Урок 3: Первый взгляд на код

На этом занятии будет продолжено изучение Autodesk Revit API. Мы еще более детально проанализируем код C#, который был введен на первом уроке, разобравшись с каждой строкой.

Выполнив анализ кода и ознакомившись с дополнительными темами, вы подойдете ближе к пониманию понятий объектно-ориентированного программирования и классов.

Обратная связь: напишите нам об этом уроке или обо всем курсе «Моя первая программа»: myfirstplugin@autodesk.com
(Пожалуйста, пишите на английском языке)

Анализ кода из Урока 1

Ключевое слово

При первом взгляде на код в глаза бросается его расцветка: для упрощения чтения кода Visual C# Express изменяет цвета основных слов в нем. Слова, подсвеченные синим, называются ключевыми словами и могут быть использованы лишь в определенных случаях. Вы не сможете, например, объявить переменную с именем **using**, так как оно зарезервировано для других целей.

```
using System;
using System.Collections.Generic;
using System.Linq;

using Autodesk.Revit.DB;
using Autodesk.Revit.DB.Architecture;
using Autodesk.Revit.UI;
using Autodesk.Revit.UI.Selection;
using Autodesk.Revit.ApplicationServices;
using Autodesk.Revit.Attributes;
```

Пространства имен

Первые строки кода начинаются с ключевого слова **using**. Если при работе с различными сборками проектов C# довольно часто задавать одинаковые имена классам, это может привести к тому, что в проекте окажется несколько версий одного и того же класса для экземпляра в разных библиотеках. Для того чтобы избежать конфликтов имен, в .NET предусмотрена концепция пространства имен. Пространство имен позволяет организовать классы в логические группы, тем самым помогая указать конкретный класс, который вы хотите использовать.

Для наглядного примера давайте предположим, что по соседству друг с другом живут двое людей по имени Иван. Одним из наиболее явных способов указать нужного вам Ивана будет использование его фамилии. Но если у них также совпадают и фамилии (например Кузнецов), этого будет недостаточно для их идентификации. Нужен еще какой-нибудь дополнительный признак, например школа, в которой они учились. Если бы такой подход был заложен в синтаксис .NET, для идентификации вы бы использовали: ШколаА.Кузнецов.Иван и Школа Б.Кузнецов.Иван для каждого из них.

Именно таким способом определяются нужные классы в C#. Ключевое слово using дает вам прямой доступ к классам, находящимся в пространстве имен, что позволяет указывать необходимый класс, не вводя полностью пространство имен вместе с именем класса. Это также позволяет собрать все классы из разных пространств имен в один список по технологии IntelliSense.

Ключевое слово using – это директива, которая указывает, что компилятор C# должен предоставлять прямой доступ ко всему содержимому в указанном пространстве имен для кода в данном файле проекта.

Атрибуты

Давайте взглянем на данный фрагмент кода:

```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
}
```

Для начала рассмотрим первые две строчки кода в квадратных скобках:

```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
```

Эти два атрибута позволяют управлять поведением транзакций и регенерацией модели.

1. Атрибут Transaction. Данный атрибут описывает то, как должны работать транзакции в вашем проекте. Можно указать автоматический (Automatic) и ручной (Manual) режимы. В данном примере установлен ручной (Manual) режим.

   Транзакция – это объект, который перехватывает изменения в модели Revit. Любые изменения модели Revit происходят только при активной транзакции. Транзакции могут быть приняты (то есть изменения применяются и записываются в модель) или откатаны (изменения отменяются).

   Значение TransactionMode.Manual атрибута TransactionAttribute указывает, что Revit не будет автоматически создавать транзакции (кроме внешних команд, для которых Revit всегда создает транзакции, чтобы была возможность отменить действия в случае ошибки). Ручной режим предполагает, что вы сами осуществляете создание, назначение имен, принятие и отмену транзакций.

2. Атрибут Regeneration. **Этот атрибут появляется, начиная с Revit 2011**. Атрибут регенерации указывает время, когда Revit перерисовывает объекты на мониторе компьютера. Он также может принимать значения Manual и Automatic. В данном примере установлено значение Manual.

