса выделения в **CadView**:

```
cadView.SelectionSet.Clear();
```

Снимать выделение рекомендуется перед каждой командой, требующей от пользователя выбора чего либо на **CadView**, чтобы не вводить его в заблуждение, так как некоторые команды могут работать с уже выбранными объектами на **CadView** или с некоторыми из них (например, удовлетворяющими каким-либо условиям).

Затем, вызывается команда, запрашивающая у пользователя координаты на экране - функция GetPoint курсора **CadView**:

```
Topomatic.Cad.Foundation.Vector3D rp;
while (Topomatic.Cad.View.Hints.CadCursors.GetPoint(cadView, out rp,
 "Укажите точку"))
```

Обратите внимание, что функция запроса взята в качестве условия срабатывания цикла while, что означает, что наша команда по вставке звезды будет повторяться до тех пор, пока пользователь не откажется от задания координат (нажмет правую кнопку мыши или Escape на клавиатуре).

Далее строки, запрашивающие радиус (по умолчанию радиус задается равным 50м):

```
double rr = 50.0;
GetPointResult gpr =
 Topomatic.Cad.View.Hints.CadCursors.GetDoubleWithDefault(cadView, ref rr,
 "Укажите радиус");
if (gpr != GetPointResult.Accept)
{
    return;
}
```

Оставшийся код - это вставка в чертеж полилинии в виде звезды, вписанной в указанный радиус. Блок освящен созданию объектов - узлов этой полилинии (вершин звезды), расчету их координат и добавление в список узлов полилинии.

Задается положение первой вершины - ее угол относительно центра звезды (точки вставки, заданной пользователем), а по углу - смещения координат:

```
var angle = 0.0;
var dx = Math.Cos(angle) * rr;
var dy = Math.Sin(angle) * rr;
```

И сами координаты точки:

```
var pos = new Topomatic.Cad.Foundation.Vector2D(rp.X + dx, rp.Y + dy);
```

Создается объект полилинии:

```
var polyline = new Topomatic.Dwg.Entities.DwgPolyline();
```

И в нее добавляется первый узел:

```
polyline.Add(new Topomatic.Cad.Foundation.BugleVector2D(pos, 0));
```

Узел полилинии в **Robur** - это не просто двухмерная или трехмерная точка (Vector2D или Vector3D), а специальный объект типа BugleVector2D, который и создается на базе точки pos.

Затем запускается цикл, в котором рассчитываются и добавляются остальные 5 вершин (последняя вершина будет совпадать с первой, так как звезда - замкнутая фигура).

```
for (var i = 0; i < 6; i++)
{
    angle += Math.PI / 1.25;
    dx = Math.Cos(angle) * rr;
    dy = Math.Sin(angle) * rr;
    pos = new Topomatic.Cad.Foundation.Vector2D(rp.X + dx, rp.Y + dy);
    polyline.Add(new Topomatic.Cad.Foundation.BugleVector2D(pos, 0));
}
```

И последним шагом полилиния добавляется в чертеж:

```
drawing.ActiveSpace.Add(polyline);
```

Наш модуль готов.

Если собрать его и создать файл **"AddIn"**, как было описано в предыдущем примере, то после закпска Robur и открытия проекта команда "mdrawp" запросит координаты и радиус и вставит в активный чертеж полилинию в виде звезды.

Если Вы внимательно просмотрели код, то наверняка заметили, что в большинстве мест мы указывали полный путь в пространстве имен к некоторым типам данных (например, `Topomatic.Cad.View.Hints.CadCursors.GetDoubleWithDefault(...)`). Это было сделано намерено, чтобы показать Вам, из каких библиотек "берутся" те или иные типы. В собственных наработках Вы, наверняка, будете использовать директиву **using**, чтобы избежать громоздкого кода.

