

TABLE OF CONTENTS

1	WHAT IS THE CODE CHECKING SDK	5
1.1	WHAT IS CODE CHECKING FRAMEWORK FOR REVIT USERS?.....	5
1.2	WHAT IS CODE CHECKING FRAMEWORK FOR DEVELOPERS?.....	9
1.3	WHAT IS CODE CHECKING FRAMEWORK SDK?.....	10
1.4	INSTALLATION AND CONFIGURATION.....	10
1.4.1	<i>System requirements.</i>	10
1.4.2	<i>Content.</i>	11
1.4.3	<i>Configuration</i>	12
1.5	VISUAL STUDIO	13
1.5.1	<i>Visual Studio template.</i>	13
1.5.2	<i>Project structure.</i>	17
1.5.3	<i>Visual Studio Addin</i>	21
2	EXTENSIBLE STORAGE FRAMEWORK	22
2.1	EXTENSIBLE STORAGE	22
2.2	EXTENSIBLE STORAGE FRAMEWORK	22
2.3	DATA MANAGEMENT AND SERIALIZATION	23
2.3.1	<i>SchemaClass</i>	24
2.3.2	<i>SchemaAttribute</i>	25
2.3.3	<i>Versioning</i>	25
2.3.4	<i>SchemaPropertyAttribute</i>	27
2.3.5	<i>Subschemas</i>	30
2.3.6	<i>Arrays and maps</i>	30
2.3.7	<i>Revit elements and enum types</i>	31
2.3.8	<i>Example</i>	31
2.4	UI	32
2.4.1	<i>Layout creation</i>	32
2.4.2	<i>UI attributes</i>	34
2.4.3	<i>IServerUI</i>	51
2.4.4	<i>ISchemaEditor</i>	53
2.4.5	<i>Layout data initialization</i>	54
2.4.6	<i>Example</i>	55
2.5	DOCUMENTATION	56
2.5.1	<i>Document Body</i>	56
2.5.2	<i>Document element creation</i>	63
2.5.3	<i>Attributes</i>	64
2.5.4	<i>SubSchemaListTableAttribute</i>	67
2.5.5	<i>Example</i>	68
2.6	PROPERTY GENERATOR	69
2.7	SCHEMA CLASS TEMPLATE	72
2.8	COMPLEX EXAMPLE	74
2.8.1	<i>Design</i>	74
2.8.2	<i>Class definition</i>	74
2.8.3	<i>UI</i>	76
2.8.4	<i>Documentation</i>	86
2.8.5	<i>Serialization to ProjectInfo</i>	91
2.8.6	<i>Complete Code</i>	91
3	CODE CHECKING FRAMEWORK	96
3.1	WORKFLOW	96
3.2	HAVE TO KNOW	96
3.3	COMPONENTS	97

3.4	CODE CHECKING APPLICATION	97
3.4.1	<i>Data</i>	98
3.4.2	<i>Servers</i>	98
3.4.3	<i>Revit Applications</i>	98
3.5	ICODECHECKINGSERVER	99
3.5.1	<i>Categories</i>	99
3.5.2	<i>Definition of data structures</i>	100
3.5.3	<i>Server</i>	101
3.5.4	<i>MultiStructureServer</i>	101
3.5.5	<i>Notification Service</i>	102
3.5.6	<i>Load Combinations and Load Cases</i>	103
3.5.7	<i>Results Builder</i>	104
3.6	DOCUMENTATION	105
3.6.1	<i>ICodeCheckingServerDocumentation</i>	105
3.7	STORAGE	106
3.7.1	<i>StorageService</i>	106
3.7.2	<i>Label</i>	107
3.7.3	<i>CalculationParameters</i>	107
3.7.4	<i>Results</i>	108
3.7.5	<i>Result Status</i>	109
3.8	UI	110
3.8.1	<i>ICodeCheckingServerUI</i>	110
3.8.2	<i>BuildLayout</i>	111
3.8.3	<i>GetHelp</i>	111
3.8.4	<i>ServerUI and MultiStructureServerUI</i>	112
3.9	EXAMPLE	114
3.9.1	<i>Goal</i> :	114
3.9.2	<i>Design</i>	114
3.9.3	<i>Project</i>	115
3.9.4	<i>Data</i>	115
3.9.5	<i>Servers</i>	119
3.9.6	<i>Revit Applications</i>	121
3.9.7	<i>Calculation</i>	123
3.9.8	<i>Load cases, combinations and result builder</i>	129
3.9.9	<i>Wizard</i>	132
4	CODE CHECKING FRAMEWORK CONCRETE	135
4.1	DESCRIPTION	135
4.2	PROJECT CREATION	135
4.2.1	<i>Setting general options</i>	137
4.2.2	<i>Setting Code Checking options</i>	138
4.2.3	<i>Setting General concrete options</i>	139
4.3	DEFAULT DATA AND CONTROLS	143
4.3.1	<i>Code Settings</i>	143
4.3.2	<i>Element Settings</i>	143
5	CONCRETE PROJECT - MAIN CALCULATION LOOP	146
5.1	ENGINE	146
5.1.1	<i>Running main loop</i>	148
5.1.2	<i>Internal algorithm of main loop</i>	148
5.2	INTERFACE IENGINEDATA	149
5.2.1	<i>GetInputDataUnitSystem</i>	149
5.2.2	<i>GetNumberOfThreads</i>	149
5.2.3	<i>CreateCalculationScenario</i>	149
5.2.4	<i>CreateCommonParameters</i>	150
5.2.5	<i>CreateCalcPointsForElement</i>	150

5.2.6	<i>CreateSectionData</i>	151
5.2.7	<i>CreateElementData</i>	151
5.2.8	<i>CreateElementResult</i>	152
5.2.9	<i>ReadCalculationParameter</i>	153
5.2.10	<i>ReadElementLabel</i>	153
5.2.11	<i>VerifyElementLabel</i>	154
5.2.12	<i>FilterElementForCalculation</i>	155
5.2.13	<i>ReadFromRevitDB</i>	156
5.2.14	<i>SaveToRevitDB</i>	156
5.3	IMPLEMENTATION OF CODE	156
5.3.1	<i>Element objects</i>	159
5.3.2	<i>Section objects</i>	159
5.3.3	<i>CommonParameters</i>	160
5.3.4	<i>Calculation objects</i>	161
5.3.5	<i>Calculation scenario</i>	162
6	REINFORCEMENT CONCRETE COMPONENT	165
6.1	UNITS	165
6.2	CONCRETE MATERIAL	166
6.2.1	<i>Rectangular model</i>	167
6.2.2	<i>Linear model</i>	168
6.2.3	<i>Bilinear model</i>	169
6.2.4	<i>Parabolic-Rectangular model</i>	170
6.2.5	<i>Power-Rectangular model</i>	171
6.3	STEEL MATERIAL	172
6.4	REBAR DEFINITION	173
6.5	GEOMETRY	175
6.6	RCSOLVER	176
6.6.1	<i>SolveResistance - section resistance</i>	176
6.6.2	<i>SolveResistanceF - section resistance for fixed moments</i>	177
6.6.3	<i>SolveResistanceM - section resistance for fixed axial force</i>	178
6.6.4	<i>SolveForces - forces at a given strain</i>	179
6.6.5	<i>Simple output</i>	180
6.6.6	<i>Calculation results</i>	182
6.8	VERIFICATION EXAMPLES	188
6.8.1	<i>Case 1</i>	188
6.8.2	<i>Case 2</i>	188
6.8.3	<i>Case 3</i>	188
6.8.4	<i>Case 4</i>	188
6.8.5	<i>Case 5</i>	188
6.8.6	<i>Case 6</i>	189
6.8.7	<i>Case 7</i>	189
6.8.8	<i>Case 8</i>	189
6.8.9	<i>Case 9</i>	189
6.8.10	<i>Case 10</i>	189
6.8.11	<i>Case 11</i>	189
6.8.12	<i>Case 12</i>	190
6.8.13	<i>Case 13</i>	190
6.8.14	<i>Case 14</i>	190
6.8.15	<i>Case 15</i>	190
6.8.16	<i>Case 16</i>	190
6.8.17	<i>Case 17</i>	191
6.8.18	<i>Case 18</i>	191
7	ENGINEERING COMPONENT	192
7.1	ELEMENT ANALYZER	192

7.1.1	<i>Units</i>	192
7.1.2	<i>Element</i>	193
7.1.3	<i>Material</i>	193
7.1.4	<i>Cover</i>	197
7.1.5	<i>Element Sections Parameters and Sections Dimensions</i>	199
7.1.6	<i>Element Slabs</i>	206
7.2	FORCE RESULT CACHE	211
7.2.1	<i>ForceLoadCaseDescriptor</i>	211
7.2.2	<i>ForceResultsPackageDescriptor</i>	212
7.2.3	<i>ForceCalculationDataDescriptor</i>	212
7.2.4	<i>ForceCalculationDataDescriptorLinear</i>	213
7.2.5	<i>ForceResultCache</i>	213

1 WHAT IS THE CODE CHECKING SDK

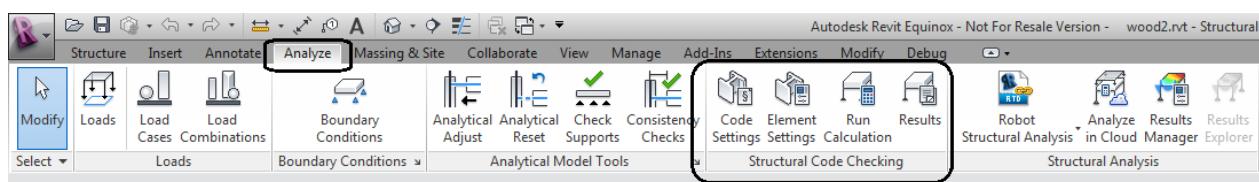
1.1 What is Code Checking Framework for Revit users?

The Structural Code Checking Framework is a new Revit feature to give Revit users the ability to perform a Code Checking process inside a native Revit model.

Revit users will now be able to define some rules for an overall Code Checking application and a set of parameters defining expected behavior for individual analytical model elements or group of analytical model elements. These parameters will extend Revit model and can be used as attributes for the calculation process itself.

- **Ribbon:**

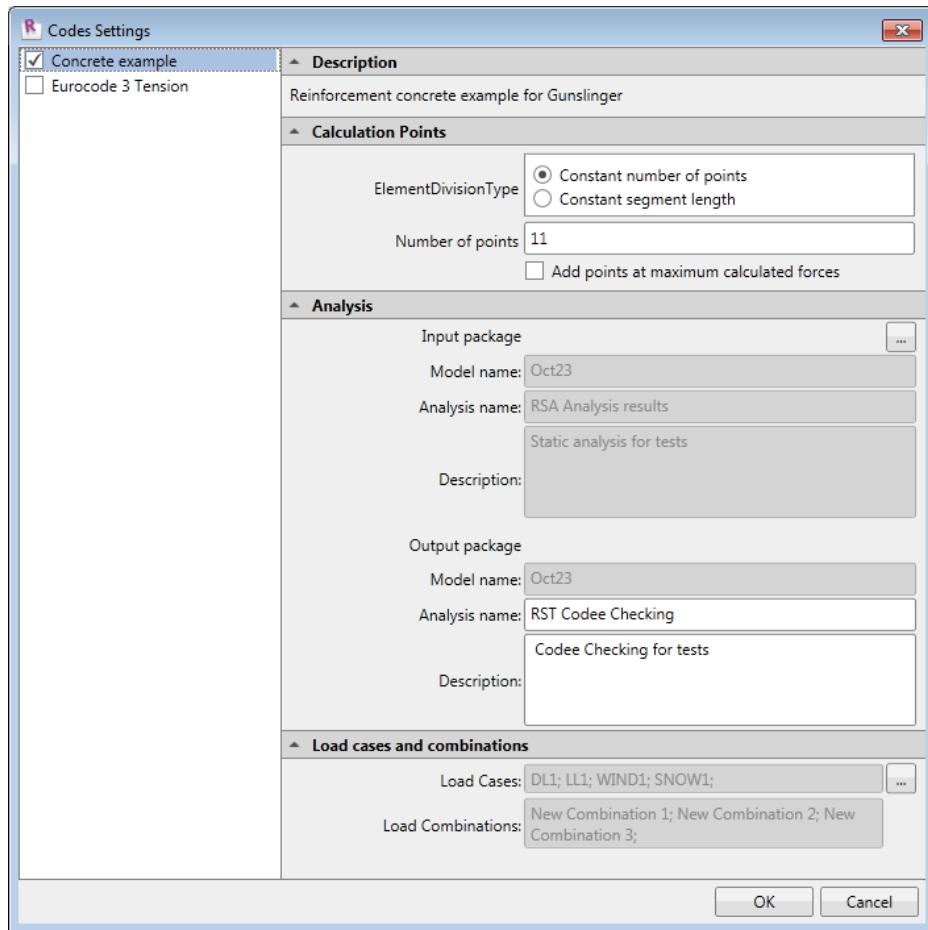
For that purpose, a set of new commands dedicated to Structural Code Checking have been exposed. These commands could be found on the ribbon tab Analyze, on the panel Structural Code Checking.



- **Command Code Settings:**

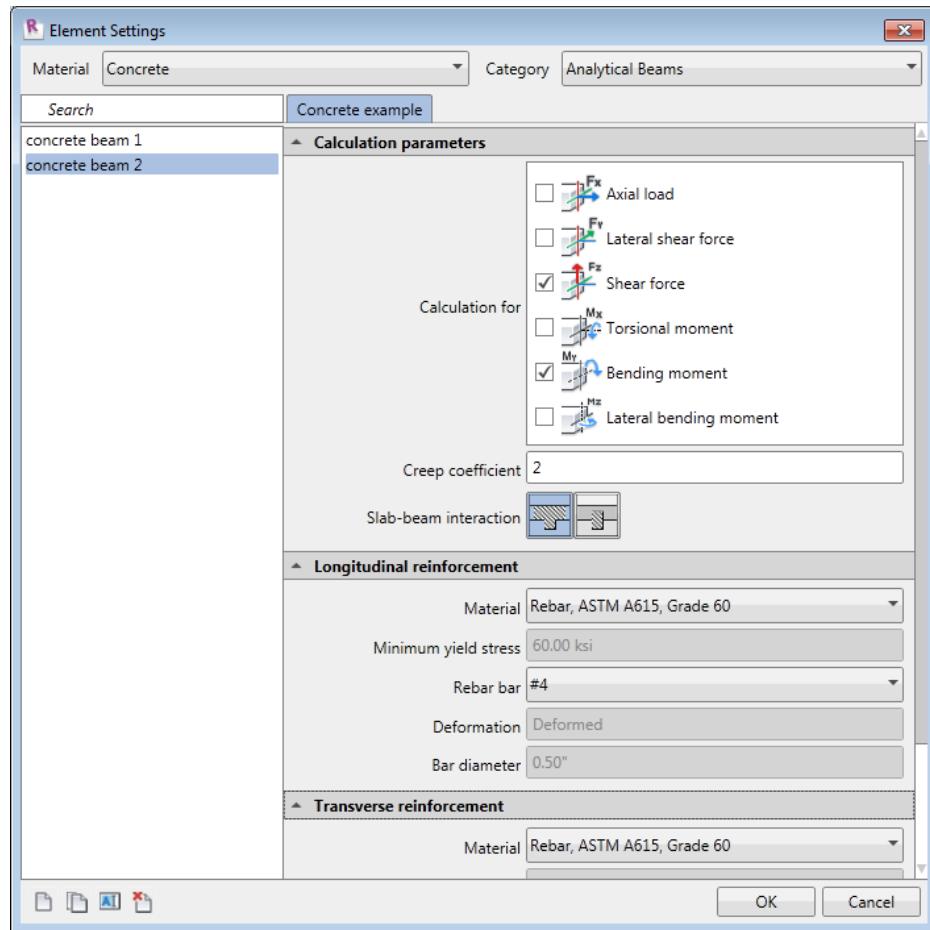
The Code Settings dialog is a placeholder to activate codes available and to define some global settings, common for all elements which will be calculated. These settings could be:

- Code specifics parameters and calculations options to take into consideration
- Structural analysis package to use as input data
- Load cases and combinations available in Revit model that will be used for the calculation process

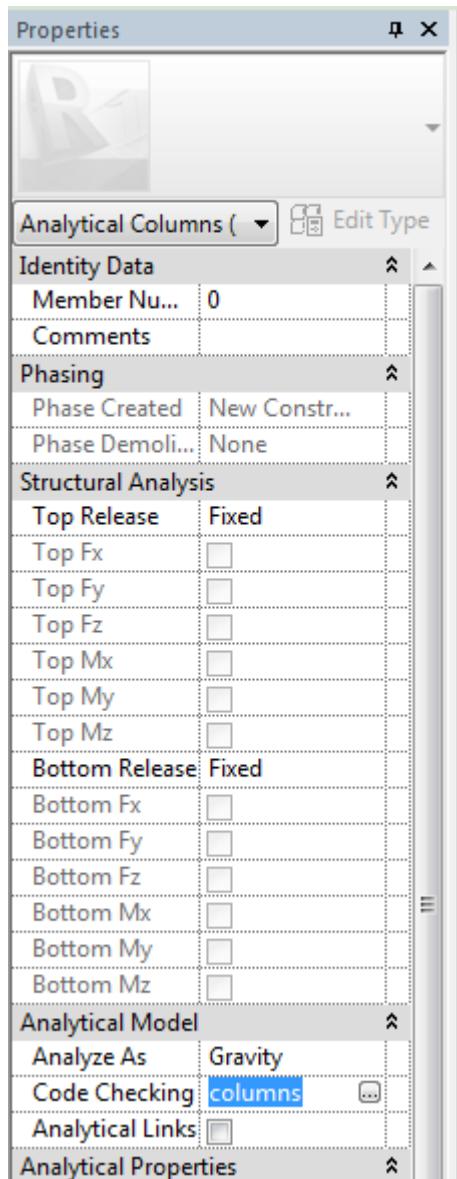


- **Command Element Settings:**

The Element Settings dialog is the placeholder to define and manage a code specific set of parameters that could be applied to elements before running calculations.



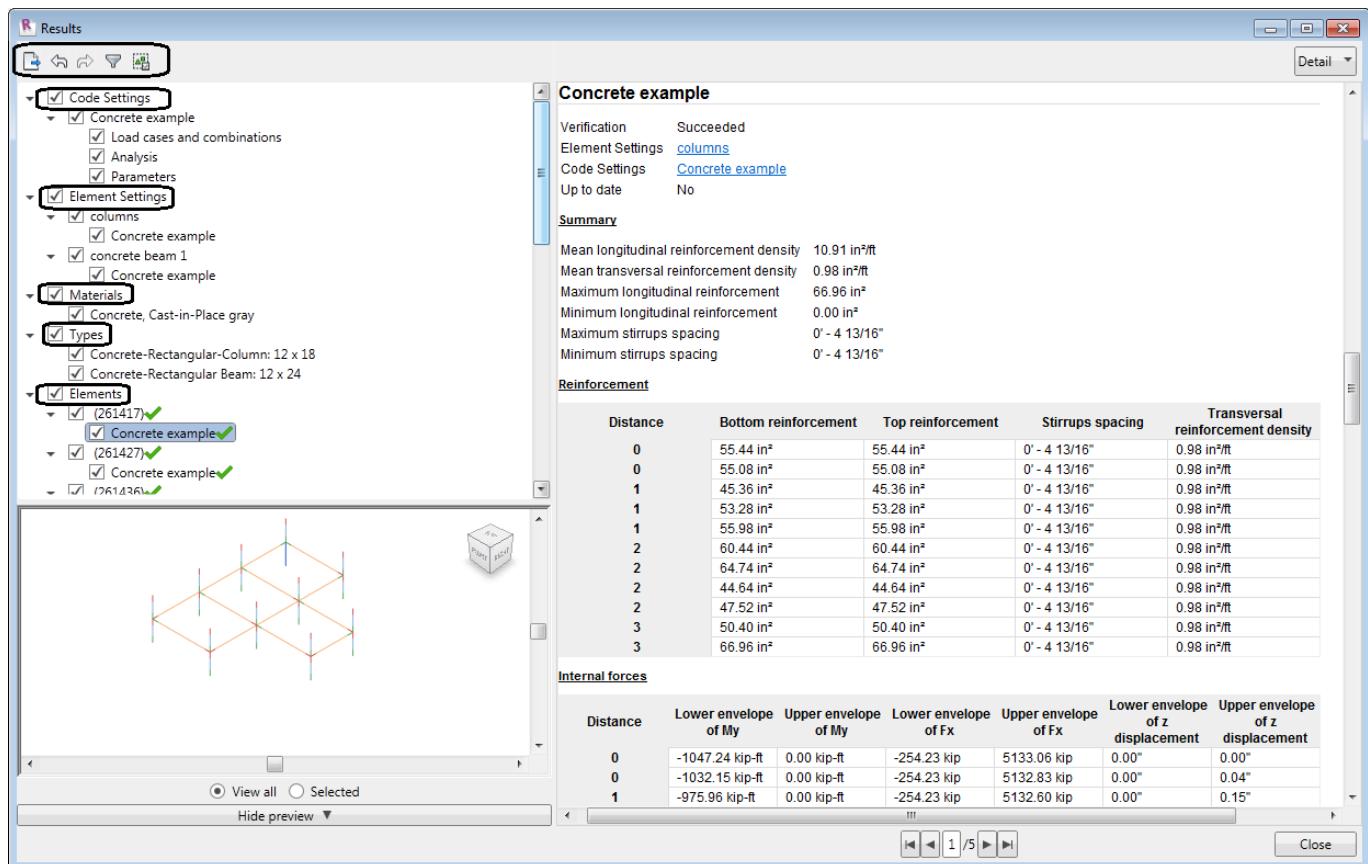
To assign a set of parameter to Revit element(s), this dialog could be called via the Code Checking Parameter available for all analytical model element categories on the Properties Palette.



- Command Run Calculations:**

The Run Calculations command will perform calculations based on selected code(s) and family instance(s).

At the end of calculation, results are exposed on a report dialog.



Some additional tools are exposed on the menu of this dialog:

- Export the report as html and Mht
- Filter elements based on their status
- Create a Revit selection that could be reused

Inside the navigation tree could be found the settings used to performed calculations and some information related to Revit model (i.e. family instance and material). All calculated elements are listed with a specific status.

The report and the 3D viewer are synchronized with the navigation tree.

- **Command Results:**

The command Results allow to retrieve calculations results for a specific set of elements.

From Revit Users perspective, to activate this functionality the Structural Analysis and Code Checking Toolkit for Autodesk Revit available for free from the Exchange app website and a Code Checking Application should be installed on the client machine.

1.2 What is Code Checking Framework for Developers?

To enable the Code Checking workflow for Revit users, some Code Checking applications should be created. To achieve this, the Code Checking Framework should be leverage by Revit API developers.

From Revit API developer perspective, the Code Checking Framework is an API platform supporting development of Structural Code Checking Applications for Revit and making them consistent and aligned with the way Revit interacts with users.

The primary goal of the Code Checking Framework is to help Revit API developers concentrate on essential development aspects when creating Structural Code Checking Applications for Revit, by providing support for typical, commonly used functionalities.

The outcome of this approach is:

- A full and consistent integration of Structural Code Checking Applications within the Revit environment (UI and behaviors)
- Acceleration of Code Checking Applications development by providing a set of tools and components for typical Structural Code Checking Application development.
- Enabling easily Structural Code Checking Applications activation and registration by Revit users

1.3 What is Code Checking Framework SDK?

The Code Checking SDK is a development environment for Rapid Application Development purposes that helps to create, deploy and activate add-ins based on the Extensible Storage Framework, Code Checking Framework and engineering components, Results builder and Revit APIs.

The core part of the SDK is implemented as a form of Microsoft Visual Studio C# templates. Using templates provided, developers can quickly build and deploy an application that has a similar look & feel and behaviors to other Code Checking solution working on top of Revit.

The Code Checking Framework SDK takes advantage of the Revit API and of a set of components including UI layout creation feature, advanced controls, Extensible Storage, External Services and structural engineering components.

The Code Checking Framework SDK is composed of:

- Visual Studio tools (C# templates and a Visual Studio addin)
 - Data management and serialization
 - UI definition
 - User interactions
 - Report generation
- Documentation
 - Getting Started
 - User Manual
 - Samples and associated documentation
 - API documentation (chm)

3 types of C# templates are provided with this SDK to create some external commands using the Extensible Storage Framework, some generic Code Checking applications and some Code Checking applications dedicated to required reinforcement for concrete beams and columns.

To support the creation of Visual Studio projects based on templates, the SDK includes a Visual Studio addin. This addin provides also a property generation feature to facilitate data definition.

1.4 Installation and configuration

1.4.1 System requirements

- Visual Studio 2010 (Express version of Visual Studio are not supported)
- .NET Framework 4.0
- Revit 2014
- Structural Analysis and Code Checking Toolkit for Autodesk Revit

1.4.2 Content

Folder	Files	Description
Documentation	Read Me First.pdf	Document containing the packing list for the Code Checking SDK.
	Getting Started Code Checking Framework SDK.pdf	Getting started document contains information about the basics of Code Checking SDK and how to create a first Code Checking application.
	User Manual for Code Checking Framework SDK.pdf	User manual document contains detailed information about the Code Checking Framework API and SDK.
	Step by Step Example – Code Checking Concrete.pdf	This document explains how the CodeChecking Concrete example in the samples folder is constructed starting from an empty project.
	Manual Verifications Concrete.pdf	This document is a validation manual for the concrete reinforcement component part of the Code Checking SDK
	Structural Analysis and Code Checking API Reference Guide.chm	This document is a chm documentation of public API exposed by the Code Checking Framework component

Folder	Files	Description
Example	ExtensibleStorageUI	Sample to learn how to take advantage of the ExtensibleStorage Framework (UI part).
	ExtensibleStorageDocumentation	Sample to learn how to take advantage of the ExtensibleStorage Framework (Report part).
	CodeCheckingConcreteExample	Sample of a Concrete Code Checking application. This example should be review with the step by step document.
	ConcreteCalculationsExample	Sample to learn how to take advantage of the Concrete calculations component. This example provides calculations for all cases listed on the calculation manual.
	SectionPropertiesExplorer	Sample to learn how to take advantage of the Engineering component.

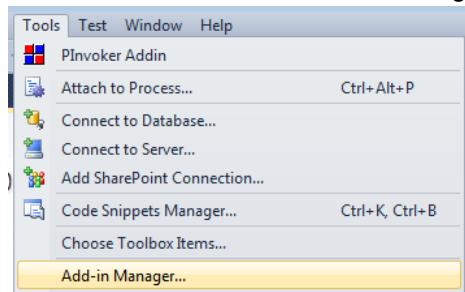
Folder	Files	Description
Visual Studio	Items	This directory contains files to copy on items template folder from Visual Studio.
	Project	This directory contains files to copy on items project folder from Visual Studio.
	Addins	This directory contains an Add-in for Visual Studio and a description how to install it.