RegenerationOption.Manual означает, что Revit не перестраивает (и не обновляет) модель на экране при каждом действии с ней, до тех пор пока вы сами не укажете, что модель должна быть перерисована. Если вы укажете значение Automatic, Revit будет перестраивать модель при каждом изменении.

IExternalCommand

После того как вы указали атрибуты, следующей строкой объявляется класс:

```
public class Lab1PlaceGroup : IExternalCommand
{

}
```

Класс можно считать своего рода элементом кода, который описывает объект. Класс подобен шаблону, в котором описаны детали объекта; этот шаблон используется для создания экземпляров объекта. Классы позволяют определять такие характеристики объекта, как атрибуты (свойства) и его поведение (методы). Для более подробной информации обратитесь к дополнительным темам.

Ключевое слово **public** говорит о том, что данный класс будет доступен не только из других классов в данном плагине, но и в других проектах Visual Studio, которые будут ссылаться на данную dll-библиотеку. Имя **Lab1PlaceGroup** – это имя вашего класса.

Следующим после указания имени класса и двоеточия идет слово **IExternalCommand**. Размещение двоеточия после имени класса и слова IExternalCommand говорит компилятору C#, что вы хотите, чтобы ваш класс наследовал интерфейс IExternalCommand (полное имя у которого, кстати, – Autodesk.Revit.UI.IExternalCommand).

С точки зрения плагина для Revit это означает, что класс представляет собой внешнюю команду. Revit сможет использовать данный класс для исполнения команды, когда получит сигнал от пользовательского интерфейса. Проще говоря, объявив класс, наследующий интерфейс IExternalCommand, мы указываем Revit ,что это команда. Интерфейсы подобны классам, за исключением того, что их методы не реализованы. Имена интерфейсов начинаются с символа «I», как в случае с IExternalCommand.

Как только вы в новом классе Lab1PlaceGroup наследовали интерфейс IExternalCommand из Revit API, требуется реализовать все нереализованные методы этого интерфейса. Возникает естественный вопрос: если я наследовал интерфейс, то откуда мне знать, какие методы я должен реализовать?

К счастью, в Visual C# Express имеется возможность создать базовый скелет для реализации всех методов. Если вы щелкнете правой кнопкой мыши на слове IExternalCommand в окне редактора кода Visual C# Express, откроется контекстное меню, содержащее элемент **Реализовать интерфейс (Implement Interface)**. Развернув его, вы увидите пункт **Явно реализовать интерфейс (Implement Interface Explicitly)**. При его выборе будут созданы пустые методы, которые вы должны заполнить.

Класс плагина Revit, который реализует этот интерфейс, также именуется *входная точка* для данного плагина; это класс, в котором Revit предполагает вызвать метод **Execute()**. Другими словами, когда пользователь Revit вызывает команду через пользовательский интерфейс Revit в меню кнопки **Внешние инструменты (External Tools)** на вкладке **Надстройки** (**Add-Ins)**, код в методе Execute() будет запущен (выполнен) из соответствующего класса, реализовавшего интерфейс IExternalCommand .

```
[TransactionAttribute(TransactionMode.Manual)]
[RegenerationAttribute(RegenerationOption.Manual)]
public class Lab1PlaceGroup : IExternalCommand
{
  public Result Execute(
    ExternalCommandData commandData,
    ref string message,
    ElementSet elements)
  {

  }
}
```

Любой блок кода класса, который выполняет какие-либо задачи (или действия), называется *методом (method)*. В данном случае объявление метода начинается с ключевого слова **public**. Вы уже знаете, что означает это слово. За более подробной информацией обращайтесь в раздел «Дополнительные темы».