Пример **3.** Выбор примитивов текущего чертежа и чтение их свойств

В принципе, в предыдущих примерах, Вы подучили все необходимые сведения для разработки полноценных модулей для **Robur**. Дальнейшие примеры будут только приемам работы с определенными и наиболее часто используемыми компонентами проекта (такими как чертежи, поверхности, подобъекты и др) или самой платформы.

Исчерпывающую информацию о свойствах и функциях всех этих объектов, а также о составе интересующих Вас подсистем Вы сможете найти в документации для разработчика платформы **Robur** (**Robur Platform API**). И все эти знания могут Вами использоваться независимо от того, будете ли Вы разрабатывать скрипты или полноценные модули на Вашем любимом языке программирования для .NET.

Этот пример продемонстрирует Вам как выбирать существующие примитивы чертежей и получать их свойства. Плагин попросит пользователя выбрать примитив типа "отрезок" (**DwgLine**), получит свойство этот отрезка - длину и вставит в активный чертеж примитив "текст" (**DwgText**) со значением длины примерно в центре отрезка (как подпись).

Как и в предыдущих примерах нам необходимо создать **action** в модуле и задать идентификатор (), имя пунктов меню () и имя команды для командной строки **mdraws**.

Полный код примера прилагается в папке **"\Sdk\samples\Modules\Sample03"**.

Рассмотрим код функции обработчика **action'а**:

Также как и в предыдущих примерах получаем **CadView** и слой активного чертежа для вставки подписи:

```
var cadView = this.CadView;
if (cadView == null)
{
    return;
}
var drawingLayer = DrawingLayer.GetDrawingLayer(cadView);
if (drawingLayer == null)
{
    return;
}
var drawing = drawingLayer.Drawing;
if (drawing == null)
{
    return;
}
```

Сбрасываем выбранные объекты в **CadView**. Кстати, можно и не сбрасывать выделенные объекты, а проверить, нет ли среди них отрезков, а в случае если они найдены, то не запрашивать у пользователя выбрать отрезки, а подписать их длину и выйти из макроса. Тогда, вместо **SelectionSet.Clear** и цикла с выбором примитивов, надо добавить код:

```
cadView.SelectionSet.FilterSelected(obj => obj is DwgLine);
if (cadView.SelectionSet.Count == 0)
{
    MessageBox.Show("Выбранные отрезки отсутствуют!");
    return;
}
foreach (var line in cadView.SelectionSet)
{
    ...
}
```

В прилагаемом примере код с таким поведением закомментирован.

Обратите внимание, что в случае, если среди выбранных объектов нет отрекзов, скрипт выдает соответствующее сообщение в окно командной строки и завершает работу. Также заметьте, что фильтр для выбора отрезков (`filter = lambda obj: isinstance(obj, Topomatic.Dwg.Entities.DwgLine)`) объявлен раньше, чем в основном примере и используется, как аргумент, фильтрующий список выбранных элементов, оставляя в нем только отрезки.

Чаще всего, оба метода комбинируются и, если,

А в нашем примере, далее следует код, запрашивающий у пользователя отрезок. Здесь также используется **SelectionSet**:

```
cadView.SelectionSet.Clear();
while (true)
{
    //var line = cadview.SelectionSet.PickOneObjectAtScreen((obj
 => obj is Topomatic.Dwg.Entities.DwgLine), "Выберите отрезок") as
 Topomatic.Dwg.Entities.DwgLine;
    var line = drawingLayer.SelectionSet.PickOneObjectAtScreen((obj
 => obj is Topomatic.Dwg.Entities.DwgLine), "Выберите отрезок") as
 Topomatic.Dwg.Entities.DwgLine;
```

```
    if (line == null)
    {
        return;
    }
```

Обратите внимание, что код содержит две похожих инструкции выбора отрезка, но одна закоментирована и не выполняется.