1.4.3 Configuration

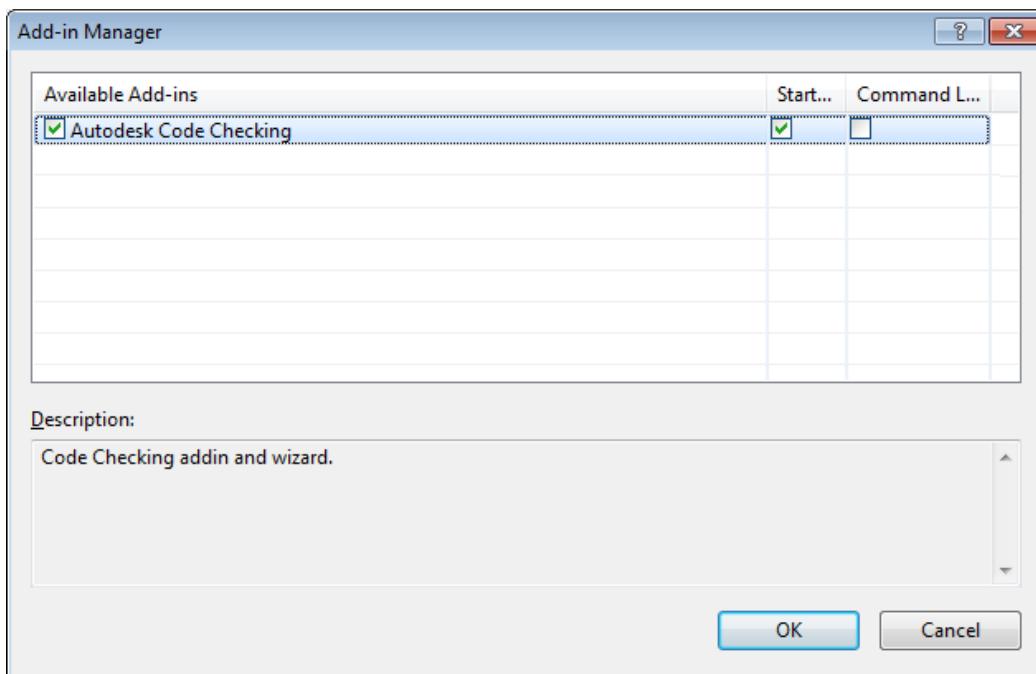
To properly configure Visual Studio 2010, templates should be copied to dedicated folders

- The content of the folder “..\\Software Development Kit \\Structural Analysis SDK\\Visual Studio\\Templates\\Items\\” should be copied to “C:\\Users\\<current user>\\Documents\\Visual Studio 2010\\Templates\\ItemTemplates\\Visual C# Autodesk\\Code Checking”
- The content of the folder “..\\Software Development Kit \\ Structural Analysis SDK\\Visual Studio\\Templates\\Projects\\” should be copied to “C:\\Users\\<current user>\\Documents\\Visual Studio 2010\\Templates\\ProjectTemplates\\Visual C# Autodesk\\Code Checking”
- The content of the folder “..\\Software Development Kit\\ Structural Analysis SDK\\Visual Studio\\Addins” should be copied to “C:\\Users\\<current user>\\Documents\\Visual Studio 2010\\Addins” (if this folder doesn’t exist, it should be created)

To complete the add-in installation, be sure that on Tools>Add-in manager...



The Autodesk Code Checking is installed and activated.

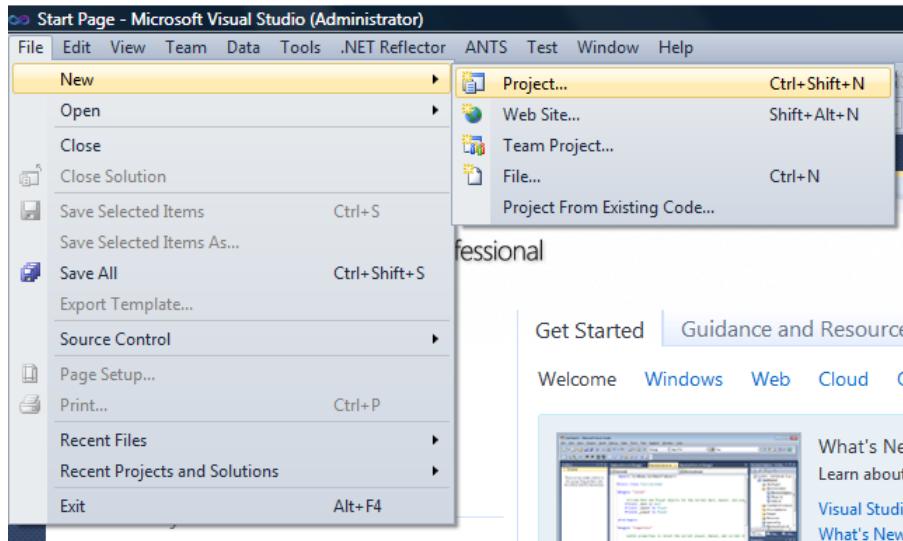


To resolve references used by examples, the StructuralAssemblyResolver.exe tools could be used. Note that the addin files will need to be updated and copied manually.

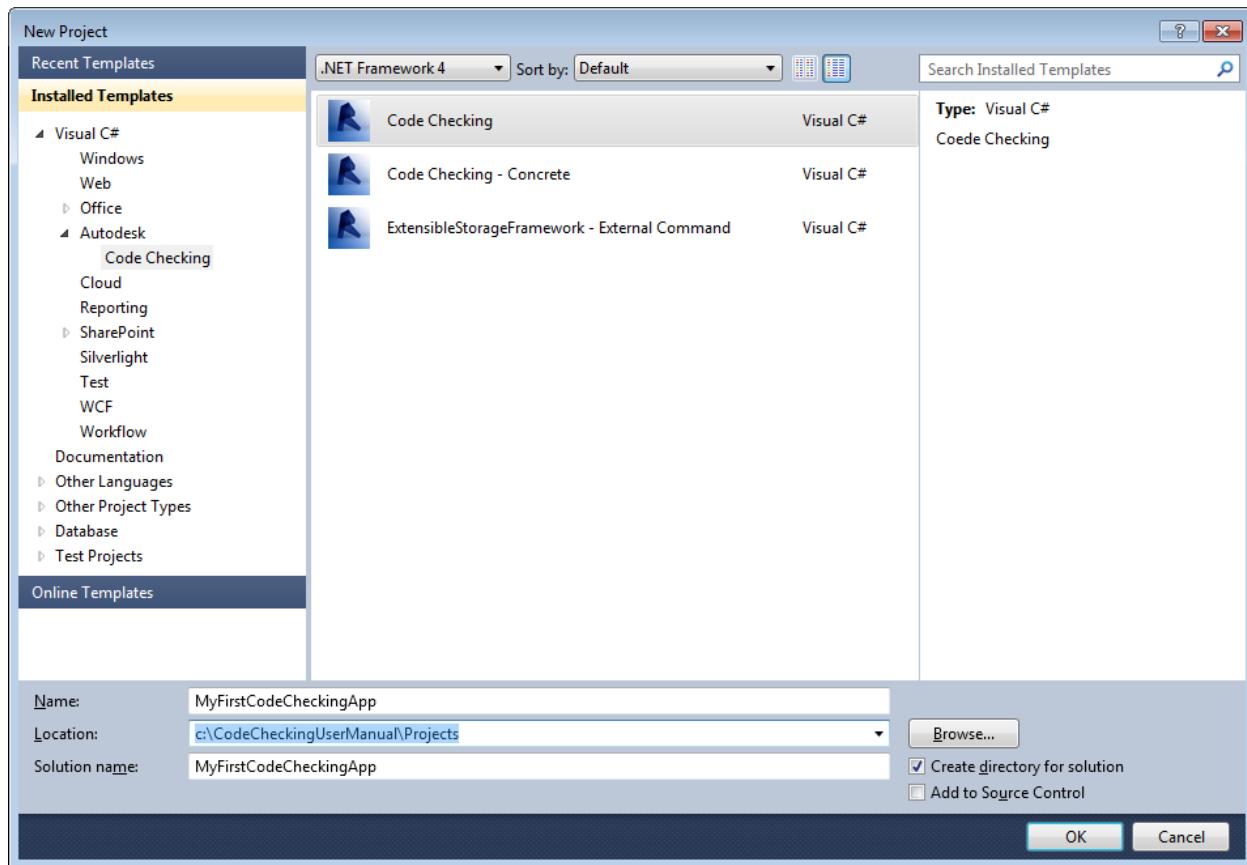
1.5 Visual studio

1.5.1 Visual Studio template

After starting visual studio, go to new project (**File / New / Project**).



Then on Visual C# installed template should be visible Code Checking templates.



3 types of projects are exposed:

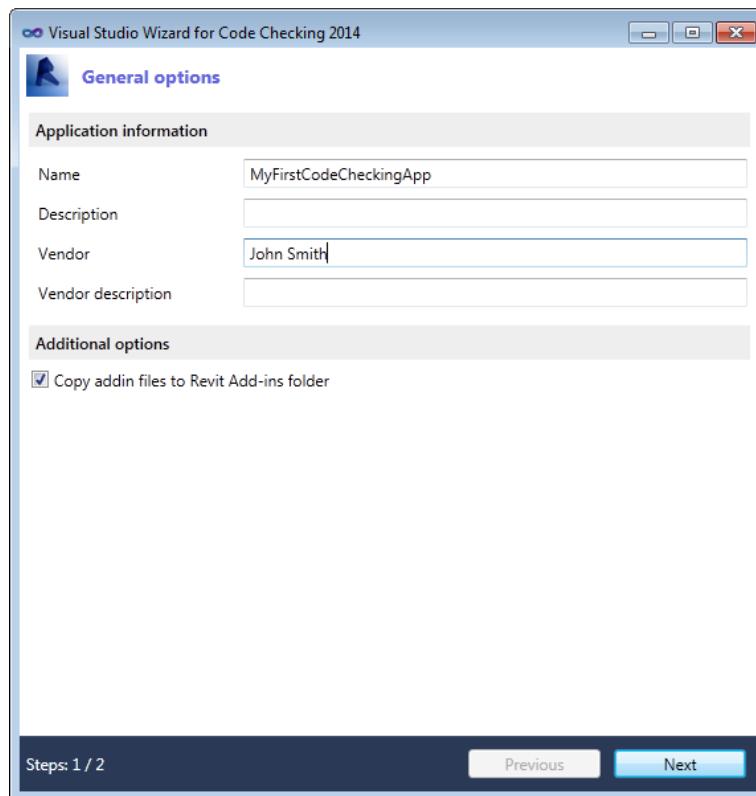
- Code Checking - to create general Code Checking applications
- Code Checking Concrete – to create required reinforcement Code Checking applications for concrete beam and column
- Extensible Storage Framework - to create External command using the Extensible Storage Framework

At this stage of the process, a template should be selected and names of the solution and project should be set.

Note that in this example, the Code Checking project template will be used. Detailed information regarding each kind of template could be found later on this document inside dedicated sections.

The project name is an important reference and should be unique (as you know, you can't load in Revit different assemblies with the same name)

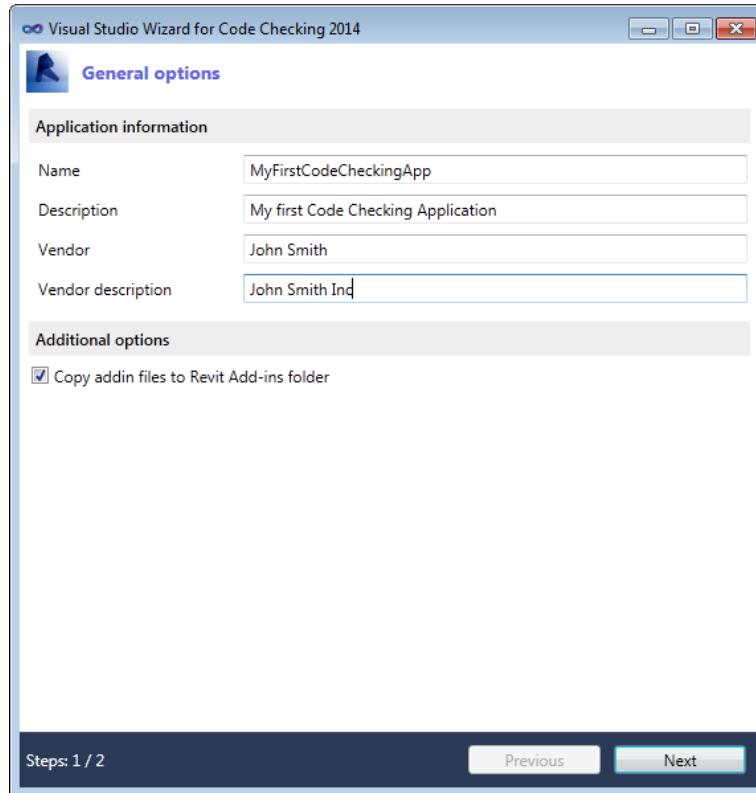
By clicking on the OK button, bellow dialog will be displayed.



The first tab of this wizard is dedicated to Revit addin manifest file creation and addition strings needed as registration of updater and Server.

Per default some fields are filled:

- Name - comes from the name of the solution.
- Description – this field is empty per default and is not mandatory for External application for the manifest file. If this field is left empty, the Name of the solution will be used to set the description of the Code Checking server.
- Vendor – comes from the current user logged.
- Vendor Description - this field is empty per default. If this field is left empty, inside the addin file the Vendor is used to fill it.



As additional option, developer could decide if Revit addin manifest file should be copied to C:\Users\John Smith\AppData\Roaming\Autodesk\Revit\Addins\2014 or not.

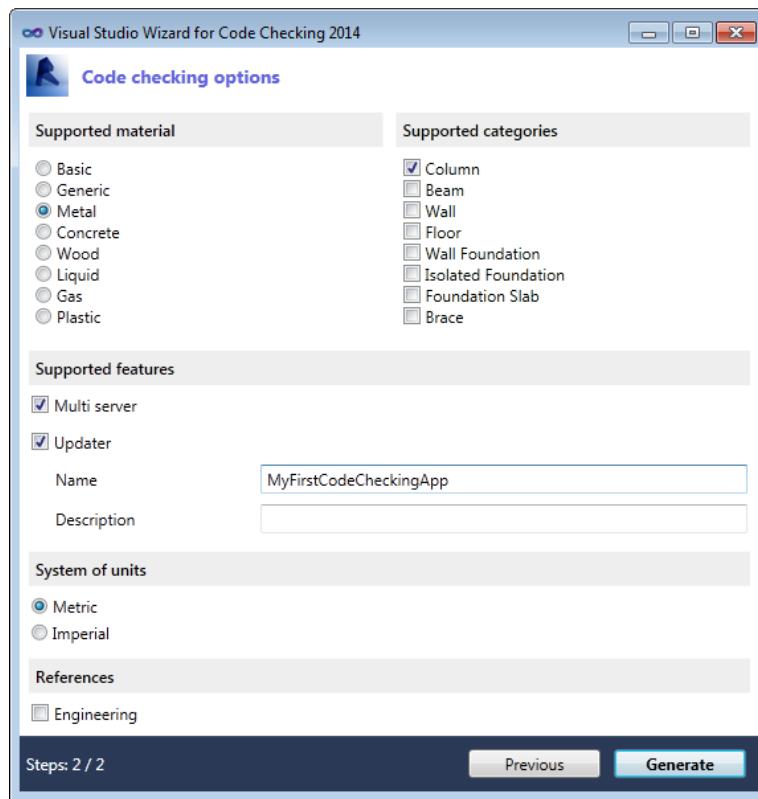
For a Code Checking Application, 2 manifest files are created for a DBapplication and an Application. As show on the code region below, texts set on the UI are taken into consideration. In addition, the path of the assembly is set to the project bin\Debug output, the guid is generated and the FullClassName attribute defined.

Code region

```
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="DBApplication">
    <Name>MyFirstCodeCheckingApp</Name>
    <Assem-
bly>c:\CodeCheckingUserManual\Projects\MyFirstCodeCheckingApp\MyFirstCodeCheckingApp\bin\
Debug\MyFirstCodeCheckingApp.dll</Assembly>
    <AddInId>715ba1d3-8848-47b2-948f-0b8ac40fb367</AddInId>
    <FullClassName>MyFirstCodeCheckingApp.RevitApplicationDB</FullClassName>
    <VendorId>John Smith</VendorId>
    <VendorDescription>my company</VendorDescription>
  </AddIn>
</RevitAddIns>

<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="Application">
    <Name>MyFirstCodeCheckingApp (UI)</Name>
    <Assem-
bly>c:\CodeCheckingUserManual\Projects\MyFirstCodeCheckingApp\MyFirstCodeCheckingApp\bin\
Debug\MyFirstCodeCheckingApp.dll</Assembly>
    <AddInId>585c0cd3-b66b-4209-89b9-7137fa923361</AddInId>
    <FullClassName>MyFirstCodeCheckingApp.RevitApplicationUI</FullClassName>
    <VendorId>John Smith</VendorId>
    <VendorDescription>my company</VendorDescription>
  </AddIn>
</RevitAddIns>
```

The next window contains more options specific for the Code Checking application.
Per default the dialog looks as below:



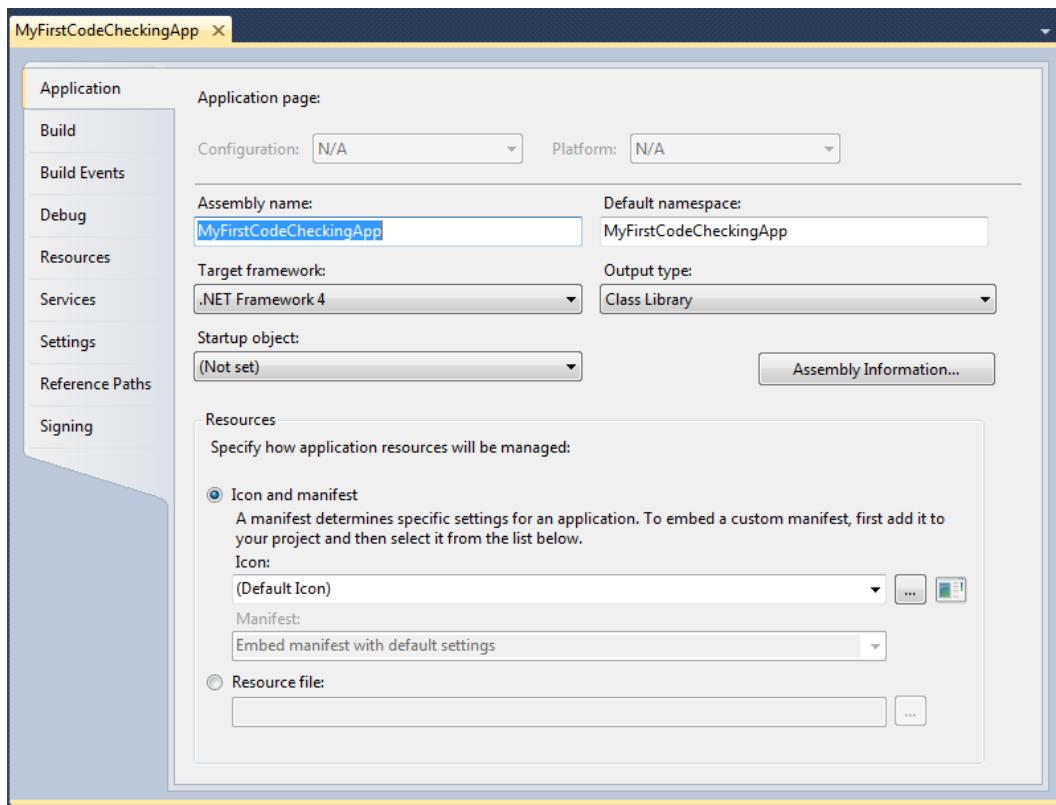
Few options could be set:

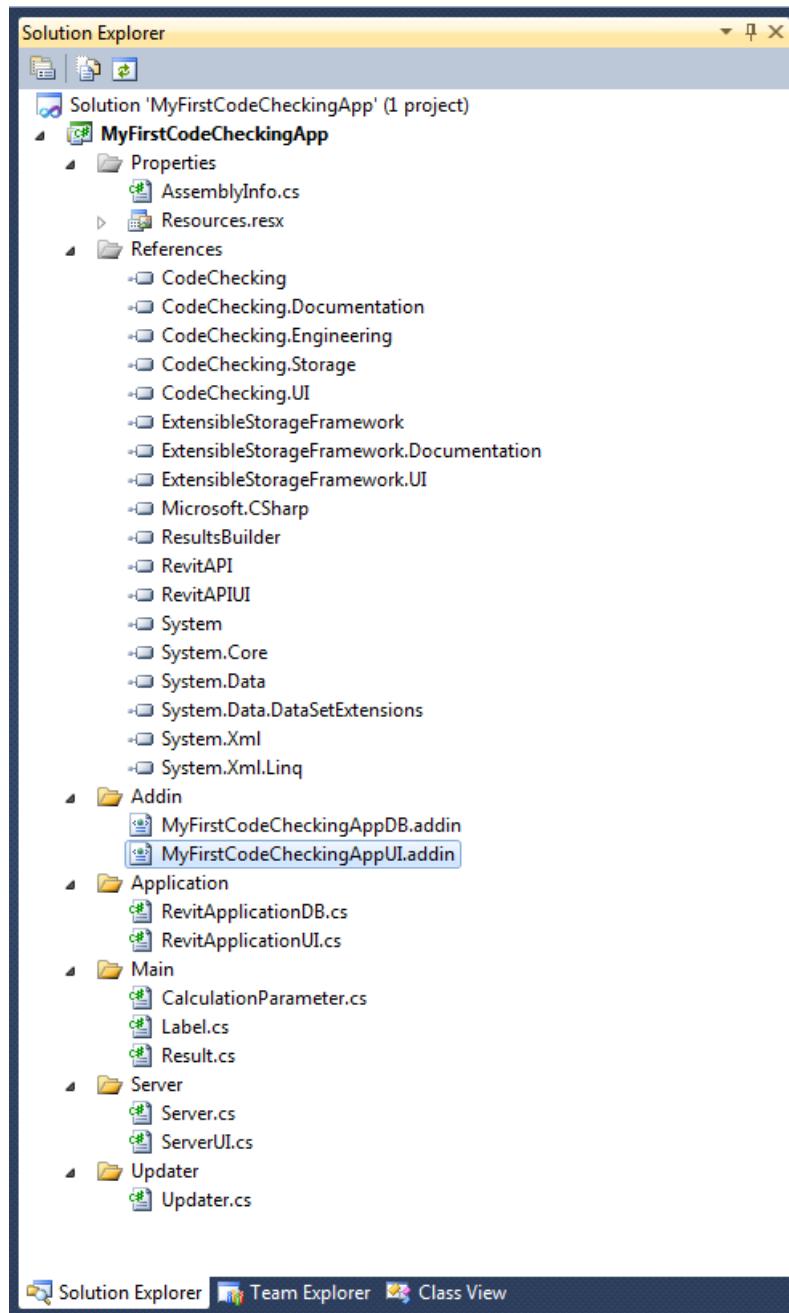
- Supported material – Let developer choose which structural material assigned to element will be supported by his application
- Supported categories - Let developer choose which analytical element categories will be supported by his application
- MultiServer - Let developer choose if MultiServer choose be used. Usage is described later on this document
- Updater - Let the developer decide if is application need an updater. Name and description of the updater could be set at the level of the wizard and the guid is generated automatically if this option is used.
- System of Unit – As describe later, Code Checking Framework leverage Extensible Storage Revit API functionality and a choice of the type of unit use is relevant here. In addition, it's useful for developer to make calculation on his preferred system of units.
- Engineering – Let the developer decide if the Code Checking engineering component should be added as reference inside the new project.

By clicking on the generate button, the visual studio project will be created.

1.5.2 Project structure

During project creation, most important project settings are predefined. The name of the solution and project have been set and as well most important project settings.