Данный метод возвращает **Результат (Result)** (полное имя Autodesk.Revit.UI.Result) в отличии от случаев, если он был бы объявлен как void (то есть ничего не возвращает). Результат, который возвращается из метода Execute(), передает Revit информацию о том, как прошло выполнение команды: успешно, с ошибкой или отменено. Если выполнение команды отличается от успешного, все изменения будут отменены (т.е. Revit откатит все транзакции, связанные с этой командой).

Метод Execute() содержит три параметра: **commandData**, **message** и **elements**. Давайте посмотрим поближе, что эти параметры означают:

1. **commandData** – это параметр типа **ExternalCommandData**, дающий вам доступ к API приложения Revit. Данный объект предоставляет доступ к активным документам в пользовательском интерфейсе и его базе данных. Все данные Revit, включая модель, доступны через параметр **commandData**.
2. **message** – это **строковый** (**string)** параметр с дополнительным ключевым словом **ref**, которое означает, что данный параметр может быть изменен в процессе реализации метода. Данный параметр будет изменен во внешней команде, когда она будет отменена или приведет к ошибке. Если этот параметр будет изменен и метод Execute() вернет ошибку или отмену в качестве результата, то в диалоговом окне ошибки Revit будет отображаться текст данного параметра.
3. **elements** – это параметр типа **ElementSet**, который позволяет указать элементы, которые будут подсвечены, если внешняя команда вернет ошибку или отмену.

Давайте теперь взглянем на код метода Execute(). Это как раз и есть тот самый набор инструкций Revit API, который содержит задачи для выполнения команды. На занятии 1 вы видели, как Revit API может быть использован для добавления внешней команды, которая просит пользователя выбрать группу и указать целевую точку, куда данная группа будет скопирована.

Давайте проанализируем код построчно:

```
    //Получение объектов приложения и документа
    UIApplication uiApp = commandData.Application;
```

В первой строчке мы используем параметр commandData, который был передан методом Execute() для доступа к его свойству **Application,** предоставляющему доступ к приложению Revit. Для более подробной информации о свойствах и обзора основных классов Revit API и их взаимосвязи, смотрите раздел Дополнительные темы.

Переменные

Для того чтобы использовать свойство Application, которое вы только что получили из параметра **commandData**, вы создали переменную **uiApp** типа **UIApplication**. Затем вы присвоили значение **commandData.Application** для дальнейшего использования в программе. Объявляя переменную, вы создаете именованную ссылку на значение для дальнейшего доступа к этим данным уже без использования данного блока кода. Переменные могут называться так, как вам захочется, но они должны быть уникальны в пределах данного блока кода и не совпадать с ключевыми словами (например с такими как “using”).

```
    Document doc = uiApp.ActiveUIDocument.Document;
```

Переменная uiApp (которая содержит объект Revit Application) дает нам доступ к активному документу пользовательского интерфейса Revit через свойство ActiveUIDocument. В предыдущей строке кода (всего одной!) вы получаете доступ ко всей базе данных активного документа (эта база представлена классом Document). Вы присваиваете объект Document переменной с именем **doc**.

Выбор объектов

Давайте посмотрим, как предоставить пользователю возможность выбрать группы, используя API.

```
    //Определение объекта-ссылки для занесения результата указания
    Reference pickedRef = null;
```

Начинается все с создания пустой переменной **pickedRef** типа **Reference**. Ей присваивается значение **null** (что, по сути, значит «ничего»). Таким образом, вы создаете пустой контейнер, который потенциально может содержать объект типа Reference. Reference – это класс, который может содержать элементы модели Revit, связанные с определенной геометрией.

```
    //Указание группы
    Selection sel = uiApp.ActiveUIDocument.Selection;
    pickedRef = sel.PickObject(ObjectType.Element,
      "Выберите группу");
    Element elem = pickedRef.Element;
    Group group = elem as Group;
```

Далее вы получаете доступ к выбору элементов через API. Выбранные пользователем объекты представлены свойством **Selection** в объекте **ActiveUIDocument**: мы присваиваем объект Selection переменной **sel** типа Selection. Объект Selection предоставляет вам метод **PickObject()**. При выполнении этого метода фокус переходит к пользовательскому интерфейсу, предлагая пользователю выбрать объект. Параметры данного метода позволяют вам указать, какие типы элементов необходимо выбрать пользователю (плоскость, элемент, грань и т.д.); здесь также вводится текст запроса, который отображается в левой части строки состояния Revit, пока плагин ждет завершения операции выбора.