Отличие этих строк в том, что одна использует для выбора **SelectinSet** всего **CadView**, то есть пользователь может выбрать отрезок, расположенный не только в активном чертеже, а в любом из чертежей, которые отображаются в данный момент на **CadView**, тогда как наша подпись будет вставлена в активный чертеж, а не обязательно в тот, где находился отрезок.

Однако объект **SelectionSet** есть и у каждого слоя **Robur**, в том числе и у слоя активного чертежа. Вторая строка, как раз, запросит отрезок именно у слоя с активным чертежом, а другие отрезки, даже, если они отображаются будут проигнорированы.

Вы можете поменять строки, закоментировав ненужную и посмотреть на поведение скрипта.

Также обратите внимание, как задается процедура фильтра (**obj => obj is Topomatic.Dwg.Entities.DwgLine**), проверяющая, что объект является наследником от **DwgLine**.

С помощью инструкции **C# as** полученный объект приводится к типу **Topomatic.Dwg.Entities.DwgLine**.

Следующие строки посвящены получению нужных свойств выбранного отрезка и добавление текста в чертеж:

```
var text = new Topomatic.Dwg.Entities.DwgText();
text.Content = Math.Round(line.Length, 2).ToString();
text.Rotation = line.Rotation;
text.Position = new Topomatic.Cad.Foundation.Vector3D(line.StartPoint.X
 + (line.EndPoint.X - line.StartPoint.X) / 2, line.StartPoint.Y
 + (line.EndPoint.Y - line.StartPoint.Y) / 2, line.StartPoint.Z +
 (line.EndPoint.Z - line.StartPoint.Z) / 2);
drawing.ActiveSpace.Add(text);
```

Также как и в предыдущем примере работа заключена в цикл **while**, пока пользователь не отменит выбор отрезков.

Пример **4.** Управлением механизма отмены команд **Robur (Undo)** и истории команд

Следующий наш пример посвящен такому механизму **Robur**, как **Undo** или **Отмена**. Как известно все современные программы хранят журнал изменения состояния объектов и по требованию пользователя могут их отменить.

Также пользователь видит список последних изменений, где каждое изменение имеет какое - то понятное название (имя команды и т.п.).

Если Вы отобразите этот список выполненных команд (историю) в **Robur**,

Запустите предыдущий пример и подпишите за одну команду несколько отрезков (так как пример содержит цикл **while**, пока пользователь не прекратит выбирать отрезки), то Вы увидите, что история отобразит каждую подпись как отдельную команду. И каждую подпись можно по отдельности отменить.

В нашем случае ситуация не драматична, так как в примере мы сначала меняем все свойства примитива

```
var text = new Topomatic.Dwg.Entities.DwgText();
text.Content = Math.Round(line.Length, 2).ToString();
text.Rotation = line.Rotation;
text.Position = new Topomatic.Cad.Foundation.Vector3D(line.StartPoint.X
 + (line.EndPoint.X - line.StartPoint.X) / 2, line.StartPoint.Y
```

```
 + (line.EndPoint.Y - line.StartPoint.Y) / 2, line.StartPoint.Z +
 (line.EndPoint.Z - line.StartPoint.Z) / 2);
```

а уже потом добавляем его в модель

```
drawing.ActiveSpace.Add(text);
```

Однако, если поменять в одной команде несколько свойств у существующего примитива, уже имеющегося в модели, то каждое изменение будет зафиксировано в истории как отдельный шаг. Это может быть недлпустимо, так как отмена только части свойств из одной команды может нарушить целостность чертежа или всей модели.

Для таких целей в платформе **Robur** предусмотрен механизм транзакций. Транзакция представляет собой логическую группировку нескольких изменений в виде единой записи в истории, откат которой может быть произведен только полностью.

В ходе выполнения кода Вы можете сами определить, какая часть операций входит в транзакцию, а также задать ей имя (для окна истории). Кроме того, если в момент выполнения команд транзакции произошла какая-то ошибка, то можно не применять транзакцию, и все уже начатые изменения применены не будут.