- Properties
 - AssemblyInfo.cs - content is filled based on data set on the wizard

Code region

```
// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("MyFirstCodeCheckingApp")]
[assembly: AssemblyDescription("John Smith")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("John Smith")]
[assembly: AssemblyProduct("MyFirstCodeCheckingApp")]
[assembly: AssemblyCopyright("Copyright © John Smith. 2014")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

- Resources.resx - placeholder for the strings to localize
- **References**
 - .Net library
 - Revit
 - Revit API - API for Revit DB layer
 - Revit API UI - API for Revit UI layer
 - Results Builder - Component to access results for Revit project
 - Code Checking
 - CodeChecking – Base classes and foundations for Code Checking applications
 - Code Checking Storage – Access to Data
 - CodeChecking Documentation – Tools to create calculation reports
 - Code Checking UI - Create and manage UI
 -
 - Extensible Storage Framework
 - Extensible Storage Framework - Data management
 - Extensible Storage Framework UI – Data UI representation
 - Extensible Storage Framework Documentation - Data documentation representation

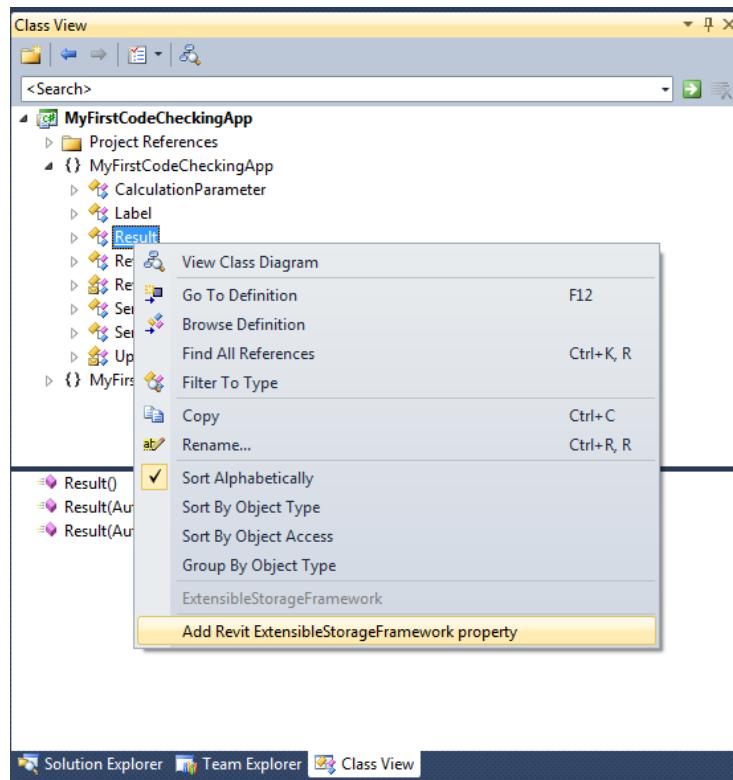
Note:

- For an Extensible storage Framework command, results builder and Code Checking are not present on the project
- For a Code Checking concrete application, two additional libraries are added – Code Checking concrete and Code Checking engineering (both could be used also on a regular Code Checking application project)
- **Addin** - Folder where the two needed manifest file are located. The first manifest file is dedicated to the DB application and the second one to the UI application
- **Application** - folder where the source of the two application are stored
- **Main** - A Code Checking application is based on three independent data structure. This folder is the place holder for these data structure.
- **Server** - A Code Checking application is based on a Server and a server UI, this folder is the placeholder for the source code of this two main classes.

- **Updater** – A Code Checking could use some updaters. In this location is the body of the updater that could be used or not according needs by the developer.

1.5.3 Visual Studio Addin

The Visual Studio Addin provides the wizard to create the C# project. A second useful functionality is exposed here, in class view and by selecting any schema classes (Label, Result, CalculationParameter in this case), developer could generates properties and attributes. Next chapter dedicated to the extensible Storage framework will show the interest of this feature.



2 EXTENSIBLE STORAGE FRAMEWORK

The Extensible Storage Framework provides developers set of tools and APIs built around Revit Extensible Storage which automates:

- Data Management and serialization
- UI creation
- Documentation creation

The Code Checking Framework SDK provides also a Visual Studio addin that facilitates operation and a template wizard to automate the Visual Studio project creation.

2.1 Extensible Storage

Since 2012 version Revit API provides Extensible Storage which allows users to store data inside any Revit element. This feature is based on schemas which define structures of data. Users are able to define any number of schemas with fields of different types and units. Below you can find the example of a simple schema which is saved to the element and later reloaded:

Code region

```
Guid id = new Guid("74818248-224C-42E6-AEB7-8FACD5DF887A");
SchemaBuilder builder = new SchemaBuilder(id);
builder.SetSchemaName("MySchema");
builder.AddSimpleField("a", typeof(double)).SetUnitType(UnitType.UT_Length);
builder.AddSimpleField("b", typeof(string));
Schema schema = builder.Finish();

Entity entity = new Entity(schema);
entity.Set<double>("a", 1, DisplayUnitType.DUT_METERS);
entity.Set<string>("b", "text");
element.SetEntity(entity);

entity = element.GetEntity(schema);
double a = entity.Get<double>("a", DisplayUnitType.DUT_CENTIMETERS);
string b = entity.Get<string>("b");
```

For more information about Extensible Storage please refer to Revit API documentation.

2.2 Extensible Storage Framework

The Extensible Storage Framework extends Extensible Storage by providing tools which helps users to develop their application faster. It is data oriented mechanism were all decisions are taken at the level of the classes definitions. Users decide about different aspects of each property especially:

- How property should be serialized in Revit Extensible Storage?
- What control should be used for edition of the field in a dialog?
- How property should be presented in the document?

The answers for these questions are delivered by users via dedicated attributes which are assigned to properties. Extensible Storage Framework provides sets of attributes for serialization, UI and documentation. Below you can find the example of a simple property with attributes for each of mentioned aspect:

Code region

```
[SchemaProperty(Unit=UnitType.UT_Length,DisplayUnit=DisplayUnitType.DUT_METERS)]
[UnitTextBox]
[ValueWithCaption]
public Double a{get;set;}
```

2.3 Data Management and Serialization

Component: ExtensibleStorageFramework.dll

Namespace: Autodesk.Revit.DB.ExtensibleStorage.Framework

Extensible Storage Framework provides mechanism which allows for:

- Automatic schema creation
- Automatic serialization and deserialization of objects
- Subschemas (any number of levels according Revit possibilities)
- Support versioning of schemas
- Definition of fields with default names and types (taken as properties are defined in the class)
- Definition of fields with names and types different than properties in the class
- Definition of converters (for types not supported by Extensible Storage e.g. DateTime)
- Extra serialization actions for specific fields
- Post and pre actions for saving and loading

Extensible Storage Framework serialization mechanism is based on class definition and attributes assigned to specific properties. Following pieces of codes are equivalent:

Code region

```
SchemaBuilder builder = new SchemaBuilder("CB03232D-B362-45D9-B695-3E372C9D8413");
builder.SetSchemaName("MySchema");

builder.AddSimpleField("a", typeof(double)).SetUnitType(UnitType.UT_Length);
builder.AddSimpleField("b", typeof(string));

Schema schema = builder.Finish();
```

Code region

```
[SchemaAttribute("CB03232D-B362-45D9-B695-3E372C9D8413", "MySchema")]
public class MySchema : SchemaClass
{
    [SchemaPropertyAttribute(UnitType.UT_Length, DisplayUnitType.DUT_METERS)]
    public double a { get; set; }

    [SchemaPropertyAttribute]
    public string b { get; set; }
}
```

To define a schema using directly Extensible Storage or using Extensible Storage Framework, will need to define a Guid and a name for the schema and define the name, type and units (if needed) for each field.

There are three main elements which users have to pay attention to:

- SchemaClass
- SchemaAttribute – indicates that a class can be converted to the schema
- SchemaPropertyAttribute – indicates that a property of the class should be converted to the field in the schema

2.3.1 SchemaClass

The SchemaClass is a base class for all classes which definition is used to build Revit schemas, it provides mechanism to create, read and load Extensible Storage schemas based on class definition

Methods	Description
<code>public virtual bool Load(Element element)</code>	Loads data from Revit element to this instance.
<code>public virtual bool Save(Element element)</code>	Saves data of this instance to Revit element.
<code>public void Remove(Element element)</code>	Removes class data from Revit element.
<code>public bool Exists(Element element)</code>	Checks if class data is stored in Revit element
<code>public void SetProperties (Entity entity, Document document = null)</code>	Sets properties of this instance based on input entity.
<code>public Entity GetEntity()</code>	Gets Entity representation of this instance.
<code>public void FillEntity(Entity entity)</code>	Fills input entity with this instance data.
<code>public int GetVersion(Guid id)</code>	Returns the version of the schema based on history kept in SchemaAttribute.
<code>public virtual bool SaveField(Entity entity, Field field)</code>	Provides special saving method for particular field. If developer decide to use this method to save a field, this method should return true; otherwise standard saving procedure will be applied.
<code>public virtual bool LoadField(Guid loaded, Entity entity, Field field)</code>	Provides special loading method for particular field. If developer decides to use this method to save a field, this method should return true; otherwise standard saving procedure will be applied.
<code>public virtual void PreLoad(Entity entity)</code>	Run before loading procedure.
<code>public virtual void PostLoad(Entity entity)</code>	Run after loading procedure.
<code>public virtual void PreSave(Entity entity)</code>	Run before saving procedure.
<code>public virtual void PostSave(Entity entity)</code>	Run after saving procedure.
<code>public static Schema GenerateSchema(Type type)</code>	Allows to generate schema based on input type.

Below there is a simple example of saving and loading data to `MySchema` class instance.

Code region

```
MySchema mySchema = new MySchema();
mySchema.a = 2;
mySchema.b = "text";
mySchema.Save(element);

MySchema newSchema = new MySchema();
newSchema.Load(element);
```

2.3.2 SchemaAttribute

`SchemaAttribute` indicates that specific class can be converted to the schema. This attribute provides information about:

- Name of the schema
- ID
- History of the schema

The code below presents a class which is registered as a schema of name „`MySchema`” and some particular id:

Code region

```
[SchemaAttribute("CB03232D-B362-45D9-B695-3E372C9D8413", "MySchema")]
public class MySchema : SchemaClass
{}
```

2.3.3 Versioning

In fact there is no versioning mechanism in Revit Extensible Storage Schemas. If there are made any changes in definition of the schema a new schema with new id has to be provided. Generally `SchemaClass` doesn't change this approach. The only thing added is `History` property which stores history of ids. If developer makes changes in the class he or she has to generate new id and place it before the current one:

Code region

```
Version 1:  
[SchemaAttribute("MySchema",  
    "1E578722-BFF5-4496-B436-CB6C659D93DF")] //Version 1 -> current version  
public class Class1 : SchemaClass  
{ }  
  
Version 2:  
[SchemaAttribute("MySchema",  
    "261177F2-0475-4B0D-B458-DF5609A330F7", //Version 2 -> current version  
    "1E578722-BFF5-4496-B436-CB6C659D93DF")]//Version 1  
public class Class1 : SchemaClass  
{ }  
  
Version 3:  
[SchemaAttribute("MySchema",  
    "5213D60D-D2F7-4E1B-8FA3-1C35227A40E4", //Version 3 -> current version  
    "261177F2-0475-4B0D-B458-DF5609A330F7", //Version 2  
    "1E578722-BFF5-4496-B436-CB6C659D93DF")]//Version 1  
public class Class1 : SchemaClass  
{ }
```

Remarks:

- If version of the class A changes and there is property of this type in the class B, the version of the class B should be upgraded – see subschemas in Revit Extensible Storage
- Based on history: data is searched in the element, loaded version is calculated etc.
- When saving data to the element older data is removed.
- If there are simple changes made in the class (e.g. new property added) there is no need to take any special action to support old data. In case of major changes (e.g. complete class reorganization) there is a set of helper methods which may be overridden in order to support such situation: LoadField, PreLoad, PostLoad – they were described before in the document.
- The version number of a schema may be taken using `GetVersion` method.

Code region

```

Version 1:
[Schema("MySchema",
    "1E578722-BFF5-4496-B436-CB6C659D93DF")]//Version 1
public class Class1 : SchemaClass
{
    [SchemaProperty(
        Unit = UnitType.UT_Length,
        DisplayUnit = DisplayUnitType.DUT_METERS)]
    public double a{ get; set; }
}

Version 2:
[Schema("MySchema",
    "261177F2-0475-4B0D-B458-DF5609A330F7", //Version 2
    "1E578722-BFF5-4496-B436-CB6C659D93DF")]//Version 1
public class Class1 : SchemaClass
{
    [SchemaProperty(
        Unit = UnitType.UT_Length,
        DisplayUnit = DisplayUnitType.DUT_METERS)]
    public double b{ get; set; }

    public override bool LoadField(Guid loaded, Entity entity, Field field)
    {
        if (GetVersion(loaded) < 2 && field.FieldName == "a")
        {
            b = entity.Get<double>("a", DisplayUnitType.DUT_METERS);
            return true;
        }
        return false;
    }
}

```

As we see the field “a” was replaced with the field “b”. In case of older version loaded there has to be taken manual action in the `LoadField` method which will assign value stored in “a” to new container “b”.

2.3.4 SchemaPropertyAttribute

`SchemaPropertyAttribute` indicates that specific property of the class should be put into the schema. It is derived from `UnitAttribute` class which provides following data:

- `Unit` – The unit category of the field (e.g. `UT_Length`)
- `DisplayUnit` – The unit in which data is stored in the class (important during saving and loading)

Code region

```
[SchemaProperty(  
    DisplayUnit = DisplayUnitType.DUT_METERS,  
    Unit = UnitType.UT_Length)]  
public double A { get; set; }
```

Additionally the user may define:

- **FieldName** - The name of the field in the schema. If not set the name of the property will be taken directly.

Code region

```
[SchemaProperty(FieldName = "B")]  
public string ValueForSchema { get; set; }
```

- **FieldType** – Revit Extensible Storage is restricted only to some specific types. If the user type is not included, he or she can specify the type which should be used for schema creation (this type has to be from set of types supported by Extensible Storage):

Code region

```
enum Option  
{  
    option1,  
    option2  
}  
  
[SchemaProperty(FieldType = typeof(int))]  
public Option B { get; set; }
```

- **ConverterType** – If some specific conversion between registered type and class type is required, it is sometimes necessary to provide a **ConverterType** which implements **IFieldToPropertyConverter**. Below you can find an example to support **DateTime** type.

Code region

```
public class DateTimeConverter : IFieldToPropertyConverter
{
    #region IFieldToPropertyConverter Members

    public object ConvertEntityFieldValToPropertyVal(object entityFieldValue)
    {
        IList<int> loaded = entityFieldValue as IList<int>;
        return new DateTime(loaded[0], loaded[1], loaded[2]);
    }

    public object ConvertPropertyValToEntityFieldVal(object propertyValue)
    {
        DateTime dt = (DateTime)propertyValue;
        return new List<int>() { dt.Year, dt.Month, dt.Day };
    }

    #endregion
}

[SchemaProperty(
    FieldType = typeof(List<int>),
    ConverterType = typeof(DateTimeConverter))]
public DateTime Date { get; set; }
```

Note: Another option could be to provide a special serialization method for this field:

Code region

```
[SchemaProperty(FieldType = typeof(List<int>))]
public DateTime Date2 { get; set; }

public override bool LoadField(Guid loaded, Entity entity, Field field)
{
    if (field.FieldName == "Date2")
    {
        IList<int> list = entity.Get<IList<int>>(field);
        Date2 = new DateTime(list[0], list[1], list[2]);
        return true;
    }

    return false;
}

public override bool SaveField(Entity entity, Field field)
{
    if (field.FieldName == "Date2")
    {
        entity.Set<IList<int>>(field, new List<int>()
            { Date2.Year, Date2.Month, Date2.Day });
        return true;
    }

    return false;
}
```

2.3.5 Subschemas

Subschemas are created automatically. The only thing that has to be done is marking the field with SchemaPropertyAttribute:

Code region

```
[SchemaProperty]
public MySchema SubSchema { get; set; }
```

If the version of a subschema changes the version of main schema (which includes subschema) has to be changed as well.

2.3.6 Arrays and maps

Arrays and maps are detected and converted automatically. The only restriction is that arrays has to implement `IList` and maps `IDictionary` interfaces:

Code region

```
[SchemaProperty(UnitType.UT_Length, DisplayUnitType.DUT_METERS)]
public List<double> DoubleList { get; set; }
```

2.3.7 Revit elements and enum types

In case of Revit elements SchemaClass converts them automatically to ElementIds:

Code region

```
[SchemaProperty()]
public Autodesk.Revit.DB.FamilySymbol FamInstance { get; set; }
```

In case of enum types SchemaClass converts it automatically to Integer

Code region

```
[SchemaProperty()]
public Option MyOption { get; set; }
```

2.3.8 Example

Code region

```
[Schema("MySchemaClass1", "31d2bb38-1b1b-40e3-a535-ae201279b6e3")]
public class MySchemaClass1 : SchemaClass
{
    [SchemaPropertyAttribute()]
    public String Name1 { get; set; }
}

Autodesk.Revit.DB.Element el = ...;

//loading from element
MySchemaClass1 instance = new MySchemaClass1();
instance.Load(el);

//modification
instance.Name1 = "x";

//saving data to element
instance.Save(el);
```

2.4 UI

Component: ExtensibleStorageFramework.UI.dll

Namespace: Autodesk.Revit.UI.ExtensibleStorage.Framework

Extensible Storage Framework provides own UI layer with own definition of layout structure. It is technology independent and the user is isolated from implementation. Note that currently WPF technology is used.

The layout represents the schema in UI context. Each field has assigned control which allows Revit users to modify it. Additionally controls may be grouped in categories.



Except Field Controls there are following controls that can be added to layout:

- Image – represents image
- TextBlock – represents text
- SubSchemaControl – represents control which enables to launch additional dialog with SubSchema

2.4.1 Layout creation

The layout may be defined in three ways:

- Manually

Code region

```
Layout layout = new Layout();
Category category = new Category() { Name = "Category1" };
layout.Controls.Add(category);

FieldControl fieldControl = new FieldControl(schemaID, "a", "a", new TextBoxAttribute());
category.Controls.Add(fieldControl);
```

- Using attributes (recommended)

Code region

```
public class MySchemaClass1 : SchemaClass
{
    [TextBoxAttribute(Category = "Category1")]
    public String a { get; set; }
}

Layout layout = Layout.Build(typeof(MySchemaClass1), serverUI);
```

- Mixed: using attributes and manually

Code region

```
public class MySchemaClass1 : SchemaClass
{
    [TextBoxAttribute(Category = "Category1")]
    public String a { get; set; }
}

Layout layout = Layout.Generate(typeof(MySchemaClass1), serverUI);

Category category = layout.GetCategory("Category1");
Image img = new Image() { Source = new Uri("C:\\image.png") };
category.Controls.Add(img);
```

When the whole structure of the layout is defined, it is possible to get final control using WPF.

Code region

```
ILayoutControl layoutControl = Layout.BuildControl(serverUI, doc, layout, obj);
dockPanel.Children.Add(layoutControl as System.Windows.UIElement);
```

The whole code would look as follows:

Code region

```

Class definition:
public class MySchemaClass1 : SchemaClass
{
    [TextBoxAttribute(Category = "Category1")]
    public String a { get; set; }
}

Layout structure definition:
Layout layout = Layout.Build(typeof(MySchemaClass1), serverUI);

The instance of object which will be presented
MySchemaClass1 obj = new MySchemaClass1();

Creation of the control:
ILayoutControl layoutControl = Layout.BuildControl(serverUI, doc, layout, obj);

Embedding control in own dialog:
dockPanel.Children.Add(layoutControl as System.Windows.UIElement);

```

2.4.2 UI attributes

On this section are listed all available attributes for the set of control exposed by the Extensible Storage Framework. Some of these attributes are common and some others are specific to control. When some controls derived from other, further in this document are listed only new attributes.

2.4.2.1 Common attributes

Each field control consists of two elements:

- A description
- An edit control



Below is the list of common attributes for each field control.

Attributes	Description
Visible	Indicates if the field is visible.
Category	The category the field should be assigned to.
.IsEnabled	Indicates if the edit control should be enabled.
Localizable	Indicates if the field is localizable. If true, GetResource method from IServerUI will be used (e.g. for description, tooltip etc.).

Description	The description of the field. If the description is not set, the name from the schema will be taken. The description supports the localizable flag and indexes and symbols could be used in the same way as for the documentation, e.g.: Description =" Val{1} ". 
Index	Decides about position of the field on the final layout.
Tooltip	The tooltip for particular field. The tooltip supports the localizable flag.

2.4.2.2 Format controls

- As format controls are taken all controls where content is a text which needs to be formatted.

Attributes	Description
FieldFormat	Defines the type of formatter. By defining a formatter, users may customize control formatting. In this case, the type has to implement the IFieldFormat interface.

Note: Base format controls (**TextBoxAttribute**, **ComboBoxAttribute**, etc.) don't assign by default a formatter which fits to the type of data.

If the formatter is empty, .NET conversion between objects and strings will be applied. If it is not the expected behavior, **FieldFormat** should be assigned manually. It may be taken from the provided set of formatters in the Extensible Storage Framework or it could be a custom one created by hand.

However, the best way is to use a dedicated attribute, e.g. **XYZTextBoxAttribute** instead of **TextBoxAttribute** with **XYZFieldFormat**.

Below is an example of formatter created by hands:

Code region

```

FieldFormat = typeof(ValueFormat);

public class ValueFormat: IFieldFormat
{
  public string Format(object value, ProjectUnit projectUnit, UnitType unit)
  {
    return "<" + value.ToString() + ">";
  }

  public object Parse(string value, ProjectUnit projectUnit, UnitType unit)
  {
    value = value.Replace("<", "");
    value = value.Replace(">", "");

    return Convert.ToDouble(value);
  }
}
  
```

2.4.2.2.1 TextBoxAttribute

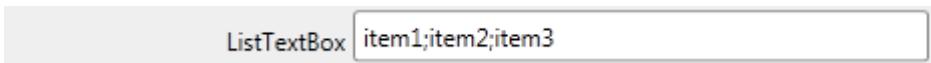
- Indicates that the field should be presented as a **TextBox**

Attributes	Description
FieldValidator	Defines the type of validator. By defining a validator, users may provide their own validation rules. In this case, the type has to implement the <code>IFieldValidator</code> interface.
Code region	
<pre>FieldValidator = <code>typeof(ValueValidator)</code> public class ValueValidator: <code>IFieldValidator</code> { public bool ValidateValue(<code>object</code> entity, <code>string</code> field, <code>object</code> value, <code>DisplayUnitType</code> unit) { if (field == "Val1") { return (<code>double</code>)value > 0; } return true; } }</pre>	

Code region
<pre>[<code>TextBox</code>] public <code>string</code> TextBox { <code>get</code>; <code>set</code>; }</pre> 

2.4.2.2.2 ListTextBoxAttribute

- Indicates that the field should be presented as a list `TextBox`
- Derived from `TextBoxAttribute`

Code region
<pre>[<code>ListTextBox</code>] public <code>List<string></code> ListTextBox { <code>get</code>; <code>set</code>; }</pre> 

2.4.2.2.3 UnitTextBoxAttribute

- Indicates that the field should be presented as `TextBox` with units.
- Derived from `TextBoxAttribute`

Attributes	Description
AttributeUnit	Defines unit type in which attribute values are defined (e.g. MaximumValue).
MinimumValue	The minimum value.
MaximumValue	The maximum value.
ValidateMinimumValue	Indicates if value should be checked against MinimumValue.
ValidateMaximumValue	Indicates if value should be checked against MaximumValue.
FieldValidator	Defines the type of validator. By defining it the developer may provide his or her own validation rules. The type has to implement the IFieldValidator interface. (see TextBox).

Code region

```
[UnitTextBox]
public double UnitTextBox { get; set; }
```

UnitTextBox 1' - 0"

2.4.2.2.4 ListUnitTextBoxAttribute

- Indicates that the field should be presented as a list TextBox with units
- Derived from **UnitTextBoxAttribute**

Attributes	Description
MaximumItemCount	Maximum number of items on the list.
MinimumItemCount	Minimum number of items on the list.
MinimumItemCountCheck	Indicates if maximum number of items should be constrained.
MaximumItemCountCheck	Indicates if minimum number of items should be constrained.

Code region

```
[ListUnitTextBox]
public List<double> ListUnitTextBox { get; set; }
```

ListUnitTextBox 1' 0";2' 0";4' 0"

2.4.2.2.5 NumericUpDownAttribute

- Derived from **TextBoxAttribute**
- Base attribute for **NumericUpDown** attributes

Attributes	Description
MinimumValue	The minimum value.
MaximumValue	The maximum value.
Step	The step.

2.4.2.2.6 IntNumericUpDownAttribute

- Indicates that the field should be presented as `NumericUpDown` control for integer values
- Derived from `NumericUpDownAttribute`

Code region

```
[IntNumericUpDown(0, 100, 2)]
public int NumUpDownInt { get; set; }
```



2.4.2.2.7 UnitNumericUpDownAttribute

- Indicates that the field should be presented as `NumericUpDown` control with units.
- Derived from `NumericUpDownAttribute`

Attributes	Description
<code>AttributeUnit</code>	Defines unit type in which attribute values are defined (e.g. <code>MaximumValue</code>).

Code region

```
[UnitNumericUpDown(0, 100, 10, DisplayUnitType.DUT_SQUARE_METERS)]
public double NumUpDownUnit { get; set; }
```



2.4.2.3 Data source controls

- Data source controls are controls where available values are set predefined (e.g. combobox items).

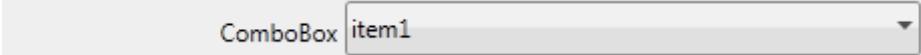
Attributes	Description
<code>Items</code>	List of items provided directly.
<code>DataSourceKey</code>	The key based on which the source will be taken from <code>IServerUI</code> (using <code>GetDataSource</code> method). The priority is set to <code>Items</code> property.

2.4.2.3.1 ComboBoxAttribute

- Indicates that particular field should be edited with `ComboBoxUnit` control. Used for single selection.

Code region

```
[ComboBox("item1", "item2")]
public string ComboBox { get; set; }
```

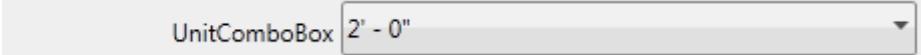


2.4.2.3.2 UnitComboBoxAttribute

- Indicates that particular field should be edited with `UnitComboBox` control. Used for single selection.
- Derived from `ComboBoxAttribute`

Code region

```
[UnitComboBox("comboDataSource")]
public double UnitComboBox { get; set; }
```



2.4.2.3.3 ElementComboBoxAttribute

- Indicates that particular field should be edited with `ElementComboBox` control. Used for single selection.
- Derived from `ComboBoxAttribute`

Code region

```
[ElementComboBox("comboDataSource2")]
public FamilySymbol ElementComboBox { get; set; }
```



2.4.2.3.4 CheckedListAttribute

- Indicates that particular field should be edited with `CheckedList` control. Used for multiply selection.

Code region

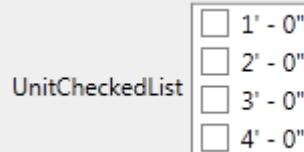
```
[CheckedList("number 1", "number 2", "number 3")]
public List<string> CheckedList { get; set; }
```

**2.4.2.3.5 UnitCheckedListAttribute**

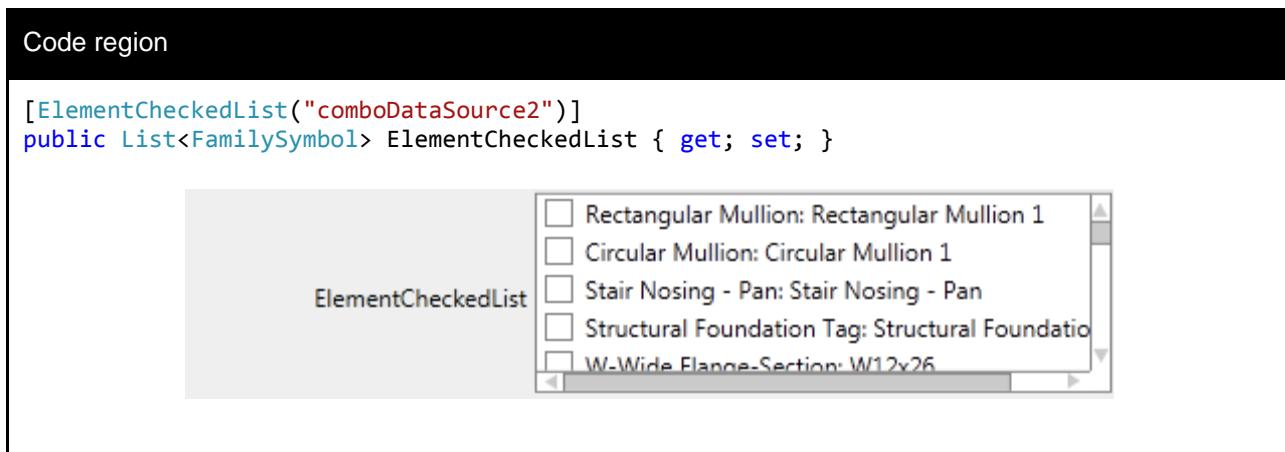
- Indicates that particular field should be edited with `UnitCheckedList` control. Used for multiple selection.
- Derived from `CheckedListAttribute`

Code region