Выбранный объект **Group** содержит в себе геометрию? которая ассоциирована с ним. Она была сохранена в ранее объявленную переменную refPicked. Затем вы использовали свойство **Element** для получения доступа к элементу, связанному с данной геометрией: в данном примере вы присваиваете этот элемент переменной **elem**, которая имеет тип Element. Как и ожидалось, данный объект будет типа **Group**, и строкой ниже мы присваиваем элемент как тип **Group** переменной group.

Примечание: если вы работаете с Revit 2012 API, следующая строка даст предупреждение, что 'Autodesk.Revit.DB.Reference.Element' является устаревшим.

```
    Element elem = pickedRef.Element;
```

Чтобы данное сообщение не появлялось в версии Revit 2012, замените его на следующую строку:

```
    Element elem = doc.GetElement(pickedRef);
```

Примечание: Если вы работаете с Revit 2013 API, вам также будет необходимо проделать указанные выше действия, для построения и успешного запуска программного кода.

Создание

В промышленном производстве термин «создание» можно трактовать как процесс придания нужной формы заготовке. В программировании процесс создания связан с попыткой передать значение одного элемента другому. Делая запросы, компилятор языка рассматривает значение разными способами. В процессе создания компилятор просит предоставить результат в другом виде. В последней строке вы передаете значение типа **Element** (который, по сути, является группой, выбранной пользователем), но при этом указываете, что оно должно преобразоваться в тип **Group**. Оператор **as** в C# заставляет компилятор проверить тип объекта, который будет создаваться: если он будет несовместим с требуемым типом, оператор вернет null.

Затем вы просите пользователя выбрать целевую точку, в которую будет скопирована группа.

```
    //Указание точки
    XYZ point = sel.PickPoint("Укажите точку для размещения группы");
```

Для этого вы опять воспользовались переменной **sel** типа **Selection** – как и раньше, она дала доступ к методу PickObject(), а также к методу **PickPoint**. Метод **PickPoint()** имеет всего один параметр: сообщение, которое будет отображаться в левом части строки состояния, пока плагин ждет выбора точки пользователем. После того как будет выбрана точка, метод вернет объект типа **XYZ**, который будет присвоен переменной **point**.

```
    //Размещение группы
    Transaction trans = new Transaction(doc);
    trans.Start("Lab");
    doc.Create.PlaceGroup(point, group.GroupType);
    trans.Commit();
```

Мы уже упоминали о транзакциях в контексте Revit API. Был установлен параметр транзакций *manual*, и Revit ожидает от вашего плагина, что он сам будет управлять объектами типа **Transaction** для внесения изменений в модель. Для создания новой группы требуется создать свою транзакцию. Вы начинаете с объявления переменной **trans**, которой вы назначите экземпляр классаTransaction, создав его с помощью ключевого слова. **Constructor(Конструктор)** (особый метод для создания экземпляра класса) класса Transaction в качестве

параметра ждет класс с базой данных активного документа, чтобы транзакция знала, с каким документом она связана. Затем вам нужно запустить транзакцию с помощью метода **Start()**, позволив внести изменения в модель Revit.

Целью нашего плагина является размещение копии выбранной группы в выбранной точке. Для этого вам понадобится метод **PlaceGroup()**, который находится в базе данных документа среди методов для создания объектов в свойстве **Create**. Свойство Create дает возможность создавать новые экземпляры элементов в модели Revit, такие как группы. Метод PlaceGroup(), как и ожидалось, требует указать место, где будет размещена группа и тип этой группы (в контексте Revit, а не C#).