Перейдем к коду примера. В качестве основы был взят предыдущий пример и дополнен необходимым кодом.

Перед началом цикла **while** запускаем транзакцию (`drawing.BeginUpdate("Добавление подписей к отрезкам");`), а после выхода, применяем ее (`drawing.EndUpdate();`).

в нашем примере код заключен в блок `try...finally`, чтобы **BeginUpdate()** и **EndUpdate()** вызвались в любом случае, даже если возникнут ошибки или выход по **return**.

Теперь, если мы запустим команду и подпишем несколько примитивов, то в окне истории все равно отобразится только одна операция.

Если бы нам было необходимо отменить транзакцию, то перед строкой `drawing.EndUpdate();` нужно вызвать

```
drawing.TransactionManager.CurrentTransaction.Undo();
```

Строго, говоря не у каждого объекта есть **TransactionManager**, позволяющий управлять транзакциями, а также не каждый объект разрешает отмену, поэтому в Ваших разработках, если операции отмены являются критичными, то нужно сначала проверять назначен ли **TransactionManager**, а уже потом начинать работать с транзакциями (вызывать **BeginUpdate**).

Также надо иметь ввиду, что если транзакция началась корректно (корректно отработал **BeginUpdate**), что у **TransactionManager** будет установлено свойство **CurrentTransaction**.

Также у **CurrentTransaction** есть булево свойство **CanUndo**, показывающее возможна ли отмена данной транзакции. Некоторые редкие операции не допускают отмены.

Благодаря этому примеру Вы научились применять механизм транзакций в своих командах и делить выполняемые изменения на логические части.

Пример **5.** Работа с поверхностями **Robur (**ЦММ**)**

В предыдущих примерах мы обращались к активному чертежу проекта: вносили новые примитивы или выбирали отрезки и читали их параметры. Следующие два примера посвящены двум другим типам элементов проекта - поверхностям (**ЦММ**) и подобъектам (изыскательским трассам, автомобильным и железным дорогам), конечно, каждому типу можно посвятить очень много примеров, так как функционал этих подсистем очень богатый. Однако, все объекты и функции подобъектов и поверхностей подробно описаны в документации **Robur API**, а наша цель лишь рассмотреть способы их использования с помощью языков программирования для **.NET** (на примере **C#**).

Первый пример посвящен поверхности. В нем мы разместим два **action'a**.

Первый запрашивает у пользователя несколько координат и вставляет в них точки поверхности, а потом спрашивает, желает ли пользователь создать структурную линию проходящую через эти точки. При положительном ответе мы создаем структурную линию.

Точки и структурные линии - это основные составляющие поверхности, поэтому научившись оперировать с ними, Вы сможете решать большинство собственных задач. Второй макрос скрипта, запрашивает у пользователя структурные линии и поднимает их отметки на 20 метров (отметка каждой точки структурной линии поднимается на 20 м.).

Рассмотрим код обработчика первого **action'a**:

Создание плагина для **Robur**, объявление используемых пространств имен и задание свойств **action'a** рассмотрены в *предыдущих примерах* и аналогичны им.

Обратите внимание! В предыдущих примерах мы пользовались функциями чертежей и их слоев, в данном примере к проекту в отличие от **Topomatic.Dwg.dll** и **Topomatic.Dwg.Layer.dll** надо подключить **Topomatic.Sfc.dll** и **Topomatic.Sfc.Layer.dll**. Однако, **Topomatic.Dwg.dll** также должна быть подключена, так как подсистема поверхностей в **Robur** для своей работы использует некоторые функции чертежа. Кроме того, для работы надо подключить и системную сборку **System.Drawing.dll** (подробно подключение сборок описано в *примере 1*).