```
[UnitCheckedList("comboDataSource")]
public List<double> UnitCheckedList { get; set; }
```

**2.4.2.3.6 ElementCheckedListAttribute**

- Indicates that particular field should be edited with `ElementCheckedList` control. Used for multiply selection.
- Derived from `CheckedListAttribute`



2.4.2.4 Grid controls

- Grid controls are assigned to maps and arrays.
- They are presented as GridViews on layout

Attributes	Description
AllowToAddElements	If items can be added to grid.
AllowToRemoveElements	If items can be removed from grid.
DefaultValue	The value set with new items.
AttributeUnit	Describes the unit of attribute.
DefaultValueProvider	The provider of default value for new items. It has to implement IDefaultValueProvider interface.

There are following grids attributes provided:

- GridCheckBoxAttribute
- GridComboBoxAttribute
- GridElementComboBoxAttribute
- GridEnumControlAttribute
- GridIntNumericUpDownAttribute
- GridTextBoxAttribute
- GridUnitNumericUpDownAttribute
- GridUnitTextBoxAttribute

They have the same attributes as their equivalents for simple fields so please refer to them for details

- GridCheckBoxAttribute – CheckBoxAttribute
- GridComboBoxAttribute – ComboBoxAttribute
- GridElementComboBoxAttribute – ElementComboBoxAttribute
- GridEnumControlAttribute – EnumControlAttribute
- GridIntNumericUpDownAttribute – NumericUpDownAttribute
- GridTextBoxAttribute – TextBoxAttribute
- GridUnitNumericUpDownAttribute – UnitNumericUpDownAttribute
- GridUnitTextBoxAttribute – UnitTextBoxAttribute

Code region

```
[GridUnitTextBox(Category = "Category",
    AllowToAddElements = true,
    AllowToRemoveElements = true)]
public List<double> DoubleList { get; set; }
```

DoubleList

2	32 SF
3	32 SF
4	32 SF
5	32 SF
6	32 SF
7	32 SF

For dictionaries there are additional attributes for keys:

- GridKeyComboBoxAttribute
- GridKeyTextBoxAttribute

Code region

```
[GridKeyTextBox(Category = "Category",
    DefaultValue = "a",
    AllowToAddElements = true,
    AllowToRemoveElements = true,
    Index = 1)]
[GridUnitTextBox(Category = "Category",
    DefaultValue = 1,
    DefaultValueUnit = DisplayUnitType.DUT_SQUARE_METERS,
    AllowToAddElements = true,
    AllowToRemoveElements = true,
    Index = 1)]
public Dictionary<string, double> DoubleListDict { get; set; }
```

DoubleListDict

a	11 SF
b	11 SF
c	11 SF
d	11 SF

If default value defined in attribute is not enough the user can use `DefaultValueProvider`. In that case a new class which implements `IDefaultValueProvider` interface has to be created. This class will be used by the Framework to set appropriate values when needed. Below is an example of implementation:

Code region

```
public class DefaultValueProvider : IDefaultValueProvider
{
    public object GetDefaultValue(object sender, DefaultValueEventArgs e)
    {
        return new XYZ(1, 1, 1);
    }
}

[GridXYZTextBox(DefaultValueProvider = typeof(DefaultValueProvider))]
public List<XYZ> Points { get; set; }
```

2.4.2.5 Other Controls

2.4.2.5.1 DefaultFieldControlAttribute

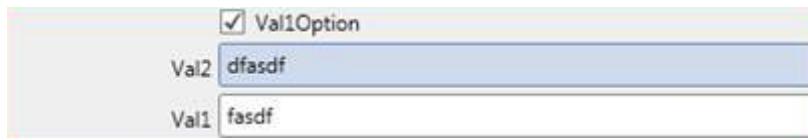
If the field is matched with `DefaultFieldControlAttribute` the system will select appropriate one based on type:

Code region

```
[Schema ("MySchemaClass1", "d87317ab-9d96-4395-b876-7c0f230af813")]
public class MySchemaClass1 : SchemaClass
{
    [DefaultFieldControl]
    public string Val1 { get; set; }

    [DefaultFieldControl]
    public string Val2 { get; set; }

    [DefaultFieldControl]
    public bool Val1Option { get; set; }
}
```



2.4.2.5.2 EnumControlAttribute

- Indicates that a particular field should be edited with `EnumControl` control. This control is provided for enum types.

Attributes	Description
EnumType	The enum type.
Presentation	Presentation mode allows developers to specify if the enum will be presented, as a ComboBox, an OptionList or a ToggleButton.
Item	Presentation item: Text, Image, Text with image.
ImageSize	The size of the image.

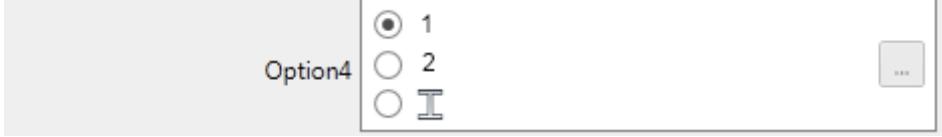
- Additionally it is possible to define buttons next to specific item which runs subschema dialog as shown below:

Code region

```
[EnumControl(typeof(Option),
    PresentationMode.ComboBox,
    PresentationItem.ImageWithText,
    Description = "fieldName")]
public Option Option2 { get; set; }
```




```
[EnumControl(typeof(Option),
    PresentationMode.OptionList,
    PresentationItem.Image, new string[] { "SubLabel2" },
    new object[] { Option.option2 })]
public Option Option4 { get; set; }
```




```
[EnumControl(typeof(Option),
    PresentationMode.ToggleButton,
    PresentationItem.ImageWithText)]
public Option Option8 { get; set; }
```



2.4.2.5.3 CheckBoxAttribute

- Indicates that particular field should be edited with CheckBox control.

Code region

```
[CheckBox()]
public bool CheckBox { get; set; }
```

A screenshot of a user interface showing a checkbox labeled "CheckBox". The checkbox is currently unchecked.

2.4.2.5.4 CategoryCheckBoxAttribute

- Binds category checkbox with property which reflects in enable state of whole category

Code region

```
[CategoryCheckBox("Category")]
public bool CategoryEnable { get; set; }
```

A screenshot of a user interface showing two categories. The first category, "Category", has its checkbox checked, and its properties are displayed: Val4 (0 SF), Val3 (<0>), Length (0), and Val2 (22 SF). The second category, also labeled "Category", has its checkbox unchecked, and its properties are displayed: Val4 (0 SF), Val3 (<0>), Length (0), and Val2 (22 SF).

2.4.2.5.5 SubSchemaControlAttribute

- Indicates that particular field should be edited with SubSchema control:

Attributes	Description
Text	The text which is next to the button.

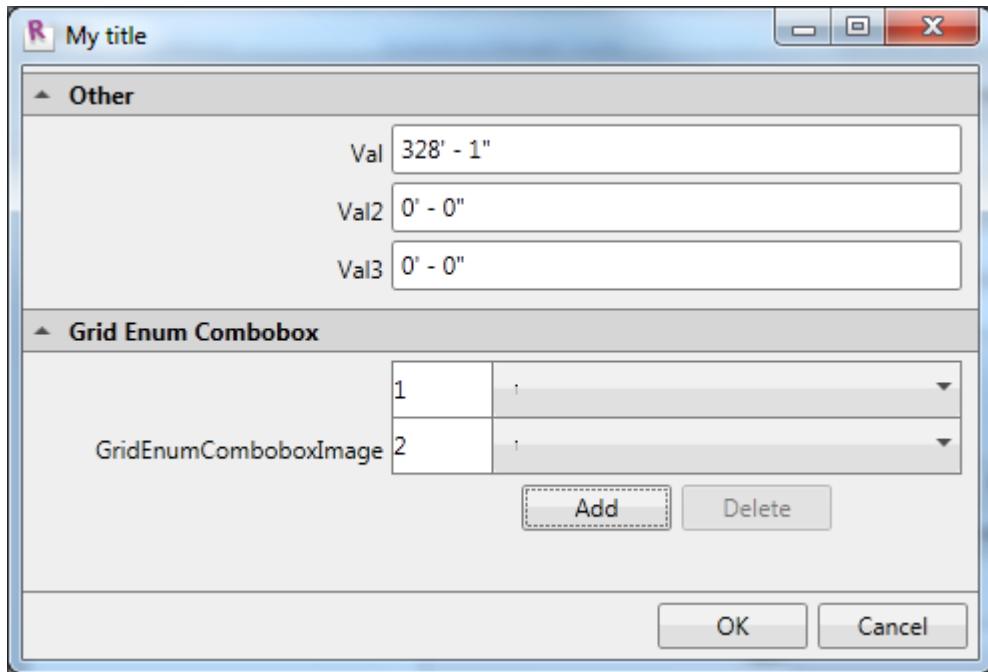
Code region

```
[SubSchemaControl(Text = "Here is my text")]
public MySubLabel SubLabel1 { get; set; }
```

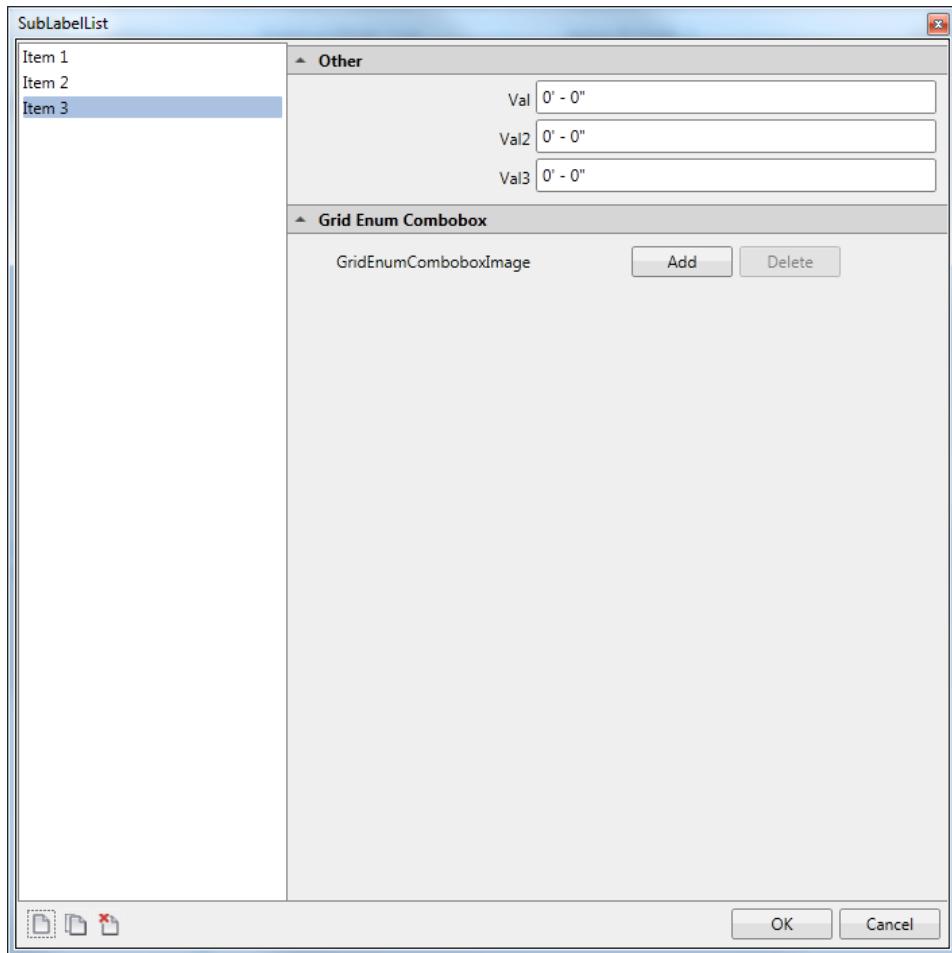
SubLabel1 Here is my text 

After clicking the button, an additional dialog is showed were the Revit user can set data for sub element.
Sub elements could be:

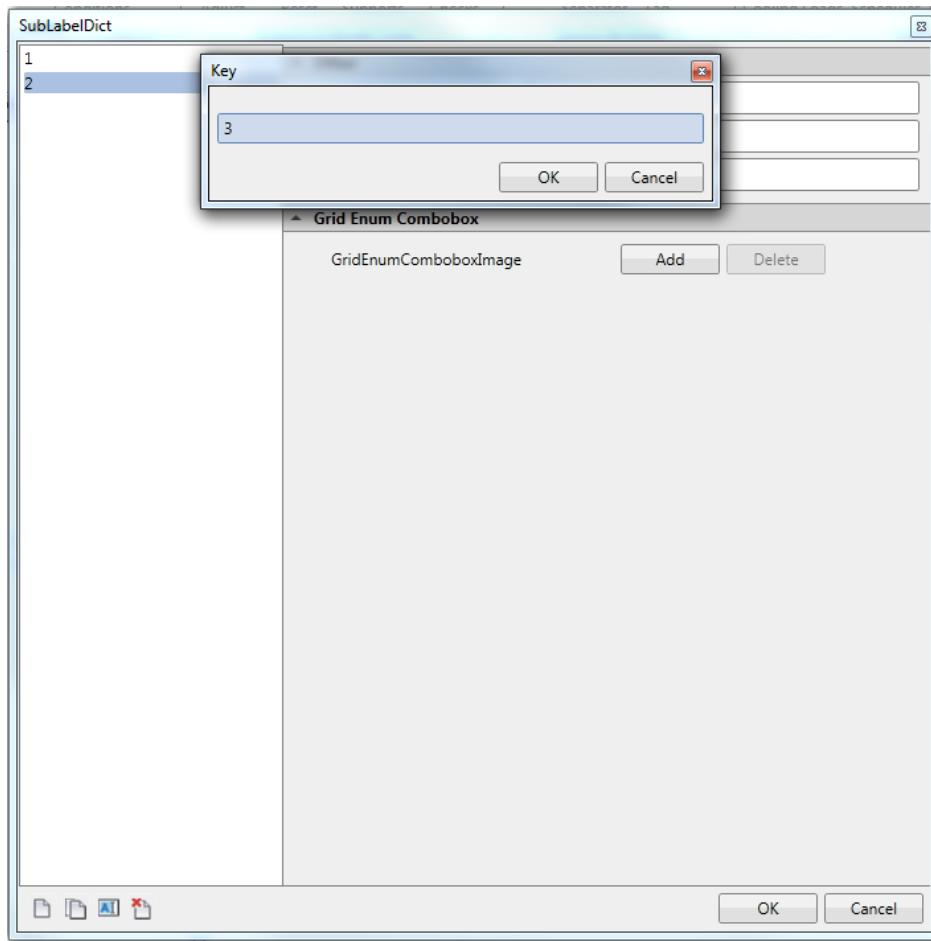
- A schema



- A list of schemas



- A dictionary of schemas



2.4.2.5.6 SubSchemaListTableAttribute

- Indicates that particular list of subschemas should be presented as a table
- Dedicated to lists of subschemas only

Attributes	Description
AllowToAddElements	If items can be added to grid.
AllowToRemoveElements	If items can be removed from grid.
MaxHeight	Maximum height of the table.

Code region

```
[Schema("MySchemaClass2", "C3ABDF1B-310F-4D86-9E04-29F2CC8EFAB1")]
public class MySchemaClass2 : SchemaClass
{
    [CheckBox()]
    public Boolean CheckBox { get; set; }

    [ComboBox("item1", "item2", "item3")]
    public String Selection { get; set; }

    [UnitTextBox()]
    [Unit(DisplayUnit= DisplayUnitType.DUT_METERS, Unit = UnitType.UT_Length)]
    public Double Value { get; set; }

    [TextBox()]
    public String Text { get; set; }

    public MySchemaClass2()
    {
        Selection = "item1";
        Value = 2;
        Text = "Name";
    }
}

[Schema("MySchemaClass1", "ec708e62-eebd-4ddf-9942-c3221913f6a3")]
public class MySchemaClass1 : SchemaClass
{
    [SubSchemaListTable()]
    public List<MySchemaClass2> Table { get; set; }

    public MySchemaClass1()
    {
        Table = new List<MySchemaClass2>();
    }
}
```

	Value	Text	CheckBox	Selection
Table	6' - 6 3/4"	Name	<input type="checkbox"/>	item1
	6' - 6 3/4"	Name	<input type="checkbox"/>	item1
	6' - 6 3/4"	Name	<input type="checkbox"/>	item1
	6' - 6 3/4"	Name	<input type="checkbox"/>	item1

Add Delete

2.4.2.5.7 SubSchemaEmbeddedControlAttribute

- Indicates that a particular subschema should be embedded in a parent schema layout

Code region

```
[Schema ("LabelSteel", "2174007A-EE52-4477-9181-FF53F3EEC1DC")]
public class MyLabelSteel : SchemaClass
{
    [SubSchemaEmbeddedControl(Category = "Category")]
    public MySchemaClass1 OwnDataEmbedded { get; set; }

    [TextBox(Category = "Category")]
    public double SteelVal4 { get; set; }

    [TextBox(Category = "Category")]
    public double SteelVal5 { get; set; }

}

[Schema ("MySchemaClass1", "d87317ab-9d96-4395-b876-7c0f230af813")]
public class MySchemaClass1 : SchemaClass
{
    [DefaultFieldControl]
    public string Val1 { get; set; }

    [DefaultFieldControl]
    public string Val2 { get; set; }

    [DefaultFieldControl]
    public bool Val1Option { get; set; }
}
```



2.4.2.5.8 CategoriesAttribute

- Allows the user to define categories at the class level and its position in layout:

Code region

```
[Categories(new string[] { "name2", "name1", "name3" }, new int[] { 2, 1, 3 })]
public class Class1 : SchemaClass
{
}
```

The screenshot shows a user interface for managing schema elements. At the top, there is a code block defining a class named Class1 that extends SchemaClass. The class has three properties: name1, name2, and name3. Below the code, a tree view displays these elements. Each element has a corresponding control: name1 has two controls labeled 'RefVal2' and 'RefValNew' both containing '0' - '0"'; name2 has one control labeled 'RefVal1' containing '0' - '0"'; and name3 has one control labeled 'TextBox'.

2.4.3 IServerUI

In order to interact with layout users can provide the object which implements `IServerUI` interface:

Code region

```
public interface IServerUI
{
    IList GetDataSource(string key, Document document, DisplayUnitType unitType);
    string GetResource(string key);
    Uri GetResourceImage(string key);
    void LayoutInitialized(object sender, LayoutInitializedEventArgs e);
    void ValueChanged(object sender, ValueChangedEventArgs e);
}
```

Extensible Storage Framework uses this interface to inform the user about different actions (like `ValueChanged`) as well as to get some additional information (e.g. data source):

2.4.3.1 LayoutInitialized

- Informs the user that the layout was initialized with data.

LayoutInitializedEventArgs	Description
Document	The Revit document.
ISchemaEditor	Schema editor.
Entity	The current value of object described by layout.

Developers may take some actions (e.g. hide some controls, change image, etc.) during the layout initialization and this may be done via `ISchemaEditor` interface.

2.4.3.2 ValueChanged

- Informs the user that value for the field was changed.

ValueChangedEventArgs	Description
Document	The Revit document.
ISchemaEditor	Schema editor (see LayoutInitialized).
Entity	The current value of object described by layout.
FieldName	The name of changed field.
FieldEditor	The editor of the field (e.g. ITableEditor).
AdditionalInfo	Additional information about field (only for selected e.g. TableCellIdentification , ListItemIdentification , DictionaryItemIdentification).

Developers may take some actions as well here and modify current layout via [ISchemaEditor](#) object, below is an example of implementation:

Code region

```
public void ValueChanged(object sender, ValueChangedEventArgs e)
{
    if (e.FieldName == "Mode")
    {
        e.Editor.SetAttribute("Val1",
            FieldUIAttribute.PropertyVisible,
            true,
            DisplayUnitType.DUT_UNDEFINED);
    }
}
```

2.4.3.3 GetResource

- Returns the resource text based on key and context (used when [Localizable](#) flag is set to true and for enum controls)
- The usage of this method is highly recommended to develop localized applications.

Code region

```
public string GetResource(string key, string context)
{
    string txt = ExtensibleStorageFramework7.Texts.ResourceManager.GetString(key);

    if (!string.IsNullOrEmpty(txt))
        return txt;

    return key;
}
```

2.4.3.4 GetResourceImage

- Returns the image resource based on key and context.

Code region

```
public Uri GetResourceImage(string key , string context)
{
    if (key == "key1")
        return new Uri(@"pack://application:,,,/MyAssembly;component/bmp_1.png");
    else if (key == "key2")
        return new Uri(@"pack://application:,,,/ MyAssembly;component/bmp_2.png");
}
```

2.4.3.5 GetDataSource

- Returns the data source based on specific key. This method is used by controls that support DataSource (e.g. ComboBoxAttribute, CheckedListAttribute, etc.).

Code region

```
public IList GetDataSource(string key, Document document, DisplayUnitType unitType)
{
    if (key == "comboDataSource")
    {
        List<double> vals = new List<double>() { 1, 2, 3, 4 };
        return vals;
    }
    else if (key == "comboDataSource2")
    {
        FilteredElementCollector coll = new FilteredElementCollector(document);
        IList<Element> elems = coll.OfClass(typeof(FamilySymbol)).ToElements();
        return elems.ToList();
    }

    return null;
}
```

2.4.4 ISchemaEditor

`ISchemaEditor` allows the user to communicate with layout. Following methods are available:

2.4.4.1 SetAttribute

- Sets particular attribute to the field.
- Names for attributes are defined as constant strings in attribute classes e.g.: `FieldUIAttribute.PropertyVisible`.

2.4.4.2 SetCategoryAttribute

- Sets category attribute (visibility only with current version).

2.4.4.3 SetImageSourceAttribute

- Sets image source attribute.

2.4.4.4 SetValue

- Sets value to specific field

2.4.4.5 GetValue

- Gets value of specific field

2.4.4.6 GetFieldEditor

- Gets editor for specific field. It can be the general or a dedicated one (e.g. ITableEditor)

2.4.5 Layout data initialization

There are two ways of for the layout data initialization:

- Based on Revit entity

Code region

```
//creation instance
MySchemaClass1 instance = new MySchemaClass1();
//definition of virtual structure
Layout structure = Layout.Build(typeof(MySchemaClass1), this);

//create wpf control
ILayoutControl layoutCtr = Layout.BuildControl(this,
revitDocument,structure,instance.GetEntity());

//getting data
Entity entity = layoutCtr.GetEntity();

//setting data
instance.SetProperties(entity);
```

- Based on instance of .NET class

Code region

```
//creation instance
MySchemaClass1 instance = new MySchemaClass1();

//definition of virtual structure
Layout structure = Layout.Build(typeof(MySchemaClass1), this);

//create control
ILayoutControl layoutCtr = Layout.BuildControl(this, revitDocument, structure, instance);
```

Note:

- Instance is modified by the layout
- If there are lists or dictionaries with user types (subschemas), they should implement ICloneable interface and contain empty constructor (SchemaClass provides cloning automatically)

2.4.6 Example

Code region

```
[Schema("MySchemaClass1", "ec708e62-eebd-4ddf-9942-c3221913f6a3")]
public class MySchemaClass1 : SchemaClass
{
    [SchemaProperty(Unit= UnitType.UT_Length, DisplayUnit=DisplayUnitType.DUT_METERS)]
    [UnitTextBox]
    public Double b{get;set;}

    [SchemaProperty()]
    [TextBox]
    public String a{get;set;}

    public MySchemaClass1()
    {
    }
}

ServerUI serverUI = new ServerUI();
MySchemaClass1 obj = new MySchemaClass1();
Layout layout = Layout.Build(typeof(MySchemaClass1), serverUI);
ILayoutControl layoutControl = Layout.BuildControl(serverUI, revitDocument, layout, obj);
window.dockPanel.Children.Add(layoutControl as System.Windows.UIElement);
Layout.ApplyStyle(window);

window.ShowDialog();
```



2.5 Documentation

Component: ExtensibleStorageFramework.Documentation.dll

Namespace: Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation

Extensible Storage Framework provides own documentation layer with own definition of document structure.

Following output formats are supported:

- Html
- Mht
- Txt

2.5.1 Document Body

The document body consists of a list of Document elements. Below is the list of elements that could be added to this list:

2.5.1.1 DocumentText

- Represents a simple text.

Parameters	Description
Text	The text to display.

Code region

```
body.Elements.Add(new DocumentText("This is text"));
```

This is text

Note:

- In all elements (it doesn't concern DocumentText only) you can use indexes in the following way

Code region

```
body.Elements.Add(new DocumentText("text{index}"));
```

text_{index}

```
body.Elements.Add(new DocumentText("text{^index}"));
```

text^{index}

- It is also possible to insert symbol:

Code region

```
body.Elements.Add(new DocumentText("@a"));
```

a

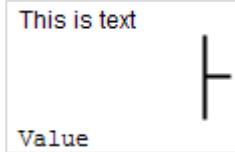
2.5.1.2 DocumentLineBreak

- Represents a line break.

Parameters	Description
Count	The number of line breaks.

Code region

```
body.Elements.Add(new DocumentLineBreak(4));
```



2.5.1.3 DocumentValue

- Represents a value

Parameters	Description
Value	The field value to display.

Code region

```
body.Elements.Add(new DocumentValue("Value"));
```

Value

2.5.1.4 DocumentValueWithName

- Represents a value with a name

Parameters	Description
Name	Field name to display.
Value	Field value to display.

Code region

```
body.Elements.Add(new DocumentValueWithName("FieldName",1));
```

FieldName 1

2.5.1.5 DocumentValueWithDescription

- Represents a value with a name, a description, a placeholder for units and for an additional note.

Parameters	Description
Name	The field name to display.
Value	The field value to display.
Unit	The unit symbol to display.
Description	The description to display.
Note	The note to display (e.g. to some section in code).

Code region

```
body.Elements.Add(new DocumentValueWithDescription("FieldName", 1,"m","This is my value",
"Note 1.2.3"));
```

FieldName = 1 [m] This is my value

Note 1.2.3

2.5.1.6 DocumentStatus

- Represents a status element

Parameter	Description
Name	The field name to display.
Value	The field value to display.
Condition	Description of condition the value meets.
Message	Additional information.
Correct	Flag to set if the value is correct or not.

Code region

```
body.Elements.Add(DocumentStatus("FieldName", "> 24", "value is correct", true, 25));
```

FieldName	> 24	value is correct	25
-----------	------	------------------	----

2.5.1.7 DocumentTable

- Represents a table

Parameters	Description
Cells	The two dimensional array of cells.
ColumnsCount	The number of columns.
RowsCount	The number of rows.
HeaderColumnsCount	The number of header columns.
HeaderRowsCount	The number of header rows.
PercentageColumnWidth	The percentages width of particular columns.

Code region