Наконец, вы подтверждаете транзакцию с помощью метода **Commit()**. Данный метод подтверждает изменения в модели Revit, выполненные данной транзакцией.

```
    return Result.Succeeded;
```

Как вы помните, метод Execute(), входная точка плагина Revit, ждет возврата значения **Result**. Это значение говорит Revit, была ли успешно выполнена команда, или она отменена или прошла с ошибкой. В данном случае все прошло успешно, и в качестве результата вы возвращаете значение **Succeeded**.

Отлично! Вы прошли занятие и получили много новой информации.

На данном занятии вы охватили новые темы, в том числе основы объектно-ориентированного программирования, включая такие фундаментальные понятия языка C#, как классы и объекты, а также изучили часть среды разработки дополнений для Revit. Вам были представлены основные классы Revit API и проанализирован код, который вы создали на занятии 1. Наконец, вы просмотрели различные классы и методы Revit API, которые позволяют вам взаимодействовать с тем, что выбирает пользователь и которые помогают создать экземпляры групп.

Дополнительные темы

Объектно-ориентированное программирование
На предыдущем занятии вы узнали, как запрограммировать набор инструкций, которые говорят компьютеру, что он должен сделать. Простые программы состоят из последовательности инструкций, которые оперируют переменными – в этом случае легко понять, что произойдет в программе при ее исполнении. Но с увеличением сложности такие линейные инструкции (такой подход называется процедурным программированием) трудно применимы для более комплексных задач.

Чтобы решить данную проблему, был разработан другой подход, известный как объектно-ориентированное программирование (ООП), который позволяет по-другому структурировать код, создавая компоненты для повторного использования. ООП создает масштабируемые повторно используемые объекты, логически связанные между собой и нацеленные на создание объектов и взаимодействий между ними. Хороший пример данной схемы – это карта. Карта города представляет собой упрощенную модель, которая вам нужна для ориентирования в нем.

Ключом к качественному проектированию дизайна объектно-ориентированной модели является создание индивидуальных элементов и взаимосвязей между ними.

Классы

Класс можно воспринимать как *тип*, который используется для представления уникальности объекта. Класс можно считать своего рода шаблоном, который описывает элементы объекта для создания экземпляров. Класс может помочь описать основные характеристики объекта: атрибуты (*свойства*) и поведение (*методы*) объекта.

Объект – это экземпляр класса. Эти основные элементы объектно-ориентированного программирования могут выступать в качестве переменных, которые сосредотачивают в себе как данные, так и поведение.

Чтобы более наглядно описать взаимодействие классов и объектов, достаточно вспомнить о взаимодействии семейства и экземпляра семейства в Revit. Вы можете представить себе системное семейство стены как класс. Это семейство, в котором описаны базовые параметры стены. Когда вы создаете экземпляр данного системного семейства стены в модели, вы создаете объект.
Семейство стены – своего рода план или шаблон для всех стен в модели здания. Каждая стена содержит в себе одинаковые параметры в соответствии с этим шаблоном, но при этом имеет различные значения: для каждого экземпляра они могут иметь различную огнестойкость или расположение.

Как определяется класс в C#?

Код ниже содержит пример объявления простого класса в C#. Во время описания мы также поближе познакомимся с синтаксисом языка C#.

```
public class Point
{
  private int x, y;

  public void setLocation(int x, int y)
  {
    this.x = x;
    this.y = y;

    // Далее выполняются некоторые вычисления
    //
  }
}
```

Первое слово **public** задает степень доступности класса. Использование ключевого слова public означает, что любой разработчик имеет доступ к данному классу и может его использовать. Другие возможные варианты – **internal** (доступно только в данном проекте) или **private** (используется только в пределах данного класса).

Следующее слово **class** указывает на объявление класса. Слово **Point** определяет имя класса.

Затем идет символ “**{**“. Каждая открывающая фигурная скобка (**{**) идет в паре с закрывающей (**}**). Компилятор в любом случае воспримет отсутствие пары для скобки как ошибку (это будет воспринято как код, находящийся в неположенном месте). Фигурные скобки позволяют создавать блоки кода. В данном случае открывающие и закрывающие скобки обозначают начало и конец класса.