Перейдем сразу к получению активной поверхности:

```
var cadView = this.CadView;
if (cadView == null)
{
    return;
}
var surfaceLayer = SurfaceLayer.GetSurfaceLayer(cadView);
if (surfaceLayer == null)
{
    return;
}
var surface = surfaceLayer.Surface;
if (surface == null)
{
    return;
}
```

Заметьте, что в этом примере мы также используем транзакции, чтобы все операции внутри нашего **action'a** были единой командой в окне истории и отменялись как одно целое. Подробнее транзакции были рассмотрены в *предыдущем примере*.

```
surface.BeginUpdate("Вставка точек и структурных линий");
```

Затем мы создаем структурную линию, несмотря на то, что структурную линию мы будем добавлять в модель только по согласию пользователя, мы создаем ее заранее и добавляем в нее точки, как в контейнер.

```
var line = new Topomatic.Sfc.StructureLine();
```

Затем, также как во втором примере, запрашиваем координаты для точек. При вводе координат учитывайте, что структурная линия будет проведена через эти точки в порядке их ввода.

```
Vector3D rp;
while (Topomatic.Cad.View.Hints.CadCursors.GetPoint(cadView, out rp,
 "Укажите точку"))
```

Следующие строки создают и добавляют точки в поверхность:

```
var r = surface.GetElevation(rp.Pos);
if (r == null)
```

```
{
    r = 100.0;
}
rp.Z = r.Value;
var pt = new Topomatic.Sfc.SurfacePoint(rp);
var index = new Topomatic.Sfc.PointEditor(surface).Add(pt);
```

И сразу же сохраняют номера (индексы) точек в качестве узлов в структурную линию:

```
line.Add(index);
```

Отметка каждой рассчитывается по **ЦММ** с помощью функции поверхности **GetElevation**, если возможно (если в указанных пользователем координатах есть поверхность), в противном случае ставится 100м.

```
var r = surface.GetElevation(rp.Pos);
if (r == null)
{
    r = 100.0;
}
rp.Z = r.Value;
```

Далее запрашивем, нужно ли создавать структурную линию по введенным точкам:

```
bool b = false;
if (!Topomatic.Cad.View.Hints.CadCursors.GetBoolean(cadView, ref b, null,
 "Создать структурную линию?:"))
{
    return;
}
if (!b)
{
    return;
}
```

Если пользователь отменяет выбор или отвечает отрицательно, то завершаем транзакцию и выходим. Также, в таких случаях, чтобы трижды не писать **EndUpdate()**, лучше использовать блок **try .. finally ..** . Если Вы имеете опыт программирования, то данный прием должен быть Вам знаком.

Если пользователь согласен, то добавляем структурную линию в модель, ну и конечно, **EndUpdate()** в блоке **try..finally**:

```
surface.StructureLines.Add(line);
```

Первый **action** завершен. Если выполнить его, то мы получим описанный выше результат.

Второй **action** запрашивает у пользователя выбрать структурную линию.

Также как и первый он получает активную **ЦММ** проекта, а затем с помощью функции-предиката (фильтра) выполняется запрос структурной линии:

```
var line = surfaceLayer.SelectionSet.PickOneObjectAtScreen((obj => obj is
 StructureLine), "Выберите структурную линию") as StructureLine;
if (line == null)
{
    return;
}
```

Изначально у **SelectionSet** есть метод выбора объекта, который позволяет выбирать любой объект на **CadView**, но в качестве параметра ему можно передать функцию-предикат, уточняющую выбор, необходимый Вам. В нашем случае эта функция очень простая - `(obj => obj is StructureLine)`, она просто разрешает выбор, если объект является структурной линией.

Однако в своих разработках Вы можете проверять и очень сложные условия, зависящие от многих факторов. Главное описать в предикате подходит ли передаваемый ему объект и вернуть соответствующий результат.

Полученный объект сразу приводится к типу **StructureLine** (с помощью **as**).