```
DocumentTable table = new DocumentTable(2, 2);

table[0, 0].Elements.Add(new DocumentText("txt1"));
table[0, 1].Elements.Add(new DocumentText("txt2"));
table[1, 0].Elements.Add(new DocumentText("txt3"));
table[1, 1].Elements.Add(new DocumentText("txt4"));

table.PercentageColumnWidth[0] = 10;
table.PercentageColumnWidth[1] = 90;

table.HeaderColumnsCount = 1;
table.HeaderRowsCount = 1;
body.Elements.Add(table);
```

txt1	txt2
txt3	txt4

2.5.1.8 DocumentAnchor

- Represents an anchor element.
- Its usage is strongly connected to DocumentAnchorReference.

Parameters	Description
Name	The name of anchor (used for identification).
Text	The text of anchor (may be empty).

2.5.1.9 DocumentAnchorReference

- Represents anchor reference. It allows to switch to anchor place automatically from the note.

Parameters	Description
Reference	The anchor reference.
Text	The link text .

Code region

```
DocumentAnchor anchor = new DocumentAnchor("anchor", "My anchor");
body.Elements.Add(anchor);
body.Elements.Add(new DocumentLineBreak());
body.Elements.Add(new DocumentAnchorReference(anchor, "Go to my anchor"));
```

My anchor
[Go to my anchor](#)

2.5.1.10 DocumentSection

- Represents section element. It allows to divide document into logic parts and generate automatically the table content:

Parameters	Description
Name	The name.
Title	The title.
Body	The body of section.

Code region

```
DocumentSection section = new DocumentSection("My section",4);
section.Body.Elements.Add(new DocumentText("Text inside section"));
body.Elements.Add(section);
```

MY SECTION
Text inside section

2.5.1.11 DocumentImage

- Represents an image

Parameters	Description
Source	The image source.

Code region

```
body.Elements.Add(new DocumentImage(new Uri(@"pack://application:,,,/
MyAssembly;component/bmp_Isection.png"));
```

**2.5.1.12 DocumentDiagram**

- Represents a diagram

Parameters	Description
Series	The list of series.
AxisX	The x axis.
AxisY	The y axis.
Title	The diagram title.
Width	The width of diagram.
Height	The height of diagram.
Legend	Indicates if legend should be exposed.

2.5.1.12.1 DocumentDiagramSeries

- Represents a diagram series.

Parameters	Description
Name	The name.
DiagramType	The type of diagram.
Color	The color of series.
Title	The diagram title.
Width	The width of border.

2.5.1.12.2 DocumentDiagramAxis

- Represents a diagram axis

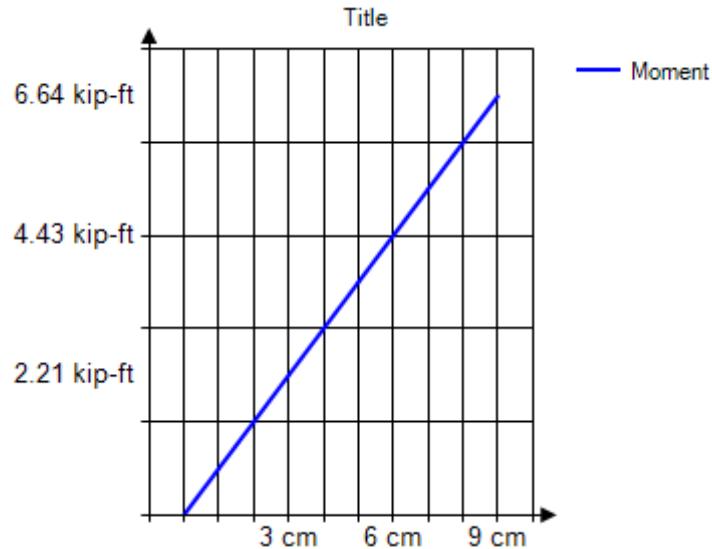
Parameters	Description
Title	The title of axis.
Labels	The list of axis labels.

Code region

```
DocumentDiagram diag = new DocumentDiagram("Title", "Moment");

for (int i = 0; i < 10; i++)
    diagram.Series[0].AddXY(i, i);

diag.SetLabelsX(UnitType.UT_Length, DisplayUnitType.DUT_CENTIMETERS, document, 3);
diag.SetLabelsY(UnitType.UT_Moment, DisplayUnitType.DUT_KILONEWTON_METERS, document, 3);
```



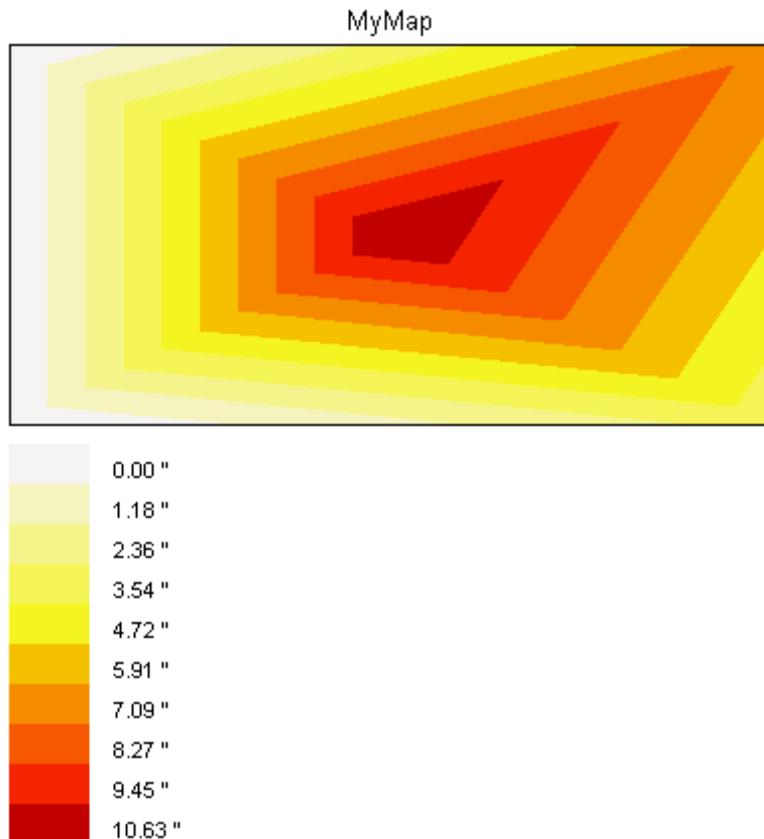
2.5.1.12.3 DocumentMap

- Represents a map to represent 2D results.

Parameters	Description
Series	The list of series.
DisplayUnitType	Unit data is defined in.
UnitType	The type of unit.
Title	The diagram title.
Width	The width of map.
Legend	Indicates if legend should be exposed.

Code region

```
DocumentMap map = new DocumentMap(UnitType.UT_Displacement_Deflection,  
DisplayUnitType.DUT_CENTIMETERS);  
//External contour  
map.AddPoint(0, 0, 0, true);  
map.AddPoint(2, 0, 10, true);  
map.AddPoint(2, 1, 20, true);  
map.AddPoint(0, 1, 0, true);  
//Internal points  
map.AddPoint(1, 0.5, 30);  
//Title  
map.Title = "MyMap";  
//Legend colors  
map.SetColors(10);  
body.Elements.Add(map);
```



2.5.2 Document element creation

The body may be defined in three ways:

- Manually

Code region

```
Document document = new Document();
document.Main.Elements.Add(new DocumentValue("Value"));
document.Main.Elements.Add(new DocumentValueWithName("A", "Value"));
document.Main.Elements.Add(new DocumentValueWithDescription("B", 1, "Value", "This is my
value", "Note 1.2.3));
```

- Using attributes

By setting attributes to appropriate properties in the class as follow:

Code region

```
//class
[Value()]
public String Val1 { get; set; }

[ValueWithName(Name = "A")]
public String Val2 { get; set; }

[ValueWithDescription(Description="This is my value", Note="Note 1.2.3", Name="B")]
public String Val3 { get; set; }

//method
Document document = new Document();
DocumentBody.FillBody(instance, document.Main, this, revitDocument);
```

- Mixed – using attributes and manually

Code region

```
//class
[Value()]
public String Val1 { get; set; }

[ValueWithName(Name = "A")]
public String Val2 { get; set; }

//method
Document document = new Document();
DocumentBody.FillBody(instance, document.Main, this, revitDocument);

document.Main.Elements.Add(new DocumentValueWithDescription("B", 1, "Value", "This is my
value", "Note 1.2.3));
```

2.5.3 Attributes

It is also possible to use attributes assigned to properties of classes. Such fields will be automatically converted to appropriate DocumentElements.

2.5.3.1 Common properties

Attributes	Description
Level	The detail level of element.
Localizable	Indicates if the field is localizable.

2.5.3.2 ValueAttribute

- Represents a value

Code region
<pre>[ValueAttribute()] public String Value{get;set;}</pre> <p style="text-align: center;">Value</p>

2.5.3.3 ValueWithNameAttribute

- Represents a value with name

Attributes	Description
Name	Field name.
NamePostfix	The text which will be added at the end of the name (default „:”).

Code region
<pre>[ValueWithNameAttribute(Name="FieldName")] public String Value{get;set;}</pre> <p style="text-align: center;">FieldName 1</p>

2.5.3.4 ValueWithDescriptionAttribute

- Represents a value with description

Attributes	Description
Name	Field name.
NamePostfix	The text which will be added at the end of the name (default „:”).
Description	Description.

Code region

```
[ValueWithDescriptionAttribute(Description="This is my value",Note="Note  
1.2.3",Name="FieldName")]  
public String FieldName{get;set;}
```

FieldName = 1 [m] This is my value

Note 1.2.3

2.5.3.5 ListValueAttribute

- Represents a list of values

Code region

```
[ListValueAttribute()]  
public List<String> MyField{get;set;}
```

MyField item1;item2;item3

2.5.3.6 TableAttribute

- Represents a table

Code region

```
[TableAttribute()]  
public List<String> MyField{get;set;}
```

MYFIELD

item1
item2
item3

2.5.3.7 RatioAttribute

- Represents the ratio element

Code region

```
[Ratio()]
public Double Ratio{get;set;}
```

Ratio	< 1	Succeeded	0.749
-------	-----	-----------	-------

2.5.4 SubSchemaListTableAttribute

- Represents list of subschema in table

Code region

```
[Schema("SubLabel", "CB0E232D-B362-45D9-B695-6E372C9D8413")]
public class MySubLabel:Autodesk.Revit.RAF.SchemaClass
{
    [DefaultElement]
    [SchemaProperty(UnitType.UT_Length, DisplayUnitType.DUT_METERS)]
    public double Val { get; set; }
    [DefaultElement]
    [SchemaProperty(UnitType.UT_Length, DisplayUnitType.DUT_METERS)]
    public double Val2 { get; set; }

    [DefaultElement]
    [SchemaProperty(UnitType.UT_Length, DisplayUnitType.DUT_METERS)]
    public double Val3 { get; set; }

    public MySubLabel()
    {
    }

    public MySubLabel(Entity entity, Document document)
        : base(entity,document)
    {
    }
}

[Schema("MySchemaClass1", "afca32ae-5eed-4c8f-9a1a-f2fb93df1a3")]
public class MySchemaClass1 : Autodesk.Revit.RAF.SchemaClass
{
    [SubSchemaListTable]
    [SchemaProperty()]
    public List<MySubLabel> SubLabelList2 { get; set; }
}
```

Val3	Val2	Val
1' - 0"	2' - 0"	3' - 0"
1' - 0"	2' - 0"	3' - 0"

2.5.5 Example

Code region

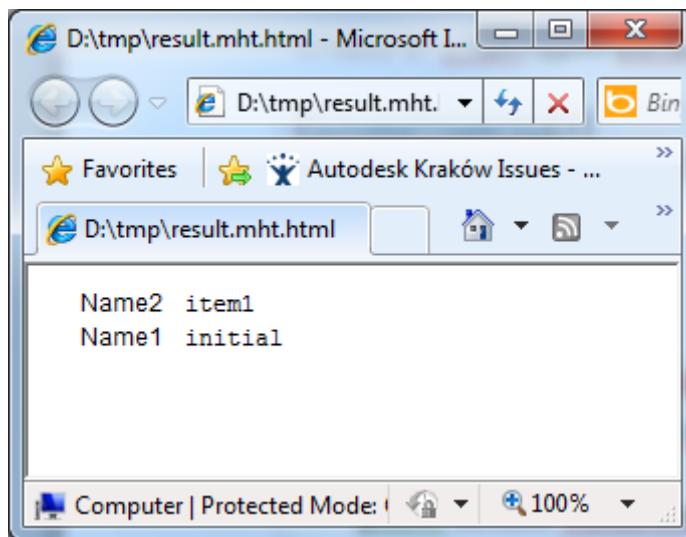
```
[Schema("MySchemaClass1", "31d2bb38-1b1b-40e3-a535-ae201279b6e3")]
public class MySchemaClass1 : SchemaClass
{
    [SchemaPropertyAttribute()]
    [ValueWithName()]
    public String Name2{get;set;}

    [SchemaPropertyAttribute()]
    [ValueWithName()]
    public String Name1{get;set;}

    public MySchemaClass1()
    {
        Name1 = "initial";
        Name2 = "item1";
    }

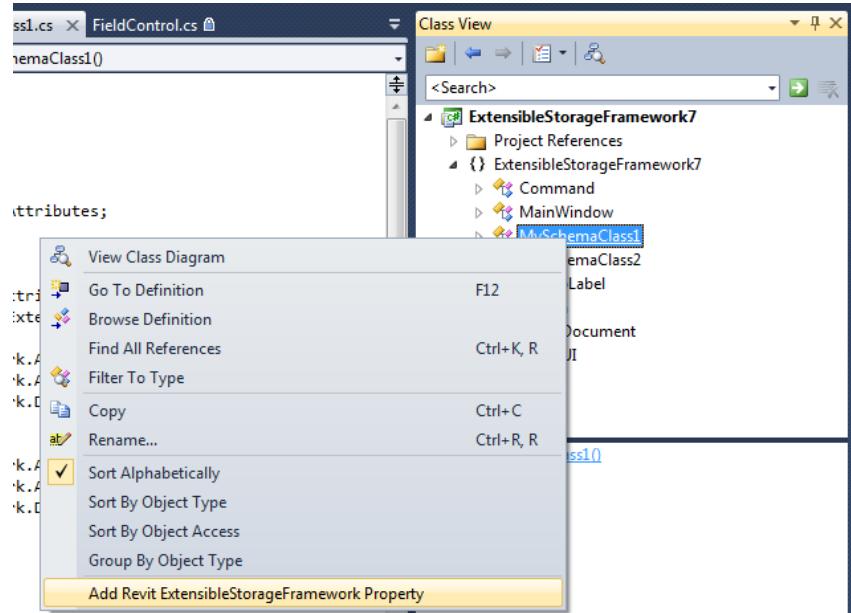
    public MySchemaClass1(Entity entity, Document document)
        : base(entity, document)
    {
    }
}

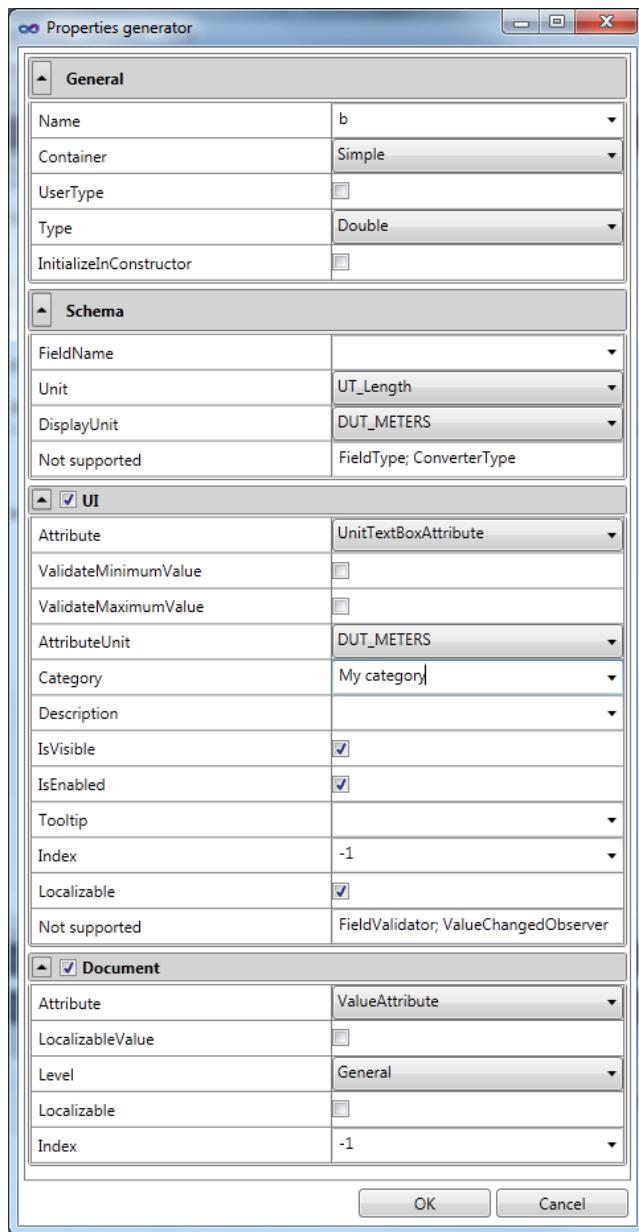
//creation instance
MySchemaClass1 instance = new MySchemaClass1();
//Create document
Document document = new Document();
DocumentBody.FillBody(instance, document.Main, this, revitDocument);
//Create builder
HtmlBuilder builder = new HtmlBuilder();
//Save file
builder.BuildDocument(document, revitDocument, "D:\\tmp\\result", DetailLevel.Detail);
```



2.6 Property generator

In order to simplify generation of properties with attributes Extensible Storage Framework provides Visual Studio Addin “Properties generator”. It generates properties and attributes of all types (serialization, UI, documentation) in selected class (the class has to be selected in class view):



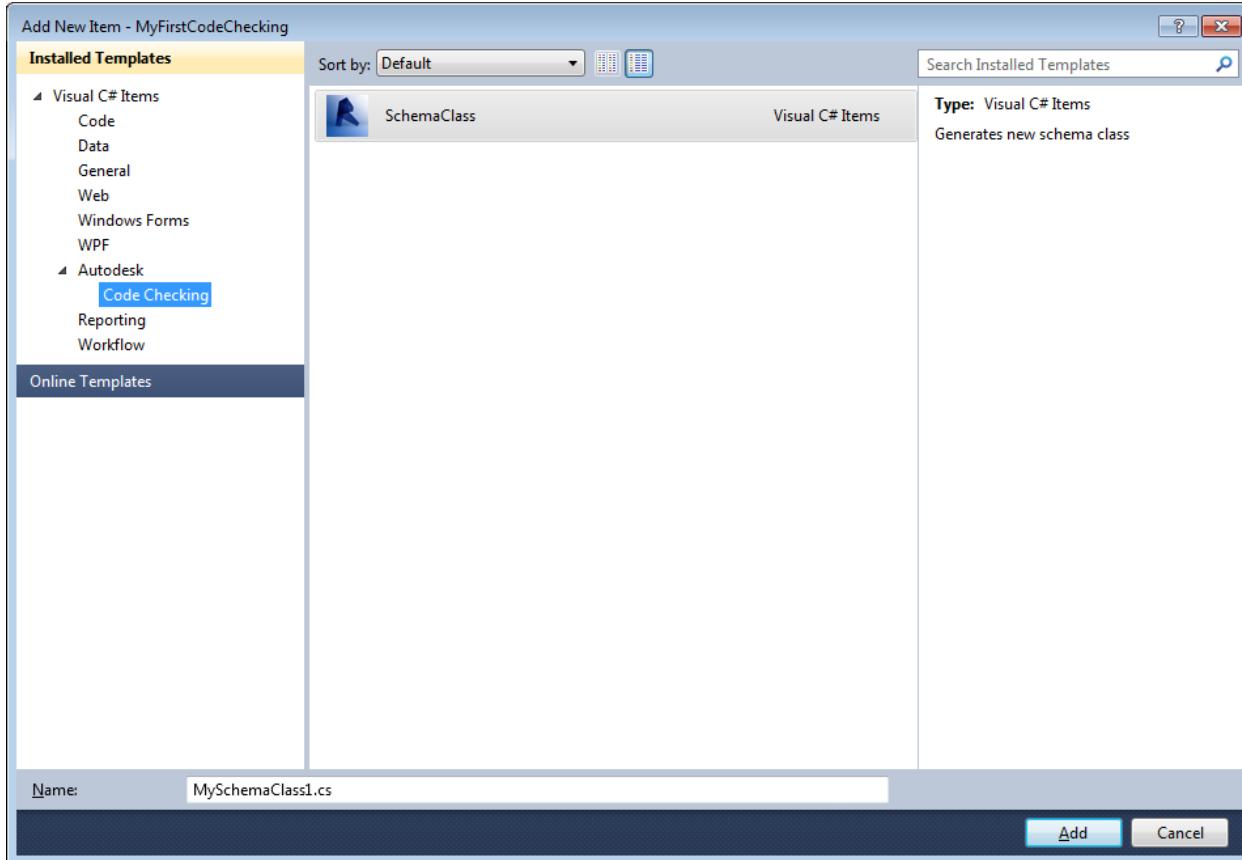


Code region

```
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.SchemaProperty(
    Unit=Autodesk.Revit.DB.UnitType.UT_Length,
    DisplayUnit=Autodesk.Revit.DB.DisplayUnitType.DUT_METERS)]
[Autodesk.Revit.UI.ExtensibleStorage.Framework.Attributes.UnitTextBox(
    ValidateMinimumValue=false,
    ValidateMaximumValue=false,
    AttributeUnit=Autodesk.Revit.DB.DisplayUnitType.DUT_METERS,
    Category="My category",
    IsVisible=true,
    IsEnabled=true,
    Index=-1,
    Localizable=true)]
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.Attributes.Value(
    LocalizableValue=false,
    Lev-
    el=Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.DetailLevel.General
    ,
    Localizable=false,
    Index=-1)]
public Double b{get;set;}
```

2.7 Schema class template

Extensible Storage Framework provides Visual Studio class template to generate class derived from SchemaClass with automatically generated guids:



Code region

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Autodesk.Revit.DB.ExtensibleStorage;
using Autodesk.Revit.DB;

namespace ExtensibleStorageFramework7
{
    [Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.Schema("MySchemaClass3",
    "e4367a0c-4019-4606-8692-baf51f819b2e")]
    public class MySchemaClass3 : Auto-
    desk.Revit.DB.ExtensibleStorage.Framework.SchemaClass
    {
        public MySchemaClass3()
        {

        }

        public MySchemaClass3(Document document)
        {
        }

        public MySchemaClass3(Entity entity, Document document)
            : base(entity, document)
        {
        }
    }
}
```

2.8 Complex example

The goal of this example is to create simple calculator supporting basic operations on two values of length type (+, -, /, *), the result of the operation should be printed to html.

2.8.1 Design

- UI:

The diagram shows a user interface with four input fields. The first field is labeled 'a' and contains a text input box. The second field is labeled 'operation' and contains a dropdown menu with a downward arrow. The third field is labeled 'b' and contains a text input box. The fourth field is labeled 'result' and contains a text input box.

- Document:

○ A	value
○ B	value
○ Operation	type of operation
○ Result	value

2.8.2 Class definition

The best thing to start with is the creation of the main class which in this case will be a `Calculator` class:

Code region

```
[Schema("CalculationData", "ec708e62-eebd-4ddf-9942-c3221913f6a3")]
public class Calculator : SchemaClass
{
    public Calculator()
    {
    }
}
```

In case of A and B parameters there is no particular doubts about their definitions:

Code region

```
[SchemaProperty(Unit=UnitType.UT_Length,DisplayUnit=DisplayUnitType.DUT_METERS)]
public Double A{get;set;}

[SchemaProperty(Unit = UnitType.UT_Length, DisplayUnit = DisplayUnitType.DUT_METERS)]
public Double B { get; set; }
```

For operation parameter there may be created dedicated enum:

Code region

```
enum OperationType
{
    add,
    subtract,
    divide,
    multiply
}

[SchemaProperty]
public OperationType Operation { get; set; }
```

In case of results it is not so simple. We have to remember that different operations give results in different units. It is possible to create one result without units and later play around formatting but in this example we will create three types of results which visibility will be controlled depends on current operation:

Code region

```
[SchemaProperty(
    Unit = UnitType.UT_Length,
    DisplayUnit = DisplayUnitType.DUT_METERS)]
public Double ResultLength { get; set; }

[SchemaProperty(
    Unit = UnitType.UT_Number,
    DisplayUnit = DisplayUnitType.DUT_GENERAL)]
public Double ResultNumber { get; set; }

[SchemaProperty(
    Unit = UnitType.UT_Area,
    DisplayUnit = DisplayUnitType.DUT_SQUARE_METERS)]
public Double ResultSurface { get; set; }
```

At this stage we can prepare simple `Calculate` method:

Code region

```
public void Calculate()
{
    switch (this.Operation)
    {
        case OperationType.add:
            this.ResultLength = this.A + this.B;
            break;
        case OperationType.subtract:
            this.ResultLength = this.A - this.B;
            break;
        case OperationType.divide:
            this.ResultNumber = this.A / this.B;
            break;
        case OperationType.multiply:
            this.ResultSurface = this.A * this.B;
            break;
    }
}
```

2.8.3 UI

2.8.3.1 Attributes

Now when we have the data structure ready we can switch to UI creation. First let's plan what controls will be used for fields visualization. Taking to consideration our design:

- A – editable TextBox supporting units
- B – editable TextBox supporting units
- Operation – combobox presenting enum
- ResultLength – not editable TextBox supporting units visible depends on current operation
- ResultNumber – not editable TextBox supporting units visible depends on current operation
- ResultSurface – not editable TextBox supporting units visible depends on current operation

Based on it we can add UI attributes to properties of `Calculator` class:

Code region

```
[SchemaProperty(Unit=UnitType.UT_Length,DisplayUnit=DisplayUnitType.DUT_METERS)]
[UnitTextBox]
public Double A{get;set;}

[SchemaProperty(Unit = UnitType.UT_Length, DisplayUnit = DisplayUnitType.DUT_METERS)]
[UnitTextBox]
public Double B { get; set; }

[SchemaProperty]
[EnumControl(typeof(OperationType),PresentationMode.Combobox,PresentationItem.Text)]
public OperationType Operation { get; set; }

[SchemaProperty(Unit = UnitType.UT_Length, DisplayUnit = DisplayUnitType.DUT_METERS)]
[UnitTextBox(IsEnabled = false)]
public Double ResultLength { get; set; }

[SchemaProperty(Unit = UnitType.UT_Number, DisplayUnit = DisplayUnitType.DUT_GENERAL)]
[UnitTextBox(IsEnabled = false)]
public Double ResultNumber { get; set; }

[SchemaProperty(Unit = UnitType.UT_Area, DisplayUnit = DisplayUnitType.DUT_SQUARE_METERS)]
[UnitTextBox(IsEnabled = false)]
public Double ResultSurface { get; set; }
```

2.8.3.2 First launch

Now we can run our application just to see the current state. Before however we have to create addin which will run a window with our layout (please refer to RevitAPI documentation to learn about creation of Revit Addin file):