Объявление переменных

Далее объявляются две *переменные*: Эти переменные будут использоваться в классе и помогут обмениваться данными внутри блоков класса . Вы объявляете данные в одном месте и затем можете их использовать в любых других местах класса.

```
  private int x, y;
```

Строки начинаются с ключевого слова **private,** которое определяет доступ к данным. В данном случае данные доступны только в пределах данного класса.

Ключевое слово **int** определяет тип «целочисленное», тем самым сообщая компилятору что вы хотите зарезервировать место для хранения целочисленных данных. Наиболее часто используемые типы в C# – это числа с плавающей точкой, целочисленные и текстовые (строковые данные). **x** и **y** – это имена которые помогают идентифицировать данные.

Код C# состоит из операторов, каждый из которых завершается точкой с запятой (;)

Методы

Данный блок кода называется *методом*:

```
  public void setLocation(int x, int y)
  {
    this.x = x;
    this.y = y;

    // Далее выполняются некоторые вычисления
    //
  }
```

Любой блок кода, который предполагает какую-либо задачу (или действие), называется методом. Объявление метода начинается с ключевого слова **public** . Вы уже знаете, что это означает.

Следующее слово – **void**. Слово на этом месте указывает, какие данные возвращает метод после исполнения задач внутри себя. Например, если метод выполняет какие-то вычисления, будет возвращен их результат. В данном случае вы указываете “void”; это означает, что результат не будет возвращаться. То есть при вызове данного метода вы не сможете присвоить его результат какой-либо переменной.

Слово **setLocation** – это имя метода для идентификации и его использования.

Текст в скобках содержит *параметры* методов. Параметр (или набор параметров) – это входящие данные, которыми метод может оперировать. Параметры так же могут быть использованы для возвращения данных к вызывающему данный метод коду.

Следующей строкой у нас идет открывающая фигурная скобка, которая свидетельствует о начале тела блока объявленного метода. В теле метода вы видите эти строки:

```
    this.x = x;
    this.y = y;
```

Ключевое слово **this** указывает на данный экземпляр вашего класса. В данной строке вы получаете доступ к переменной **x** данного экземпляра используя конструкцию **this.x**. знак “**=**” называется *оператором присваивания* и означает, что контейнер с левой стороны оператора получает значение, указанное с правой стороны. В данном случае вы присваиваете значение переменной x текущего экземпляра класса Point (используя ключевое слово this) значению параметра x, переданного в метод.

Следующая строка использует то же решение, но уже для переменной **y**.

Строки, начинающиеся с “//”, не учитываются в C#. Данным способом можно добавлять комментарии в ваш код, чтобы описать его доступным языком или временно закомментировать код, который должен быть проигнорирован.

И наконец, закрывающая фигурная скобка означает конец блока.

Если вы посмотрите дальше по коду, то увидите еще несколько закрывающих фигурных скобок. Они указывают на конец блока класса (для лучшего восприятия они имеют другой отступ).

Теперь у нас есть простой класс, объявленный с использовением C#. Следующий вопрос вполне естественный…

Как вы будете создавать объекты данного класса?

Создание объекта класса

Как уже говорилось ранее, объект – это экземпляр класса. Код ниже показывает, как создавать объект типового класса. Эти строки должны содержать реализацию класса (почти весь код программы на C# содержится в классе – вы увидите позже, как код данного класса вызывается).

```
    Point pt = new Point();
    pt.setLocation(10, 10);
```

В первой строке вы объявляете новую переменную **pt**, которая соответствует типу **Point**, и присваиваете через оператор new новый экземпляр класса Point. В следующей строке показано, как получить доступ к методу setLocation(), передавая величины для параметров (x и y).

Свойства

Ключевой особенностью языка C#, а также других .NET-языков является возможность использовать *свойства*. Свойство – это самый простой путь получить доступ к данным объекта. Они позволяют сохранять локальные переменные в пределах класса: благодаря свойствам осуществляется контроль над данными (вы можете потом изменить выполнение родительского класса без изменения способа использования вашего класса).