Если структурная линия выбрана, то проходим по всем ее точкам и увеличиваем их отметку.

```
for (int i = 0; i < line.Count; i++)
{
    var node = line[i];
    var newNode = new Topomatic.Sfc.StructureLineNode(node.Index,
 node.Elevation + 20);
    line[i] = newNode;
}
```

После подключения скрипта к **Robur** попробуйте вызвать обе команды, чтобы посмотреть результат.

Пример **6.** Работа с подобъектами **Robur**

Этот пример рассматривает работу с подобъектами, в нем мы определим пикетажное положение указанной пользователем точки относительно текущего подобъекта и вставим в текущий чертеж соответствующую подпись.

Пожалуйста, посмотрите прилагающийся к описанию пример 6 - раздел References (ссылки на используемые сборки и библиотеки). При реализации Вашего собственного варианта необходимо подключить их же.

Рассмотрим код обработчика action'a, добавляющего подписи.

Сначала, как в предыдущих примерах получаем текущий чертеж:

```
var cadView = this.CadView;
if (cadView == null)
{
    return;
}
var drawingLayer = DrawingLayer.GetDrawingLayer(cadView);
if (drawingLayer == null)
{
    return;
}
var drawing = drawingLayer.Drawing;
if (drawing == null)
{
    return;
}
```

Затем, получаем текущий подобъект

```
var model =
 Topomatic.Alg.Runtime.ServiceClasses.ModelWatcher.Current.ActiveModel as
 AlignmentModel;
if (model == null)
{
    return;
}
var alignment = model.Alignment;
```

Подобъект - одна из самых основных частей проекта **Robur**. Именно подобъект реализует, как общий функционал пространственной кривой - трассы: план трассы, профили (черные и проектные), струны смещений профилей, пикетаж, и поперечники... Также подобъект реализует и частные задачи автомобильных

и железных дорог или геодезических трасс (расчеты верха земполотна автодорог, конструирование поперечных профилей железных дорог и т.п.).

Если Вы обладаете навыками программирования для **Microsoft .NET**, то настоятельно рекомендуем Вам ознакомиться с составом сборок **Topomatic.Alg.dll**, начиная с класса **Alignment** и его функций и свойств.

Специфические функции железнодорожных подобъектов находятся в сборке **Topomatic.Alg.Rail.dll**, а автодорожных - в **Topomatic.Alg.Road.dll**.

Также основные функции подобъектов и их параметры подробно рассмотрены в руководстве **API** платформы **Топоматик АР3**.

Затем, как в примере 2, в цикле запрашиваем у пользователя координаты точек:

```
while (Topomatic.Cad.View.Hints.CadCursors.GetPoint(cadView, out rp,
 "Укажите точку"))
```

Далее мы обращаемся к плану подобъекта и вычисляем смещение в метрах относительно начала трассы для указанной точки с помошью функции **PosToStaOffs** (эта функция также определяет смещение по касательной в сторону от подобъекта).

PosToStaOffset - это основная функция при работе с планом линейного объекта, так как 90% задач тем или иным способом сводятся к расчету координат точки по смещению от началу трассы (sta) и смещению от оси (offset) и наоборот. Обратная функция называется **StaOffsetToPos**.

Блок кода, использующий функцию и добавляющий примитив (текст **DwgText**) с подписью пикета:

```
if (alignment.Plan.CompoundLine.PosToStaOffset(rp.Pos, out sta, out offs))
{
    var text = new Topomatic.Dwg.Entities.DwgText();
    text.Content = alignment.Stationing.StationToString(sta);
    text.Position = rp;
    drawing.ActiveSpace.Add(text);
}
```

Обратите внимание!

Для получения имени пикета мы используем свойство подобъекта **Stationing** - реализующее таблицу пикетажа и все его функции. Для получения текста с именем пикета из смещения от начала трассы в метрах мы вызываем его функцию **StationToString.**

Для получения сведений об орстальных свойствах пожалуйста обратитесь к руководству **API** платформы **Топоматик АР3**.