Code region

```
public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
    ref string message, Autodesk.Revit.DB.ElementSet elements)
{
    Document revitDoc = commandData.Application.ActiveUIDocument.Document;
    Calculator calc = new Calculator();

    Layout layout = Layout.Build(typeof(Calculator), null);

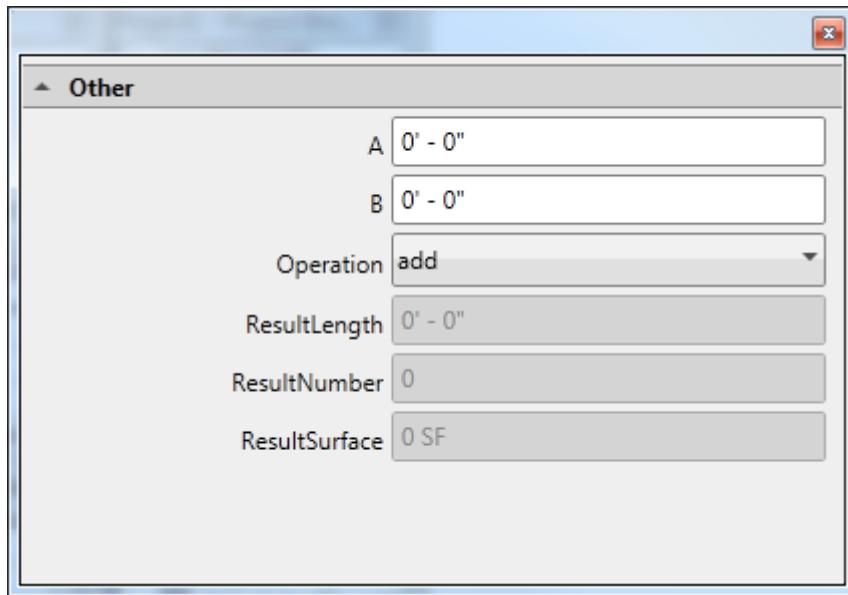
    ILayoutControl layoutControl = Layout.BuildControl(null,revitDoc,layout,
        calc.GetEntity());

    System.Windows.Window wnd = new System.Windows.Window();
    wnd.Content = layoutControl;
    Layout.ApplyStyle(wnd);
    wnd.ShowDialog();

    return Autodesk.Revit.UI.Result.Succeeded;
}
```

Note: Revit Extensible Storage Entity mode was selected (`calc.GetEntity()`)

As a result we get:



2.8.3.3 ServerUI

Now it is time to make calculation when values change. In order to do it we have to create an object which implements `ISeverUI` interface which will allow us to interact with the dialog:

Code region

```
class ServerUI : IServerUI
{
    public IList GetDataSource(string key, Document document, DisplayUnitType unitType)
    {
        return null;
    }

    public void LayoutInitialized(object sender, LayoutInitializedEventArgs e)
    {
    }

    public void ValueChanged(object sender, ValueChangedEventArgs e)
    {
    }

    public string GetResource(string key, string context)
    {
        return key;
    }

    public Uri GetResourceImage(string key, string context)
    {
        return null;
    }
}
```

Reaction on value changed should be taken in `ValueChanged` method:

Code region

```

public void ValueChanged(object sender, ValueChangedEventArgs e)
{
    Calculator calculator = new Calculator();
    calculator.SetProperties(e.Entity as Entity);

    if (e.FieldName == "Operation")
    {
        bool lengthVisible = calculator.Operation == OperationType.subtract ||
            calculator.Operation == OperationType.add;
        bool surfaceVisible = calculator.Operation == OperationType.multiply;
        bool numberVisible = calculator.Operation == OperationType.divide;

        e.Editor.SetAttribute("ResultLength",
            FieldUIAttribute.PropertyIsVisible,
            lengthVisible, DisplayUnitType.DUT_UNDEFINED);
        e.Editor.SetAttribute("ResultNumber",
            FieldUIAttribute.PropertyIsVisible,
            numberVisible, DisplayUnitType.DUT_UNDEFINED);
        e.Editor.SetAttribute("ResultSurface",
            FieldUIAttribute.PropertyIsVisible,
            surfaceVisible, DisplayUnitType.DUT_UNDEFINED);
    }
}

```

Note: In current example the whole `Calculator` object is created based on `Entity`. Values can also be taken one by one from `Editor`.

Now we have to create the instance of our server and pass it to layout:

Code region

```

ServerUI serverUI = new ServerUI();

Layout layout = Layout.Build(typeof(Calculator), null);

ILayoutControl layoutControl = Layout.BuildControl(serverUI, revitDoc, layout,
calc.GetEntity());

```

After launching it appears that visibility is not set at the start. Only after changing operation parameter visibility is set properly. In order to make it work on the start lets extract part of the code to external method and call it in `LayoutInitialized`:

Code region

```
public void LayoutInitialized(object sender, LayoutInitializedEventArgs e)
{
    Calculator calculator = new Calculator();
    calculator.SetProperties(e.Entity as Entity);
    SetResultVisibility(e, calculator);
}

public void ValueChanged(object sender, ValueChangedEventArgs e)
{
    Calculator calculator = new Calculator();
    calculator.SetProperties(e.Entity as Entity);

    if (e.FieldName == "Operation")
    {
        SetResultVisibility(e, calculator);
    }
}

private static void SetResultVisibility(SchemaEditorEventArgs e, Calculator calculator)
{
    bool lengthVisible = calculator.Operation == OperationType.subtract ||
        calculator.Operation == OperationType.add;
    bool surfaceVisible = calculator.Operation == OperationType.multiply;
    bool numberVisible = calculator.Operation == OperationType.divide;

    e.Editor.SetAttribute("ResultLength",
        FieldUIAttribute.PropertyIsVisible,
        lengthVisible, DisplayUnitType.DUT_UNDEFINED);
    e.Editor.SetAttribute("ResultNumber",
        FieldUIAttribute.PropertyIsVisible,
        numberVisible, DisplayUnitType.DUT_UNDEFINED);
    e.Editor.SetAttribute("ResultSurface",
        FieldUIAttribute.PropertyIsVisible,
        surfaceVisible, DisplayUnitType.DUT_UNDEFINED);
}
```

Note: If we are sure what is the start visibility we can set it directly in attributes.

The only thing left is running calculation after parameters value change:

Code region

```

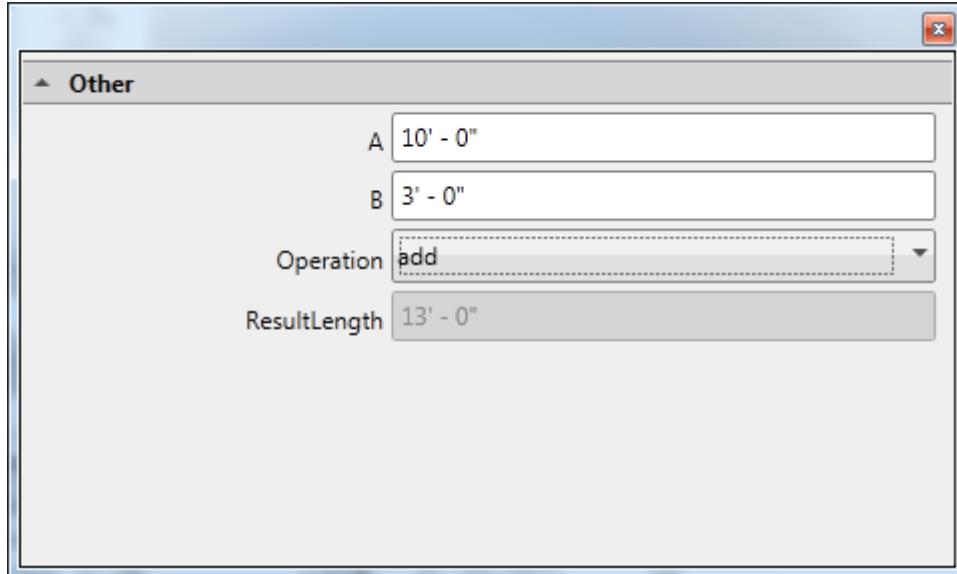
public void ValueChanged(object sender, ValueChangedEventArgs e)
{
    Calculator calculator = new Calculator();
    calculator.SetProperties(e.Entity as Entity);

    if (e.FieldName == "Operation")
    {
        SetResultVisibility(e, calculator);
    }

    if (e.FieldName == "A" || e.FieldName == "B" || e.FieldName == "Operation")
    {
        calculator.Calculate();
        e.Editor.SetValue("ResultLength",
            calculator.ResultLength, DisplayUnitType.DUT_METERS);
        e.Editor.SetValue("ResultNumber",
            calculator.ResultNumber, DisplayUnitType.DUT_GENERAL);
        e.Editor.SetValue("ResultSurface",
            calculator.ResultSurface, DisplayUnitType.DUT_SQUARE_METERS);
    }
}

```

After this change calculator starts to work:



2.8.3.4 Tuning

There are still few things to correct.

First let's order controls like in design draft. It can be done via `Index` property in attributes:

Code region

```
[UnitTextBox(Index=1)]
public Double A{get;set;}

[UnitTextBox(Index = 3)]
public Double B { get; set; }

[EnumControl(typeof(OperationType),PresentationMode.Combobox,PresentationItem.Text,
Index = 2)]
public OperationType Operation { get; set; }

[UnitTextBox(IsEnabled = false,Index=4)]
public Double ResultLength { get; set; }

[UnitTextBox(IsEnabled = false, Index = 5)]
public Double ResultNumber { get; set; }

[UnitTextBox(IsEnabled = false, Index = 6)]
public Double ResultSurface { get; set; }
```

When it is done we can change Descriptions of Results parameters to have the same name on the dialog:

Code region

```
[UnitTextBox(IsEnabled = false,Index=4,Description = "Result")]
public Double ResultLength { get; set; }

[UnitTextBox(IsEnabled = false, Index = 5, Description = "Result")]
public Double ResultNumber { get; set; }

[UnitTextBox(IsEnabled = false, Index = 6, Description = "Result")]
public Double ResultSurface { get; set; }
```

Note: In order to translate these names just mark the attribute as Localizable (Localizable = true) and use GetResource method from ServerUI.

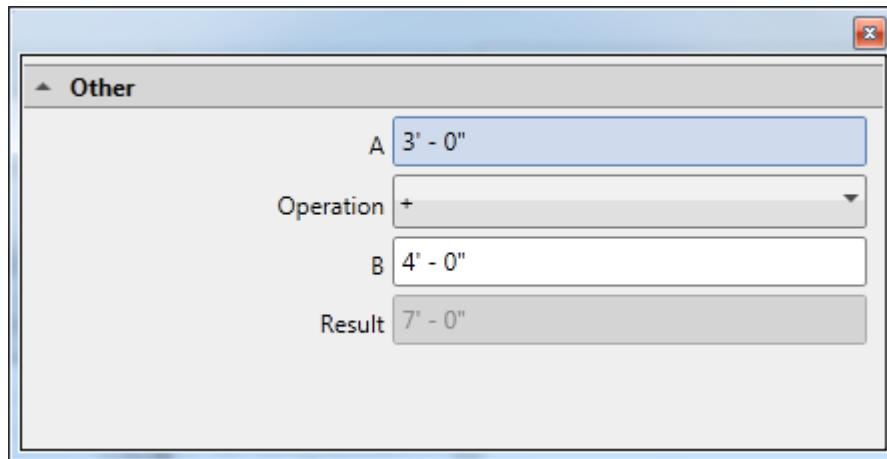
Next thing will concern Operation control. We will translate names to symbols. It may be done via GetResource method in our ServerUI:

Code region

```
public string GetResource(string key, string context)
{
    if(key == "add")
    {
        return "+";
    }
    else if(key == "subtract")
    {
        return "-";
    }
    else if(key == "divide")
    {
        return "/";
    }
    else if (key == "multiply")
    {
        return "*";
    }

    return key;
}
```

As a result of these changes we get:



As a last thing in UI part it would be good to provide validation for field A and B. Let's assume that their range will be defined as (-100ft,100ft). It can be simply done via attributes:

Code region

```
[SchemaProperty(Unit=UnitType.UT_Length,DisplayUnit=DisplayUnitType.DUT_METERS)]
[UnitTextBox(Index=1,AttributeUnit= DisplayUnitType.DUT_DECIMAL_FEET,
    ValidateMinimumValue=true,MinimumValue = -100,
    ValidateMaximumValue=true,MaximumValue = 100)]
public Double A{get;set;}

[SchemaProperty(Unit = UnitType.UT_Length, DisplayUnit = DisplayUnitType.DUT_METERS)]
[UnitTextBox(Index = 3,AttributeUnit= DisplayUnitType.DUT_DECIMAL_FEET,
    ValidateMinimumValue=true,MinimumValue = -100,
    ValidateMaximumValue=true,MaximumValue = 100)]
public Double B { get; set; }
```

In case of B parameter it is not enough. There has to be protection done against 0 but only when operation will be division. The best way to do it is to create dedicated class which implements `IFieldValidator` interface and set it to `FieldValidator` property in the `UnitTextBoxAttribute`.

Code region

```
[SchemaProperty(Unit = UnitType.UT_Length, DisplayUnit = DisplayUnitType.DUT_METERS)]
[UnitTextBox(Index = 3,AttributeUnit= DisplayUnitType.DUT_DECIMAL_FEET,
    ValidateMinimumValue=true,MinimumValue = -100,
    ValidateMaximumValue=true,MaximumValue = 100,
    FieldValidator = typeof(ParameterBValidator))]
public Double B { get; set; }

public class ParameterBValidator : IFieldValidator
{
    public bool ValidateValue(object entity, string field, object value,
        DisplayUnitType unit)
    {
        Calculator calculator = new Calculator();
        calculator.SetProperties(entity as Entity);

        double val = Convert.ToDouble(value);

        if (calculator.Operation == OperationType.divide)
        {
            return val != 0;
        }

        return true;
    }
}
```

Additionally let's change B value when it is set to 0 and operation changes to division. Following code should be added to `ValueChanged` method in `ServerUI` class:

Code region

```

if (e.FieldName == "Operation")
{
    SetResultVisibility(e, calculator);

    if (calculator.B == 0 && calculator.Operation == OperationType.divide)
    {
        calculator.B = 1;
        e.Editor.SetValue("B", 1, DisplayUnitType.DUT_METERS);
    }
}

```

2.8.4 Documentation

In the first approach let's create documentation automatically using only attributes. Each field can be defined as ValueWithName:

Code region

```

[ValueWithName]
public Double A{get;set;}

[ValueWithName]
public Double B { get; set; }

[ValueWithName]
public OperationType Operation { get; set; }

[ValueWithName]
public Double ResultLength { get; set; }

[ValueWithName]
public Double ResultNumber { get; set; }

[ValueWithName]
public Double ResultSurface { get; set; }

```

Before generating a document current value from layout should be taken:

Code region

```

Calculator result = new Calculator();
result.SetProperties(layoutControl.GetEntity());

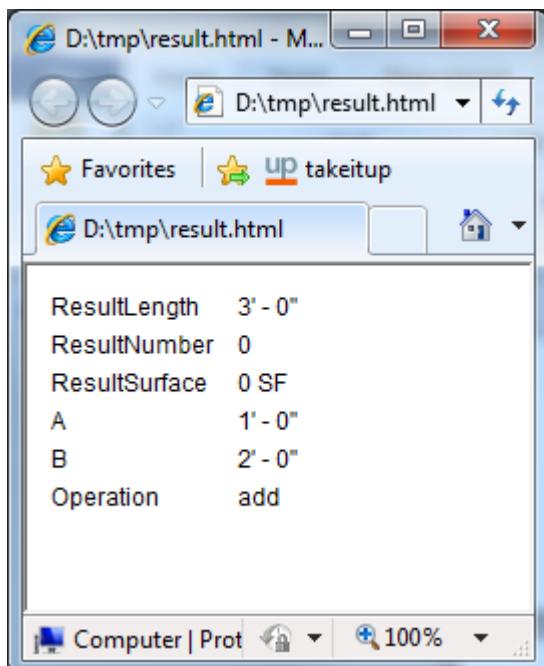
```

Now we can generate document:

Code region

```
Document document = new Document();
DocumentBody.FillBody(result, document.Main, serverUI, revitDoc);
HtmlBuilder builder = new HtmlBuilder();
builder.BuildDocument(document,revitDoc, "D:\\tmp\\result", DetailLevel.Detail);
```

As a result we get:



As we see it is not perfect. First let's order them using `Index`:

Code region

```
[ValueWithName(Index=1)]
public Double A{get;set;}

[ValueWithName(Index=2)]
public Double B { get; set; }

[ValueWithName(Index=3)]
public OperationType Operation { get; set; }

[ValueWithName(Index = 4)]
public Double ResultLength { get; set; }

[ValueWithName(Index = 4)]
public Double ResultNumber { get; set; }

[ValueWithName(Index = 4)]
public Double ResultSurface { get; set; }
```

Now let's translate name of Operation to symbol. The simplest way is to define it as LocalizableValue:

Code region

```
[ValueWithName(Index=3,LocalizableValue = true)]
public OperationType Operation { get; set; }
```

The last thing which is problematic concerns Results. There are three printed while we need only one. There are few solutions for this. If we rely on attributes and automatic documentation generation we will have to perform some post generation action:

Code region

```
foreach (DocumentElement docElement in document.Main.Elements)
{
    DocumentValueWithName val = docElement as DocumentValueWithName;
    if (val != null)
    {
        switch(val.Source)
        {
            case "ResultLength":
                val.Visible = result.Operation == OperationType.subtract ||
                    result.Operation == OperationType.add;
                break;
            case "ResultNumber":
                val.Visible = result.Operation == OperationType.divide;
                break;
            case "ResultSurface":
                val.Visible = result.Operation == OperationType.multiply;
                break;
        }
    }
}
```

Additionally we can set names to "Result" without postfix (like in UI):

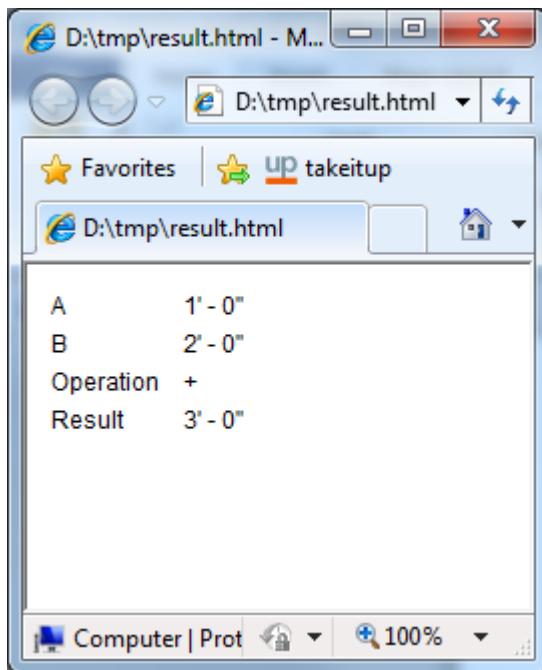
Code region

```
[ValueWithName(Index = 4,Name="Result")]
public Double ResultLength { get; set; }

[ValueWithName(Index = 4, Name = "Result")]
public Double ResultNumber { get; set; }

[ValueWithName(Index = 4, Name = "Result")]
public Double ResultSurface { get; set; }
```

The final note looks as follows:



The same effect may be achieved if we will build note manually but with attributes help:

Code region

```
document.Main.Elements.Add(  
    DocumentElement.GetDocumentElement("A",result,serverUI,revitDoc));  
document.Main.Elements.Add(  
    DocumentElement.GetDocumentElement("B",result,serverUI,revitDoc));  
document.Main.Elements.Add(  
    DocumentElement.GetDocumentElement("Operation",result,serverUI,revitDoc));  
switch (result.Operation)  
{  
    case OperationType.add:  
    case OperationType.subtract:  
        document.Main.Elements.Add(  
            DocumentElement.GetDocumentElement(  
                "ResultLength", result, serverUI, revitDoc));  
        break;  
    case OperationType.divide:  
        document.Main.Elements.Add(  
            DocumentElement.GetDocumentElement(  
                "ResultNumber", result, serverUI, revitDoc));  
        break;  
    case OperationType.multiply:  
        document.Main.Elements.Add(  
            DocumentElement.GetDocumentElement(  
                "ResultSurface", result, serverUI, revitDoc));  
        break;  
}
```

Note: We can also build document without any attributes creating each DocumentElement manually.
For example:

Code region

```
document.Main.Elements.Add(new DocumentValueWithName("Result", "3 ft."));
```

2.8.5 Serialization to ProjectInfo

At the very end we can save our data to Revit element (it will be ProjectInfo) and load it on the start:

Code region

```
public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
    ref string message, Autodesk.Revit.DB.ElementSet elements)
{
    Document revitDoc = commandData.Application.ActiveUIDocument.Document;
    Calculator calc = new Calculator();

    calc.Load(revitDoc.ProjectInformation);

    ...

    Calculator result = new Calculator();
    result.SetProperties(layoutControl.GetEntity());

    result.Save(revitDoc.ProjectInformation);
}
```

2.8.6 Complete Code

Code region – Command class

```
public class Command : IExternalCommand
{
    public Result Execute(ExternalCommandData commandData,
        ref string message, ElementSet elements)
    {
        //initialize data structure
        Document revitDoc = commandData.Application.ActiveUIDocument.Document;
        Calculator calc = new Calculator();
        calc.Load(revitDoc.ProjectInformation);
        ServerUI serverUI = new ServerUI();
        //UI
        Layout layout = Layout.Build(typeof(Calculator), null);
        ILayoutControl layoutControl = Layout.BuildControl(serverUI, revitDoc,
            layout, calc.GetEntity());
        System.Windows.Window wnd = new System.Windows.Window();
        wnd.Content = layoutControl;
        Autodesk.Revit.UI.ExtensibleStorage.Framework.Layout.ApplyStyle(wnd);
        wnd.ShowDialog();
        //documentation
        Calculator result = new Calculator();
        result.SetProperties(layoutControl.GetEntity());
        Document document = new Document();
        DocumentBody.FillBody(result, document.Main, serverUI, revitDoc);
        foreach (DocumentElement docElement in document.Main.Elements)
        {
            DocumentValueWithName val = docElement as DocumentValueWithName;
            if (val != null) {
                switch (val.Source)
                {
                    case "ResultLength":
                        val.Visible = result.Operation == OperationType.subtract ||
                            result.Operation == OperationType.add;
                        break;
                    case "ResultNumber":
                        val.Visible = result.Operation == OperationType.divide;
                        break;
                    case "ResultSurface":
                        val.Visible = result.Operation == OperationType.multiply;
                        break;
                }
            }
        }
        HtmlBuilder builder = new HtmlBuilder();
        builder.BuildDocument(document, revitDoc, "D:\\tmp\\result",
            DetailLevel.Detail);
        //save to project
        result.Save(revitDoc.ProjectInformation);
        return Autodesk.Revit.UI.Result.Succeeded;
    }
}
```

Code region – ServerUI class

```

class ServerUI : IServerUI
{
    public IList GetDataSource(string key, Document document, DisplayUnitType unitType)
    {return null; }

    public void LayoutInitialized(object sender, LayoutInitializedEventArgs e) {
        Calculator calculator = new Calculator();
        calculator.SetProperties(e.Entity as Entity);
        SetResultVisibility(e, calculator);
    }

    public void ValueChanged(object sender, ValueChangedEventArgs e) {
        Calculator calculator = new Calculator();
        calculator.SetProperties(e.Entity as Entity);
        if (e.FieldName == "Operation"){
            SetResultVisibility(e, calculator);
            if (calculator.B == 0 && calculator.Operation == OperationType.divide){
                calculator.B = 1;
                e.Editor.SetValue("B", 1, DisplayUnitType.DUT_METERS);
            }
        }
        if (e.FieldName == "A" || e.FieldName == "B" || e.FieldName == "Operation"){
            calculator.Calculate();
            e.Editor.SetValue("ResultLength", calculator.ResultLength,
                DisplayUnitType.DUT_METERS);
            e.Editor.SetValue("ResultNumber", calculator.ResultNumber,
                DisplayUnitType.DUT_GENERAL);
            e.Editor.SetValue("ResultSurface", calculator.ResultSurface,
                DisplayUnitType.DUT_SQUARE_METERS);
        }
    }

    private static void SetResultVisibility(SchemaEditorEventArgs e,
        Calculator calculator) {
        bool lengthVisible = calculator.Operation == OperationType.subtract ||
            calculator.Operation == OperationType.add;
        bool surfaceVisible = calculator.Operation == OperationType.multiply;
        bool numberVisible = calculator.Operation == OperationType.divide;
        e.Editor.SetAttribute("ResultLength", FieldUIAttribute.PropertyIsVisible,
            lengthVisible, DisplayUnitType.DUT_UNDEFINED);
        e.Editor.SetAttribute("ResultNumber", FieldUIAttribute.PropertyIsVisible,
            numberVisible, DisplayUnitType.DUT_UNDEFINED);
        e.Editor.SetAttribute("ResultSurface",
            FieldUIAttribute.PropertyIsVisible, surfaceVisible,
            DisplayUnitType.DUT_UNDEFINED);
    }

    public string GetResource(string key, string context){
        if (key == "add")return "+";
        else if (key == "subtract")return "-";
        else if (key == "divide")return "/";
        else if (key == "multiply")return "*";
        return key;
    }

    public Uri GetResourceImage(string key, string context) {return null; }
}

```

Code region – Calculator class

```
[Schema("CalculationData", "ec708e62-eebd-4ddf-9942-c3221913f6a3")]
public class Calculator : SchemaClass
{
    [SchemaProperty(Unit=UnitType.UT_Length,DisplayUnit=DisplayUnitType.DUT_METERS)]
    [UnitTextBox(Index=1,AttributeUnit= DisplayUnitType.DUT_DECIMAL_FEET,
        ValidateMinimumValue=true,MinimumValue = -100,
        ValidateMaximumValue=true,MaximumValue = 100)]
    [ValueWithName(Index=1)]
    public Double A{get;set;}

    [SchemaProperty(Unit = UnitType.UT_Length, DisplayUnit = DisplayUnitType.DUT_METERS)]
    [UnitTextBox(Index = 3,AttributeUnit= DisplayUnitType.DUT_DECIMAL_FEET,
        ValidateMinimumValue=true,MinimumValue = -100,
        ValidateMaximumValue=true,MaximumValue = 100,
        FieldValidator = typeof(ParameterBValidator))]
    [ValueWithName(Index=2)]
    public Double B { get; set; }

    [SchemaProperty]
    [EnumControl(typeof(OperationType),
        PresentationMode.ComboBox,PresentationItem.Text,Index = 2)]
    [ValueWithName(Index=3,LocalizableValue = true)]
    public OperationType Operation { get; set; }

    [SchemaProperty(Unit = UnitType.UT_Length, DisplayUnit = DisplayUnitType.DUT_METERS)]
    [UnitTextBox(IsEnabled = false,Index=4,Description = "Result")]
    [ValueWithName(Index = 4,Name="Result")]
    public Double ResultLength { get; set; }

    [SchemaProperty(Unit = UnitType.UT_Number,
        DisplayUnit = DisplayUnitType.DUT_GENERAL)]
    [UnitTextBox(IsEnabled = false, Index = 5, Description = "Result")]
    [ValueWithName(Index = 4, Name = "Result")]
    public Double ResultNumber { get; set; }

    [SchemaProperty(Unit = UnitType.UT_Area,
        DisplayUnit = DisplayUnitType.DUT_SQUARE_METERS)]
    [UnitTextBox(IsEnabled = false, Index = 6, Description = "Result")]
    [ValueWithName(Index = 4, Name = "Result")]
    public Double ResultSurface { get; set; }

    public void Calculate()
    {
        switch (this.Operation)
        {
            case OperationType.add: this.ResultLength = this.A + this.B; break;
            case OperationType.subtract: this.ResultLength = this.A - this.B; break;
            case OperationType.divide: this.ResultNumber = this.A / this.B; break;
            case OperationType.multiply: this.ResultSurface = this.A * this.B; break;
        }
    }
}
```

Code region – OperationType

```
public enum OperationType
{
    add,
    subtract,
    divide,
    multiply
}
```

Code region – Calculator class

```
public class ParameterBValidator : IFieldValidator
{
    public bool ValidateValue(object entity, string field, object value,
        DisplayUnitType unit)
    {
        Calculator calculator = new Calculator();
        calculator.SetProperties(entity as Entity);

        double val = Convert.ToDouble(value);

        if (calculator.Operation == OperationType.divide)
        {
            return val != 0;
        }

        return true;
    }
}
```

3 CODE CHECKING FRAMEWORK

The Code Checking Framework and SDK provide a common environment for developing Code Checking Applications. This includes:

- Common components (e.g. CodeChecking, CodeChecking.Storage etc.)
- Revit External Applications (UI and DB) part of the Structural Analysis and Code Checking Toolkit for Autodesk Revit package available on the Autodesk Exchange Apps.
- Engineering components – dedicated engineering components for Code Checking applications.
- SDK – documentations, templates for Visual Studio and examples.
- Common presentation framework – defined in UI components

3.1 Workflow

In order to understand Code Checking Framework it is important to learn the whole calculation process which may be simplified to following steps:

1. Selection of active code
2. Definition of common calculation parameters
3. Definition of element settings
4. Assigning element settings to elements
5. Elements selection
6. Run calculation
7. Overview results for all elements
8. View results for some specific selected element

3.2 Have to know

- **.NET**

Code Checking Framework is implemented in .NET technology and basic knowledge about it is required from the user. C# is preferred one because all templates, examples, code generation visual addin etc. are provided for this particular language.