Давайте посмотрим по коду ниже, как можно добавить свойства к классу.

```
public class Point
{
  private int x, y;

  public void setLocation(int x, int y)
  {
    this.x = x;
    this.y = y;

    // Далее выполняются некоторые вычисления
    //
  }

  public int X
  {
    set
    {
      this.x = value;
    }
    get
    {
      return this.x;
    }
  }
}
```

В коде, представленном выше, **жирным** выделено то, как вы можете использовать свойства в своем классе для решений внутри своего проекта, или используя чьи-либо имеющиеся разработки.

Объявление свойства начинается с часто используемого ключевого слова **public**. Следующим словом указывается тип свойства как целочисленное и дается имя **X**.

Часть с ключевым словом **get** описывает, что должно произойти при считывании данных из свойства, а часть с **set** указывает на то, что произойдет при присвоении значения свойству.

Вы уже сталкивались с использованием сочетания **this.x** для доступа к данным экземпляра данного класса (или объекта), но ключевое слово **value** в блоке *set* вас пока не знакомо. Это слово дает доступ к информации, назначенной свойству (что-то вроде параметра в методе). В процессе присвоения вы берете это значение и присваиваете его данным.

Когда кто-то пытается получить данные из свойства экземпляра объекта Point, блок кода get данного свойства приступит к исполнению и вернет значения после ключевого слова **return,** которым вы передаете внутренние данные класса.

Давайте посмотрим, как данное свойство будет использоваться в коде:

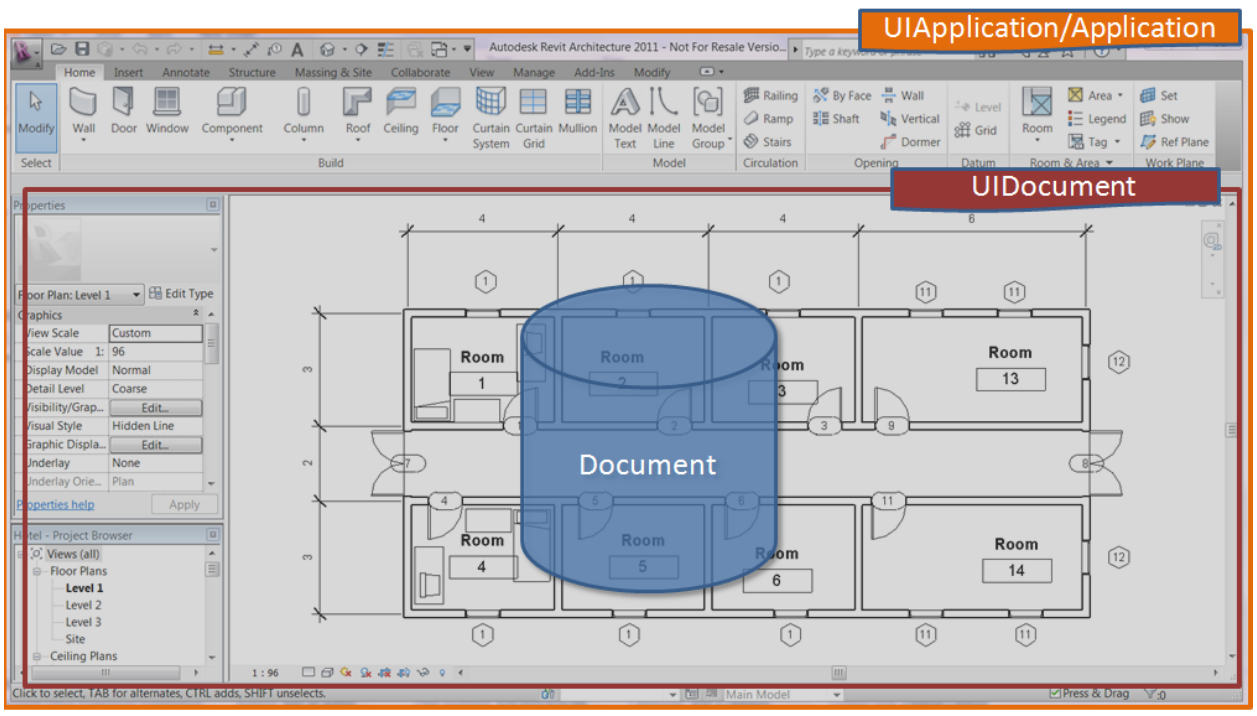
```
Point pt = new Point();
pt.setLocation(10, 10);
// получение значения свойства X
int xCoord = pt.X;
// установка значения свойства X
pt.X = 20;
```