- **RevitAPI**

Code Checking Framework is built on top of Revit API and the developer should be familiar with some aspects of it:

- **External Application**

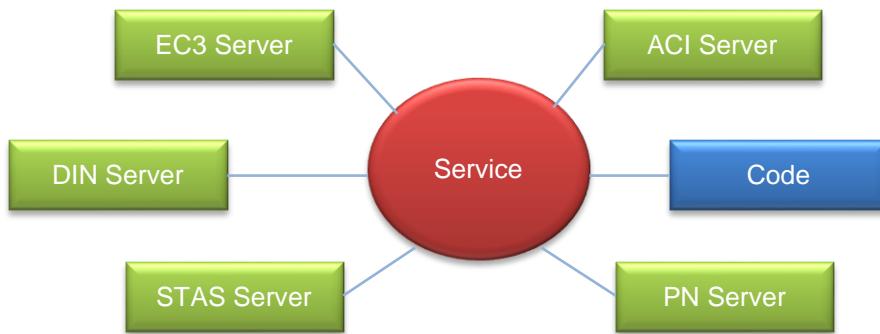
External Application is one of the mechanisms which allow the user to attach to Revit via Revit API. In this mode provided application is launched at the start of Revit. There are two types of applications: DB and UI.

- **Extensible Storage**

Extensible Storage is serialization mechanism provided with RevitAPI. Code Checking Framework uses it to store data as well as exchange information between different components (including users).

- **External Services**

External Services provides mechanism where the object called service describes some functionality and servers which are attached to the service provides some specific implementation. Functionality described by Code Checking Framework Service is Code Checking calculation and servers provide implementations of calculation for different codes:



- **Extensible Storage Framework**

The Extensible Storage Framework provides tools for serialization, UI and documentation. Code Checking Framework is entirely based on it.

- **Results Builder**

ResultsBuilder is a component designed to store results in Revit. There are two ways of using ResultsBuilder in CodeChecking: as input data provider or as output data storage.

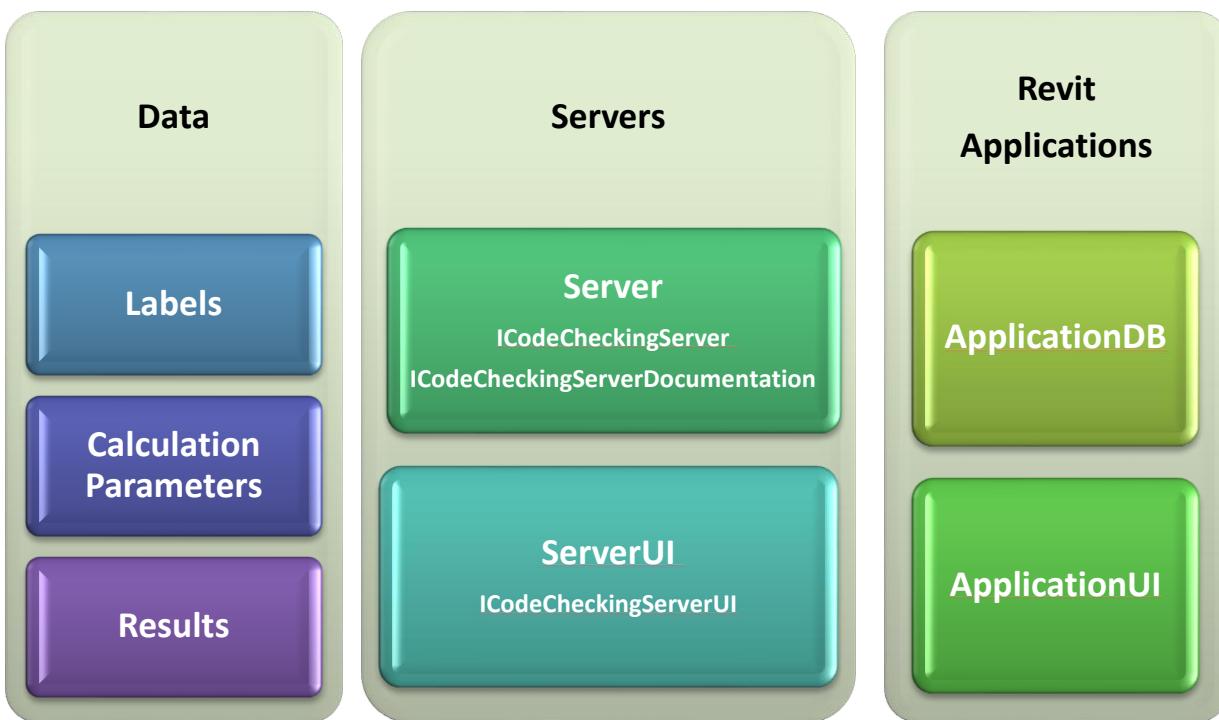
3.3 Components

Code Checking Framework consists of components. The most important are:

- **CodeChecking.dll**
Namespace: Autodesk.Revit.DB.CodeChecking
Provides base classes and foundations for Code Checking Framework such as definition of Service and interface for servers.
- **CodeChecking.Documentation.dll**
Namespace: Autodesk.Revit.DB.CodeChecking.Documentation
Provides tools needed to create document in Code Checking.
- **CodeChecking.Storage.dll**
Namespace: Autodesk.Revit.DB.CodeChecking.Storage
Provides access to data stored in Code Checking Framework
- **CodeChecking.UI.dll**
Namespace: Autodesk.Revit.UI.CodeChecking
Provides tools needed to create and control UI in Code Checking dialogs.

3.4 Code Checking Application

A Code Checking Application is a whole solution provided by the Developer to Revit users.



It consists of three main parts:

- Data – structures definitions
- Servers- Code Checking server
- Revit Applications

3.4.1 Data

There are three main types of data that built whole Code Checking Application:

- CalculationParameters – (Code Settings) describes general calculation parameters. They are common for all elements. Each code has own set of specific parameters.
- Labels – (Element Settings) – parameters common for specific groups of elements (e.g. steel beams). They are assigned directly to element. Each code has own set of specific parameters.
- Results – Results of calculation.

All data structures are based on Revit Extensible Storage and one of main tasks of the Code Checking developer is to define data structures for all presented items.

3.4.2 Servers

Code Checking Framework is based on External Services mechanism. The users register a server which is used for calculation purpose. Additionally developers need to register a UI server which allows the user to interact with UI.

3.4.3 Revit Applications

Revit applications allow the developers to be attached to Revit when it starts. There are two types of applications:

- DB – should take all actions connected with DB layer (e.g. server initialization and registration)

- UI – should take all actions connected with UI layer (e.g. server UI initialization and registration)

3.5 ICodeCheckingServer

Code Checking Service communicates with external servers via `ICodeCheckingServer` interface:

Methods	Description
<code>void Verify(ServiceData data)</code>	Runs Code Checking calculation.
<code>Schema GetLabelSchema(StructuralAssetClass material, BuiltInCategory category);</code>	Returns label schema for element of specific material and category.
<code>Schema GetCalculationParamSchema();</code>	Returns schema for calculation parameters
<code>Schema GetResultsSchema(StructuralAssetClass material, BuiltInCategory category);</code>	Returns result schema for element of specific material and category.
<code>Entity ConvertLabelToCurrentVersion(Entity oldVersion, Document document);</code>	Converts old label schema instance to current one.
<code>Entity ConvertCalcParamsToCurrentVersion(Entity oldVersion, Document document);</code>	Converts old calculation parameters schema instance to current one.
<code>Entity ConvertResultToCurrentVersion(Entity oldVersion, Document document);</code>	Converts old result schema instance to current one.
<code>Entity GetDefaultLabel(Document document, StructuralAssetClass material, BuiltInCategory category);</code>	Returns default instance of label.
<code>Entity GetDefaultCalcParams(Document document);</code>	Returns default instance of calculation parameter.
<code>IList<StructuralAssetClass> GetSupportedMaterials();</code>	Returns the list of supported materials.
<code>IList<BuiltInCategory> GetSupportedCategories(StructuralAssetClass material);</code>	Returns the list of supported categories for specific material.
<code>bool LoadCasesAndCombinationsSupport();</code>	Informs Code Checking Framework if a standard way of supporting LoadCases and Combinations is chosen (it will result e.g. in UI).
<code>bool ResultBuilderPackagesAsInputData();</code>	Informs Code Checking Framework if ResultBuilder packages are treated as input data.
<code>int GetVersion();</code>	Returns version of server. It is important for worksharing. It should be changed whenever important changes were made in data structures or calculation algorithm.
<code>ResultsPackageTypes GetOutputPackageResultTypes()</code>	Returns ResultTypes for output ResultBuilder package.
<code>public UnitsSystem GetOutputPackageUnitSystem</code>	Returns unit system for output ResultBuilder package.

Other methods are derived from `IExternalServer` interface which is required by External Services mechanism.

3.5.1 Categories

Code Checking Framework supports following categories of elements:

- `BuiltInCategory.OST_ColumnAnalytical,`

- BuiltInCategory.OST_BeamAnalytical,
- BuiltInCategory.OST_WallAnalytical,
- BuiltInCategory.OST_FloorAnalytical,
- BuiltInCategory.OST_WallFoundationAnalytical,
- BuiltInCategory.OST_IsolatedFoundationAnalytical,
- BuiltInCategory.OST_FoundationSlabAnalytical,
- BuiltInCategory.OST_BraceAnalytical

3.5.2 Definition of data structures

Code Checking Framework is based on schemas and main task for developers is to create servers and provide schemas for their data structures. It is possible to do it directly using Extensible Storage:

Code region

```
public Schema GetLabelSchema(StructuralAssetClass material, BuiltInCategory category)
{
    Schema schema = Schema.Lookup(_labelGUID);
    if (schema == null)
    {
        SchemaBuilder builder = new SchemaBuilder(_labelGUID);
        builder.SetSchemaName("LabelCode1");

        builder.AddSimpleField("a", typeof(double)).SetUnitType(UnitType.UT_Length);
        builder.AddSimpleField("b", typeof(double)).SetUnitType(UnitType.UT_Length);

        schema = builder.Finish();
    }

    return schema;
}
```

However recommended approach is using helper mechanism provided with Extensible Storage Framework which is based on reflection (class properties + appropriate attributes):

Code region

```
[Schema("CB03232D-B362-45D9-B695-3E372C9D8413", "Code3")]
public class MyLabel : SchemaClass
{
    [SchemaProperty(UnitType.UT_Length, DisplayUnitType.DUT_METERS)]
    public double a { get; set; }

    [SchemaProperty(UnitType.UT_Length, DisplayUnitType.DUT_METERS)]
    public double b { get; set; }
}

public Schema GetLabelSchema(StructuralAssetClass material, BuiltInCategory category)
{
    SchemaClass.BuildSchema(typeof(MyLabel));
}
```

Additionally UI and Document attributes may be added which will automate the whole process (see Extensible Storage Framework for details).

3.5.3 Server

If the developer is using the Extensible Storage Framework he or she can derived his main server from `Server` class. This class automates a lot of actions (e.g. schema creation, instance conversion due to version etc.). As a result user class shrinks significantly. This class is dedicated for servers which provide one type of `Label` and one type of `Result` structures. Information about version may be set via `ServerVersionAttribute` (or by overriding `GetVersion` method).

Code region

```
[Autodesk.Revit.DB.CodeChecking.Attributes.ServerVersion(1)]
class Code : Autodesk.Revit.DB.CodeChecking.Server<CalculationParameter, Label, Result>
{
    public static readonly Guid ID = new Guid("D036C7A1-DBC9-4EEC-820D-7789403D0A98");

    public override void Calculate(ServiceData data)
    {
    }
    public override string GetDescription()
    {
        return "This code calculates...";
    }
    public override string GetName()
    {
        return "Code";
    }
    public override Guid GetServerId()
    {
        return ID;
    }
    public override string GetVendorId()
    {
        return "Vendor";
    }
    public override IList<BuiltInCategory> GetSupportedCategories()
    {
        return new List<BuiltInCategory>() { BuiltInCategory.OST_StructuralFraming };
    }
    public override IList<StructuralAssetClass> GetSupportedMaterials()
    {
        return new List<StructuralAssetClass>() { StructuralAssetClass.Metal };
    }
}
```

3.5.4 MultiStructureServer

The `Server` class provides possibility of defining server for the code with one `Label` structure and one `Result` structure. If server supports more than one material or category `MultiStructureServer` class may be used. The user may derive server from `MultiStructureServer` and set appropriate attributes to it:

- `CalculationParamsStructureAttribute` – only one can be assigned
- `LabelStructureAttribute`
- `ResultStructureAttribute`

Code region

```
[ServerVersion(1)]
[CalculationParamsStructure(typeof(CalculationParameter))]
[LabelStructure(typeof(Label), BuiltInCategory.OST_BeamAnalytical)]
[LabelStructure(typeof(Label2), BuiltInCategory.OST_ColumnsAnalytical)]
[ResultStructure(typeof(Result))]
public class Server : Autodesk.Revit.DB.CodeChecking.MultiStructureServer
{
}
```

In this specific example

- for `BuiltInCategory.OST_BeamAnalytical` there is `Label` class assigned (for any material)
- for `BuiltInCategory.OST_ColumnAnalytical` there is `Label2` class assigned (for any material)

3.5.5 Notification Service

`Notification Service` provides ability to notify system about warnings, errors, and progression of the process (e.g. calculation). There are following notification types available:

- Information
 - Regular
 - About progress of taken action: Start, Step, End
- Warning
- Error

Code region

```
NotificationService.AddError("No elements selected", ID.ToString());
NotificationService.ProgressStart("Starting calculation", 3, ID.ToString());
NotificationService.ProgressStep("Next element is calculated", ID.ToString());
```

Progress break:

The special scenario of using `Notification Service` is breaking the current process (e.g. after clicking Cancel button on progress).

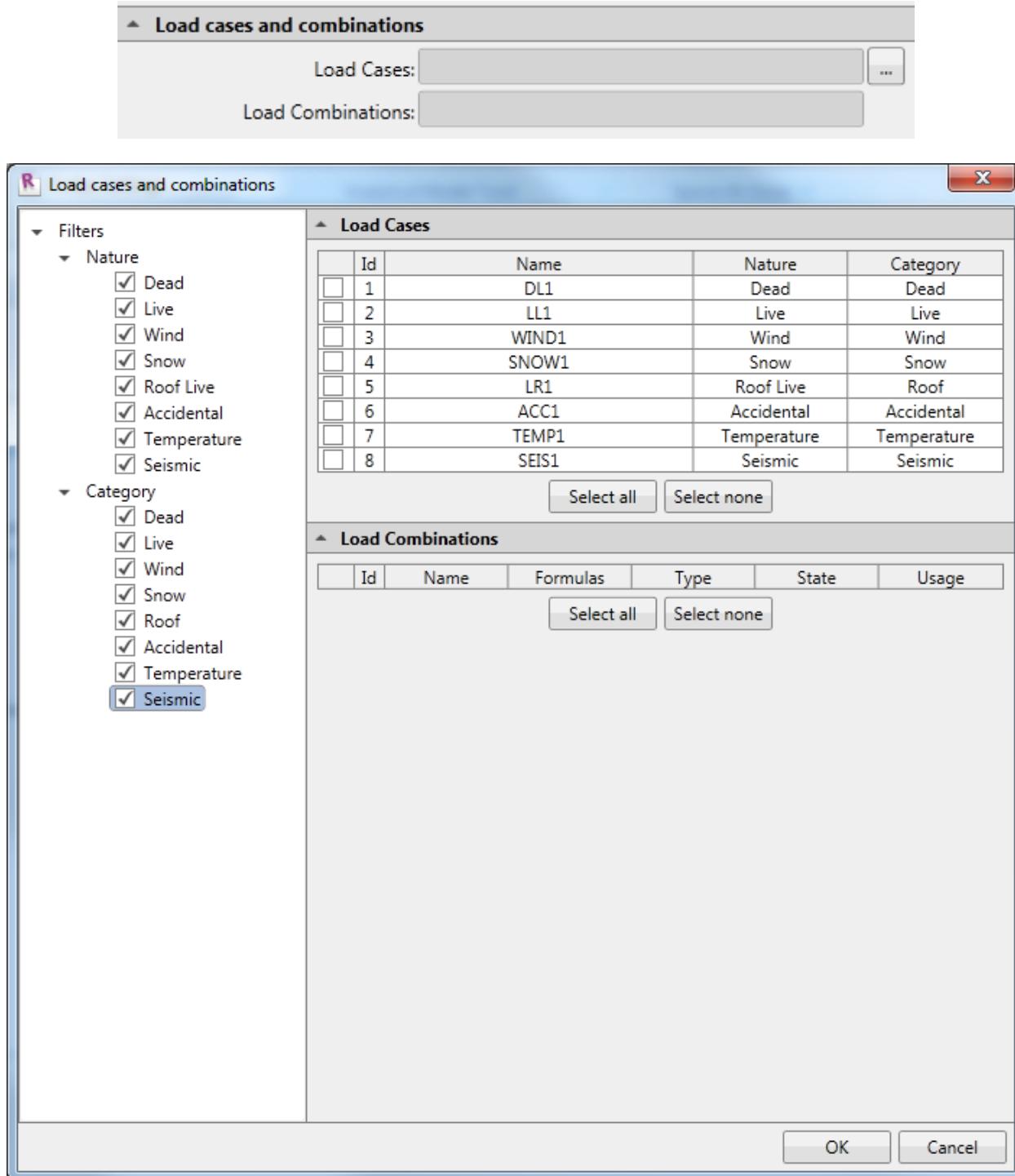
In order to learn if “Break process” is required use `ProgressBreakInvoked` method:

Code region

```
if(NotificationService.ProgressBreakInvoked())
```

3.5.6 Load Combinations and Load Cases

If Code Checking server is based on load cases and load combinations it can use the standard way of their selection and management. It can be done via `LoadCasesAndCombinationsSupport` method. If Server returns true, an additional category will appear on the dialog and offer to Revit user the possibility of selecting load cases and combinations. The access to selected items by Revit user is provided via `CalculationParamsManager` (for more information, see Storage section):



3.5.7 Results Builder

Results Builder is a component which provides common place to store structural results in Revit. ResultBuilder is based on packages. Packages may contain data for some specific analysis.

CodeChecking is based on two types of packages:

- Input – package which contains results based on which calculation will be performed (e.g. static analysis)
- Output – package which will contain results of current Code Checking server (only for Result Exploration purpose).

Code Checking server can be implemented in two ways:

- Calculation based on external package (e.g. static analysis) - recommended
- Independent calculation (all calculation is done internally)

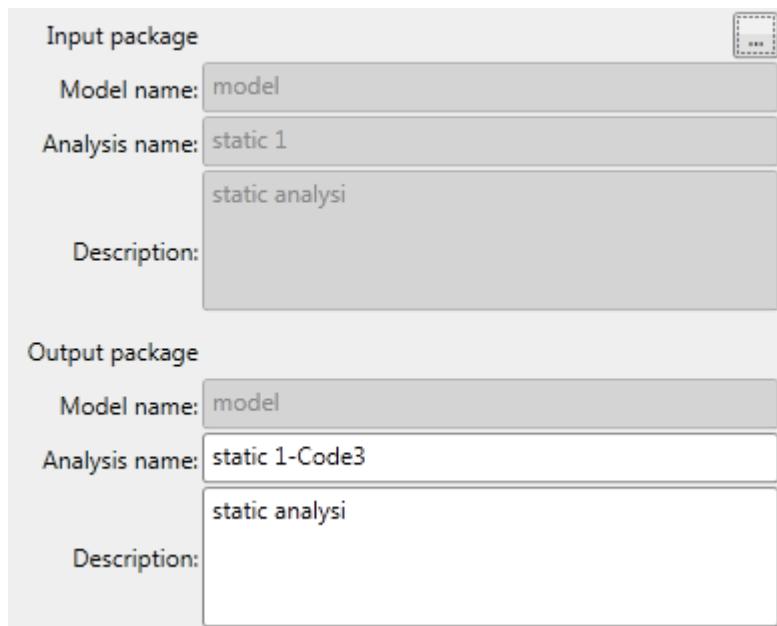
Decision about policy may be done via `ResultBuilderPackagesAsInputData` method. In case of first solution there will be additional fields with dialog in dedicated category:



No matter which way was selected the output data (Code Checking results) has to be connected with some Result Builder package. In case of Independent calculation there is only one Code Checking package for whole project. In the second scenario the output (Code Checking) package is strongly connected with input package (there is one output package assigned to one input package for each server). Additionally it is possible to store data directly in package and thanks to this Revit Results Exploration feature will be able to render them.

In order to create an output package, Results Builder needs information about it. Specifically the result types and unit system. This information is returned by server via following methods: `GetOutputPackageResultTypes`, `GetOutputPackageUnitSystem`.

Note: Code Checking results are not stored directly in Result Builder. Code Checking Framework creates packages which are connected with internal results and were the programmer can additionally store data for Results Explorer visualization (i.e. a ratio).



The access to packages may be obtained via `CalculationParamsManager`.

3.6 Documentation

Documentation creation is based on solution provided by the Extensible Storage Framework. There is independent definition of document structure which isolates user from technology. This abstract structure provides basic elements such as: tables, images, texts etc.

3.6.1 ICodeCheckingServerDocumentation

The CodeChecking application documentation may be customized via `ICodeCheckingServerDocumentation` interface.

```
Code region

public interface ICodeCheckingServerDocumentation : IResource
{
    DocumentBody BuildCalculationParamDocumentBody(Entity calcParams, Document document);
    DocumentBody BuildLabelDocumentBody(Entity label, Document document);
    DocumentBody BuildResultDocumentBody(Entity result, Element element,
                                         Document document);
}
```

The main task of the interface is to provide `DocumentBody` for:

- Labels
- Results
- Calculation parameters

`ICodeCheckingServerDocumentation` should be implemented by the main server. In case of using `Server` and `MultiStructureServer` as base classes, appropriate class should be taken from

CodeCheckingDocumentation assembly which extends them (they are called in the same way: Server, MultiStructureServer).

Server and MultiStructureServer provide default documentation based on attributes. If some manual action has to be taken manually appropriate methods should be overridden:

Code region

```
[Schema("Result", "aa106761-04f7-426b-852c-bcf35e085964")]
public class Result : SchemaClass
{
    [SchemaProperty(Unit= UnitType.UT_Length,DisplayUnit= DisplayUnitType.DUT_METERS)]
    [ValueWithCaption]
    public Double ratio{get;set;}
}

public override DocumentBody BuildResultDocumentBody(Entity result, Element element,
Document document)
{
    DocumentBody body = new DocumentBody();
    Result res = new Result();
    res.SetProperties(result);

    DocumentBody.FillBody(res, body,this,document);
    body.Elements.Add(new DocumentText("Additional text"));

    return body;
}
```

Note: ICodeCheckingServerDocumentation is derived from IResource interface which allows to return texts and images resources when needed.

3.7 Storage

Code Checking Framework is based on data set via different servers. They are stored in dedicated structures using Extensible Storage. There is no direct access to them but there are set of objects which help to easily manage them.