Вы начинаете с создания нового объекта типа **Point** с именем **pt**,.

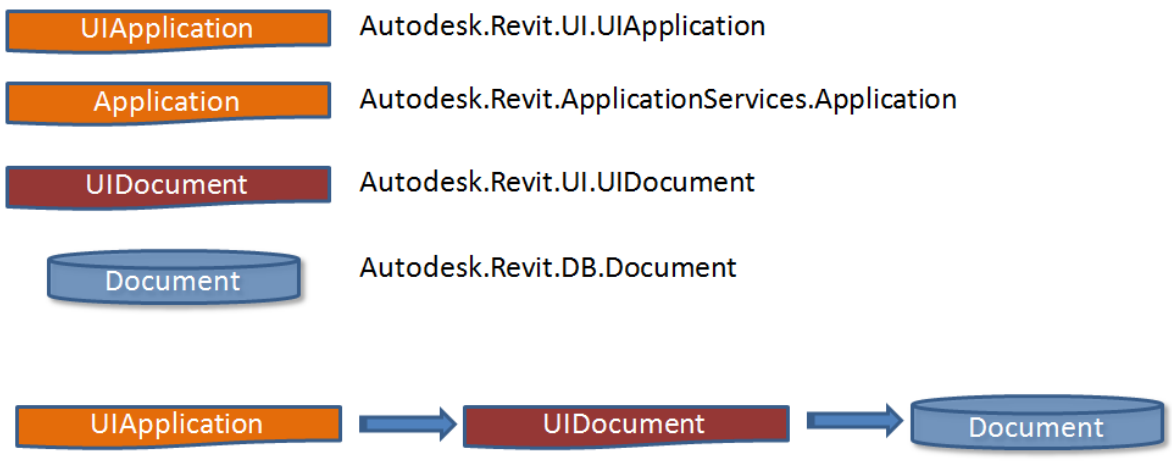
Строки, выделенные **жирным,** показывают, как получать данные в свойство **X** объекта **pt** , присваивая это значение переменной **xCoord**, и затем – как передавать данные в свойство, присваивая ему значение **20**.

Классы Autodesk Revit Application и Document

Давайте проанализируем некоторые из основных классов Revit API и посмотрим, как они соотносятся между собой. Далее на картинке показано, как эти классы Revit API представлены в контексте пользовательского интерфейса Revit.



Далее на картинке показаны данные классы с полным пространством имен, и как они доступны через свойства друг друга (то есть вы сможете получать доступ к любому из них через свойства).



Давайте разберемся, какие данные предоставляет каждый из этих классов:

Autodesk.Revit.UI.UIApplication:

Дает доступ к активному в данный момент приложению Revit и его пользовательскому интерфейсу. Предоставляет возможность изменения ленты, регистрации событий и доступ к активному документу.

Autodesk.Revit.ApplicationServices.Application:

Дает доступ к приложению Revit, документам, настройкам и другим данным по приложению.

Autodesk.Revit.UI.UIDocument:

Дает доступ к активному проекту и пользовательскому интерфейсу для документа. Позволяет оперировать пользовательским выбором, дает возможность предложить пользователю выбрать точки, объекты и т.д.

Autodesk.Revit.DB.Document:

Дает доступ к базе данных активного проекта Revit, например к элементам, видам и т.п.