3.7.1 StorageService

StorageService manages all data in Code Checking Framework. It is obtained via static method:

Code region

```
StorageDocument storageDocument = service.GetStorageDocument();
```

StorageService organizes data in objects of StorageDocument type. It is able to get it as follows:

Code region

```
StorageDocument storageDocument = service.GetStorageDocument(document);
```

Each document contains three data managers which is responsible for different resources:

Code region

```
LabelsManager labels = storageDocument.LabelsManager;
ResultsManager results = storageDocument.ResultsManager;
CalculationParamsManager calculation = storageDocument.CalculationParamsManager;
```

3.7.2 Label

In order to get whole label for particular element:

Code region

```
Label label = storageDocument.LabelsManager.GetLabel(revitElement);
```

Such object contains entities of all codes that support particular category and material. In order to get appropriate data for some specific server (indicated here by label schema):

Code region

```
Entity entity = label.GetEntity(labelGUID);
```

If the user uses Extensible Storage Framework mechanism the instance of the class may be obtained directly:

Code region

```
Label codeLabel = label.GetEntity<Label>();
```

3.7.3 CalculationParameters

Calculation Parameters (Code Settings from Revit user perspective) may be got from CalculationParamsManager. As it is organized in the same way as labels and appropriate id of registered schema should be used:

Code region

```
Entity calcParamsEntity =
storageDocument.CalculationParamsManager.CalculationParams.GetEntity(CalculationGUID);
```

or via Extensible Storage Framework:

Code region

```
CalculationParameter calcParams =
storageDocument.CalculationParamsManager.
CalculationParams.GetEntity<CalculationParameter>();
```

CalculationParamsManager provides access to load combinations and cases:

Code region

```
//getting load cases and combinations
List<ElementId> loadCasesCombinations = storageDocument.CalculationParamsManager.
CalculationParams.GetLoadCasesAndCombinations(ID);
```

As well as Result Builder packages

Code region

```
//getting input package
ResultsPackage package =
storageDocument.CalculationParamsManager.CalculationParams.GetInputResultPackage(ID);

//getting output package
storageDocument.ResultsManager.GetOutputResultPackage(ID);

//getting output package builder
ResultsPackageBuilder builder = storageDocument.CalculationParamsManager.
CalculationParams.GetOutputResultPackageBuilder(ID);
//do something here
builder.Finish();
```

3.7.4 Results

Results should be set (and get) via ResultsManager:

Code region

```
Entity entityResult =...
storageDocument.ResultsManager.SetResult(entityResult, element);
```

When result is set, it is marked as up-to-date. This flag is changed when label or package status change. If the user want to outdated it on his own (for instance using an updater when some changes were done in the model) the OutDateResults method should be used:

Code region

```
storageDocument.ResultsManager.OutDateResults(serverID, el);
```

3.7.5 Result Status

ResultStatus was created in order to provide to developers possibility to inform Revit users about warnings, errors and general state of results for specific element.

Code region

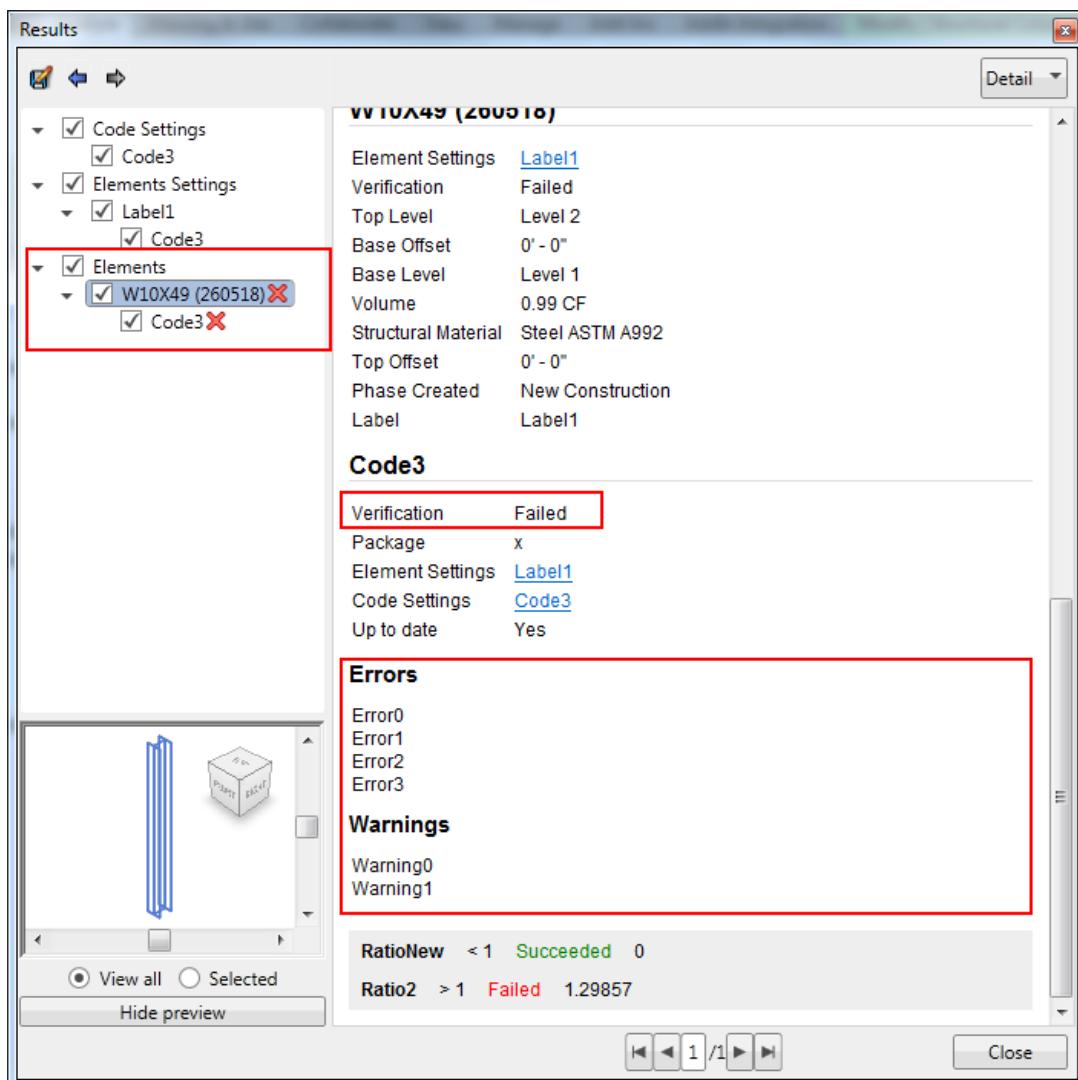
```
ResultStatus status = new ResultStatus(ServiceID);
Result myResult = new Result();

for (int i = 0; i < 4; i++)
    status.AddError("Error" + i.ToString());
for (int i = 0; i < 2; i++)
    status.AddWarning("Warning" + i.ToString());

storageDocument.ResultsManager.SetResult(myResult.GetEntity(), element, status);

//or

storageDocument.ResultsManager.SetResult(myResult.GetEntity(), element);
storageDocument.ResultsManager.SetResultStatus(element, status);
```



Note: ResultStatus informs NotificationService about added errors/warning, thanks to this for instance progress will show that some errors were detected.

3.8 UI

The main goal of Code Checking Framework is to provide consistent UI and controls to define element and code settings in different servers. That is why it was based on Extensible Storage Framework.

3.8.1 ICodeCheckingServerUI

The developer decides about UI of CodeChecking by providing object implementing ICodeCheckingServerUI interface:

Code region

```
public interface ICodeCheckingServerUI : IServerUI, IResource
{
    Layout BuildCalculationParamsLayout(Schema schema, Document document);
    Layout BuildLabelLayout(Schema schema, Document document);
    string GetHelp(string cultureInfo);
}
```

This interface may be implemented by the main server (the same class which implements `ICodeCheckingServer`) or by additional object registered in the Code Checking Framework. This registration should take place in UI application:

Code region

```
public Result OnStartup(UIControlledApplication application)
{
    ServerUI server = new ServerUI();
    ServiceUI.BindUIServerWithDBServer(Server.ID, server);
    return Autodesk.Revit.UI.Result.Succeeded;
}
```

Note: `ICodeCheckingServerUI` is derived from `IResource` interface which allows to return text and image resources when needed.

3.8.2 BuildLayout

`BuildLayout` methods return layouts for Labels and Calculation Parameters. This mechanism is based on Extensible Storage Framework and there are many ways of defining layouts. One of them is presented below:

Code region

```
public class Label : SchemaClass
{
    [TextBoxAttribute(Category = "Category1")]
    public String a { get; set; }

    Layout layout = Layout.Generate(typeof(Label), this);
```

3.8.3 GetHelp

`GetHelp` method returns location to help file which is run through Windows. It could be a network location or any file registered in Windows – pdf, html, doc, rtf.

Code region

```
public string GetHelp(string cultureInfo)
{
    string dir = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
    return System.IO.Path.Combine(dir, "Help.html");
}
```

3.8.4 ServerUI and MultiStructureServerUI

As in case of main server Code Checking Framework provides ready class for server UI which implements by default most of methods. They are `Server` and `MultiStructureServer`. If the whole application is based on attributes (Extensible Storage Framework) and no manual action is needed in UI, there is nothing special to do:

Code region

```
public class ServerUI : Server<CalculationParameter, Label>
{
    public override string GetResource(string key, string context)
    {
        string txt = Resources.ResourceManager.GetString(key);

        if (!string.IsNullOrEmpty(txt))
            return txt;

        return key;
    }

    public override Uri GetResourceImage(string key, string context)
    {
        return new Uri(@"pack://application:,,,/CodeCheckingServer;component/" + key);
    }
}
```

Otherwise appropriate methods should be overridden:

Code region

```
public class Label : SchemaClass
{
    [TextBoxAttribute(Category = "Category1")]
    public String a { get; set; }
}

public class ServerUI : Server<CalculationParameter, Label>
{
    public override Layout BuildCalculationParamsLayout(Schema schema, Document document)
    {
        Layout layout = Layout.Build(typeof(Label), this);

        Category category = layout.GetCategory("Category1");
        Image img = new Image() { Source = new Uri("C:\\\\image.png") };
        category.Controls.Add(img);

        return layout;
    }
}
```

3.9 Example

3.9.1 Goal:

As an example part of Eurocode 3 will be implemented, particularly Tension. Code will be dedicated to columns. Additional assumption will be the definition of internal force inside label (generally it should be taken from Revit project using Results Builder component).

The resistance of tension members is calculated as follows:

- For yielding of the gross cross-section

$$N_{pl,Rd} = \frac{Af_y}{\gamma_{M0}}$$

- For the ultimate resistance of the net cross-section

$$N_{u,Rd} = \frac{0.9A_{net}f_u}{\gamma_{M2}}$$

Where net area may be calculated as:

$$A_{net} = A - A_{holes}$$

- As a final result the smaller value is taken.

3.9.2 Design

Based on presented formula we can split parameters into group:

- Calculation Parameters
 - γ_{M0}
 - γ_{M2}
- Label
 - A_{holes}
 - N
- Results
 - $N_{pl,Rd}$
 - $N_{u,Rd}$
 - A_{net}
 - Ratio
- Rest of parameters may be taken from Revit model and also put to Results:
 - A
 - f_y
 - f_u

3.9.3 Project

Let's create EurocodeTension project as regular C# class library project.

Note: The whole project may be generated using template provided with SDK which will allow preparing it faster. For purpose of this example it will be created manually to show all important dependencies.

References

On start we have to add following references:

- CodeChecking
- CodeChecking.Documentation
- CodeChecking.Storage
- CodeChecking.UI
- ExtensibleStorageFramework
- ExtensibleStorageFramework.Documentation
- ExtensibleStorageFramework.UI
- RevitAPI
- RevitAPIUI

The project structure will consist of following items:

Applications

- RevitApplicationDB.cs
- RevitApplicationUI.cs

Servers

- Server.cs
- ServerUI.cs

Data

- CalculationParameter.cs
- Label.cs
- Result.cs

3.9.4 Data

Let's start with data. All classes will be derived from `SchemaClass`.

Note: Classes may be created using `SchemaClass` templates provided with SDK

Code region

```
[Schema("CalculationParam", "af7d939b-2bb4-4e73-b187-4a690a921223")]
public class CalculationParameter : SchemaClass
{
    public CalculationParameter()
    {
    }
}

[Schema("Label", "2e29b7d2-2008-4db0-8942-2c86513a70bf")]
public class Label : SchemaClass
{
    public Label()
    {
    }
}

[Schema("Result", "60494c0c-af4e-4435-bac7-b146d5daa6b1")]
public class Result : SchemaClass
{
    public Result()
    {
    }
}
```

Now we can add properties with Extensible Storage Framework attributes for UI and documentation (meaning of them won't be described here, please refer to Extensible Storage Framework for more information):

Code region

```
[Schema("CalculationParam", "af7d939b-2bb4-4e73-b187-4a690a921223")]
public class CalculationParameter : SchemaClass
{
    [SchemaProperty(Unit=UnitType.UT_Number,DisplayUnit=DisplayUnitType.DUT_1_RATIO)]
    [UnitTextBox(
        Description = "@g{M0}",
        ValidateMinimumValue=true,
        MinimumValue=0,
        AttributeUnit = DisplayUnitType.DUT_1_RATIO,
        Category="Data")]
    [ValueWithName(Name = "@g{M0}")]
    public Double PartialFactor0{get;set;}

    [SchemaProperty(Unit = UnitType.UT_Number, DisplayUnit=DisplayUnitType. DUT_1_RATIO)]
    [UnitTextBox(
        Description = "@g{M2}",
        ValidateMinimumValue = true,
        MinimumValue = 0,
        AttributeUnit = DisplayUnitType.DUT_1_RATIO,
        Category = "Data")]
    [ValueWithName(Name = "@g{M2}")]
    public Double PartialFactor2 { get; set; }

    public CalculationParameter()
    {
    }
}
```

Code region

```
[Schema("Label", "2e29b7d2-2008-4db0-8942-2c86513a70bf")]
public class Label : SchemaClass
{
    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Section_Area,
                    DisplayUnit = DisplayUnitType.DUT_SQUARE_METERS)]
    [UnitTextBox(
        Description = "A{holes}",
        ValidateMinimumValue = true,
        MinimumValue = 0,
        AttributeUnit = DisplayUnitType.DUT_SQUARE_METERS,
        Category = "Data")]
    [ValueWithName(Name="A{holes}")]
    public Double Aholes { get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Force,
                    DisplayUnit = DisplayUnitType.DUT_NEWTONS)]
    [UnitTextBox(
        ValidateMinimumValue = true,
        MinimumValue = 0,
        AttributeUnit = DisplayUnitType.DUT_NEWTONS,
        Category = "Data")]
    [ValueWithName]
    public Double N { get; set; }

    public Label()
    {
    }
}
```

Code region

```
[Schema("Result", "60494c0c-af4e-4435-bac7-b146d5daa6b1")]
public class Result : SchemaClass
{
    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Force,
        DisplayUnit = DisplayUnitType.DUT_NEWTONS)]
    [ValueWithName(Name = "N{pl,Rd}", Index = 5)]
    public Double Nplrd { get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Force,
        DisplayUnit = DisplayUnitType.DUT_NEWTONS)]
    [ValueWithName(Name = "N{u,Rd}", Index = 6)]
    public Double Nurd { get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Force,
        DisplayUnit = DisplayUnitType.DUT_NEWTONS)]
    [ValueWithName(Name = "N{Rd}", Index = 7)]
    public Double Nrd { get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Section_Area,
        DisplayUnit = DisplayUnitType.DUT_SQUARE_METERS)]
    [ValueWithName(Name = "A{net}", Index = 2)]
    public Double Anet { get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Section_Area,
        DisplayUnit = DisplayUnitType.DUT_SQUARE_METERS)]
    [ValueWithName(Index = 0)]
    public Double A{ get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Stress,
        DisplayUnit = DisplayUnitType.DUT_PASCALS)]
    [ValueWithName(Name = "f{y}", Index = 3)]
    public Double fy { get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Stress,
        DisplayUnit = DisplayUnitType.DUT_PASCALS)]
    [ValueWithName(Name = "f{u}", Index = 4)]
    public Double fu { get; set; }

    [SchemaProperty(Unit = Autodesk.Revit.DB.UnitType.UT_Number,
        DisplayUnit = DisplayUnitType.DUT_1_RATIO)]
    [Ratio(Index = 8)]
    public Double Ratio { get; set; }

    public Result()
    {
    }
}
```

3.9.5 Servers

Now the data structure is ready and we can create the Server and the ServerUI.

3.9.5.1 Server

Our code is dedicated to one type of element and material so we can derive our Server from base class provided with Code Checking Framework. It will allow us to skip implementation of many methods.

Code region

```
public class Server : Documentation.Server<CalculationParameter, Label, Result>
{
    public static readonly Guid ID =
        new Guid("7d30a3b5-29e6-48f5-a298-f9152bf7f5ab");

    public override void Verify(Autodesk.Revit.DB.CodeChecking.ServiceData data)
    {
    }

    public override IList<BuiltInCategory> GetSupportedCategories(
        StructuralAssetClass material)
    {
        return new List<BuiltInCategory>() { BuiltInCategory.OST_ColumnAnalytical};
    }

    public override IList<StructuralAssetClass> GetSupportedMaterials()
    {
        return new List<Autodesk.Revit.DB.StructuralAssetClass>(){
            Autodesk.Revit.DB.StructuralAssetClass.Metal};
    }

    public override string GetDescription()
    {
        return "EurocodeTension";
    }

    public override string GetName()
    {
        return "EurocodeTension";
    }

    public override Guid GetServerId()
    {
        return ID;
    }

    public override string GetVendorId()
    {
        return "John Smith";
    }

    public override bool LoadCasesAndCombinationsSupport()
    {
        return false;
    }

    public override bool ResultBuilderPackagesAsInputData()
    {
        return false;
    }
}
```

Note: This example doesn't support Results Builder Packages and Load Cases and Combinations so `LoadCasesAndCombinationsSupport()` and `ResultBuilderPackagesAsInputData()` methods return false.

3.9.5.2 ServerUI

In case of ServerUI we can just leave it empty:

Code region

```
public class ServerUI :  
    Autodesk.Revit.UI.CodeChecking.Server<CalculationParameter, Label>  
{  
}
```

3.9.6 Revit Applications

3.9.6.1 RevitApplicationDB

RevitApplicationDB is run at the very start of Revit. Our main task is to register our server in Code Checking service.

Code region

```
public class RevitApplicationDB : IExternalDBApplication  
{  
    Public ExternalDBApplicationResult OnShutdown(ControlledApplication application)  
    {  
        return ExternalDBApplicationResult.Succeeded;  
    }  
  
    public ExternalDBApplicationResult OnStartup(ControlledApplication application)  
    {  
        application.ApplicationInitialized += new  
EventHandler<ApplicationInitializedEventArgs>(application_ApplicationInitialized);  
        return ExternalDBApplicationResult.Succeeded;  
    }  
  
    void application_ApplicationInitialized(object sender,  
    ApplicationInitializedEventArgs e)  
    {  
        Server server = new Server();  
        Autodesk.Revit.DB.CodeChecking.Service.AddServer(server);  
    }  
}
```

Server has to be registered when application was already initialized (just to be sure that Code Checking Framework is already loaded).

In order to launch our application we have to create manifest file and copy it to Revit Addins:

Code region

```
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="DBApplication">
    <Name>EurocodeTension</Name>
    <Assembly> ..\EurocodeTension.dll</Assembly>
    <AddInId>4834f5d8-0677-4794-a39e-6955a2901638</AddInId>
    <FullClassName>EurocodeTension.RevitApplicationDB</FullClassName>
    <VendorId>John Smith</VendorId>
    <VendorDescription>John Smith</VendorDescription>
  </AddIn>
</RevitAddIns>
```

3.9.6.2 RevitApplicationUI

In case of RevitApplicationUI we have to register and initialize there our ServerUI:

Code region

```
class RevitApplicationUI : Autodesk.Revit.UI.IExternalApplication
{
  public Autodesk.Revit.UI.Result OnShutdown(UIControlledApplication application)
  {
    return Result.Succeeded;
  }

  public Result OnStartup(UIControlledApplication application)
  {
    ServerUI server = new ServerUI();
    ServiceUI.BindUIServerWithDBServer(Server.ID, server);
    return Result.Succeeded;
  }
}
```

Code region

```
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="Application">
    <Name>EurocodeTensionUI</Name>
    <Assembly>..\EurocodeTension.dll</Assembly>
    <AddInId>3d4180ad-dba7-4cf6-ab5c-d97b2c0307b5</AddInId>
    <FullClassName>EurocodeTension.RevitApplicationUI</FullClassName>
    <VendorId>John Smith</VendorId>
    <VendorDescription>John Smith</VendorDescription>
  </AddIn>
</RevitAddIns>
```

3.9.7 Calculation

Now when everything is ready we can implement calculation algorithm and fill `Verify()` method of our Server.

First let's iterate through all elements from selection and skip elements which our code doesn't support:

Code region

```
public override void Verify(ServiceData data)
{
    foreach (Element element in data.Selection)
    {
        BuiltInCategory category = Tools.GetCategoryOfElement(element);
        StructuralAssetClass material = Tools.GetClassMaterialOfElement(element);

        if (material != StructuralAssetClass.Metal ||
            category != BuiltInCategory.OST_ColumnAnalytical)
            continue;

    }
}
```

Now we can get calculation parameters and labels:

Code region

```

public override void Verify(ServiceData data)
{
    StorageService service = StorageService.GetStorageService();
    StorageDocument storageDocument = service.GetStorageDocument(data.Document);
    CalculationParameter myParams = storageDocument.CalculationParamsManager.
        CalculationParams.GetEntity<CalculationParameter>(data.Document);

    foreach (Element element in data.Selection)
    {
        BuiltInCategory category = Tools.GetCategoryOfElement(element);
        StructuralAssetClass material = Tools.GetClassMaterialOfElement(element);

        if (material != StructuralAssetClass.Metal ||
            category != BuiltInCategory.OST_ColumnAnalytical)
            continue;

        Autodesk.Revit.DB.CodeChecking.Storage.Label ccLabel =
            storageDocument.LabelsManager.GetLabel(element);

        if (ccLabel != null)
        {
            Label myLabel = ccLabel.GetEntity<Label>(data.Document);

            if (myLabel != null)
            {
                Calculate(myParams, myLabel, element, storageDocument.ResultsManager );
            }
        }
    }
}

```

Additionally there were created `Calculate` method where specific element will be calculated:

Code region

```

void Calculate(CalculationParameter parameters, Label label, Element ele-
ment, ResultsManager manager)
{
}

```

The first thing which is needed to perform calculation is material and its characteristics:

Code region

```
Material mat = Tools.GetMaterialOfElement(element);
PropertySetElement propertySetElement = mat.Document.GetElement(mat.StructuralAssetId) as PropertySetElement;
StructuralAsset structuralAsset = propertySetElement.GetStructuralAsset();
result.fy = UnitUtils.ConvertFromInternalUnits(structuralAsset.MinimumYieldStress, DisplayUnitType.DUT_PASCALS);
result.fu = UnitUtils.ConvertFromInternalUnits(structuralAsset.MinimumTensileStrength, DisplayUnitType.DUT_PASCALS);
```

The second thing important for calculation is area of section. Let's get it directly from parameters:

Code region

```
AnalyticalModel analyticalModel = element as AnalyticalModel;
FamilyInstance familyInstance = element.Document.
GetElement(analyticalModel.GetElementId()) as FamilyInstance;
Parameter parameterA = familyInstance.Symbol.get_Parameter("A");

if (parameterA != null)
{
    result.A = UnitUtils.ConvertFromInternalUnits(parameterA.AsDouble(),
        DisplayUnitType.DUT_SQUARE_METERS);
}
```

Now we can perform calculation:

Code region

```
result.Anet = result.A - label.Aholes;
result.Nplrd = result.A * result.fy / parameters.PartialFactor0;
result.Nurd = 0.9 * result.Anet * result.fu / parameters.PartialFactor2;
result.Nrd = Math.Min(result.Nplrd, result.Nurd);
result.Ratio = label.N / result.Nrd;
```

Additionally we can set status to element:

Code region

```
ResultStatus status = new ResultStatus(ID);
status.SetStatusRatioBased(result.Ratio);

manager.SetResult(result.GetEntity(), element, status);
```

Entire calculation method may be found below:

Code region

```
void Calculate(CalculationParameter parameters, Label label,
Element element, ResultsManager manager)
{
    Result result = new Result();

    Material mat = Tools.GetMaterialOfElement(element);
    PropertySetElement propertySetElement =
        mat.Document.GetElement(mat.StructuralAssetId) as PropertySetElement;
    StructuralAsset structuralAsset = propertySetElement.GetStructuralAsset();
    result.fy = UnitUtils.ConvertFromInternalUnits(structuralAsset.MinimumYieldStress,
DisplayUnitType.DUT_PASCALS);
    result.fu = UnitUtils.ConvertFromInternalUnits(
        structuralAsset.MinimumTensileStrength, DisplayUnitType.DUT_PASCALS);

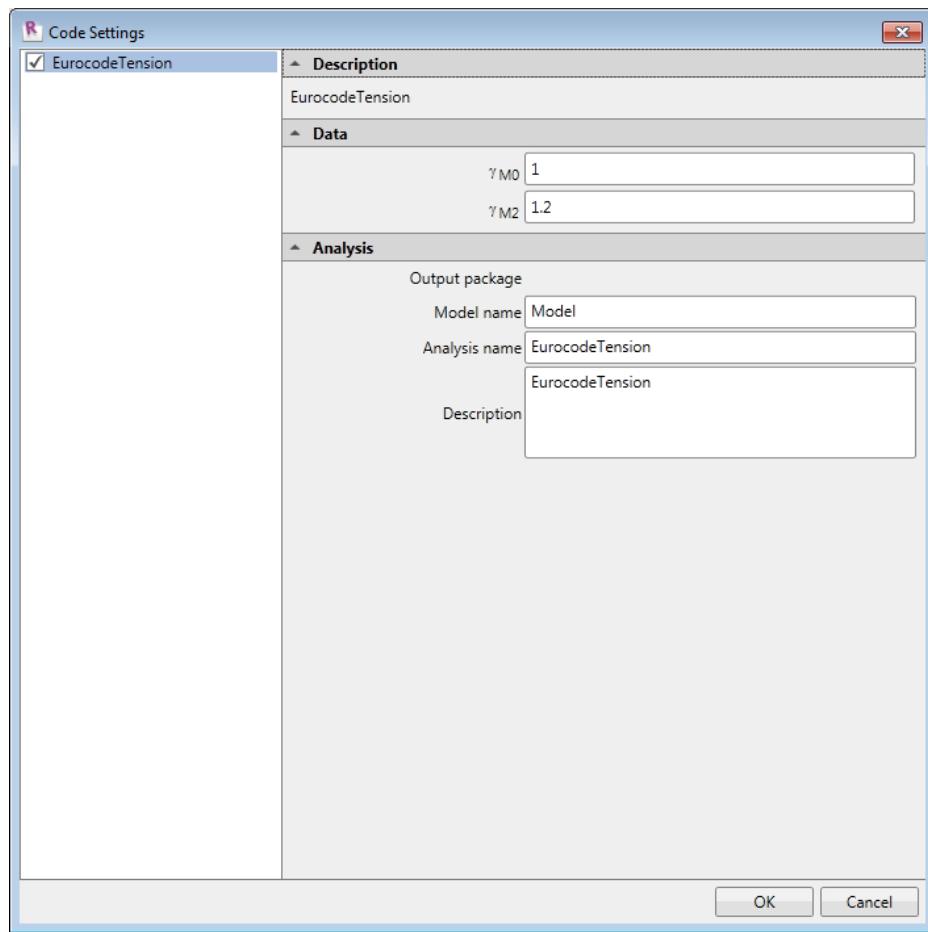
    AnalyticalModel analyticalModel = element as AnalyticalModel;
    FamilyInstance familyInstance = element.Document.
GetElement(analyticalModel.GetElementId()) as FamilyInstance;
    Parameter parameterA = familyInstance.Symbol.get_Parameter("A");

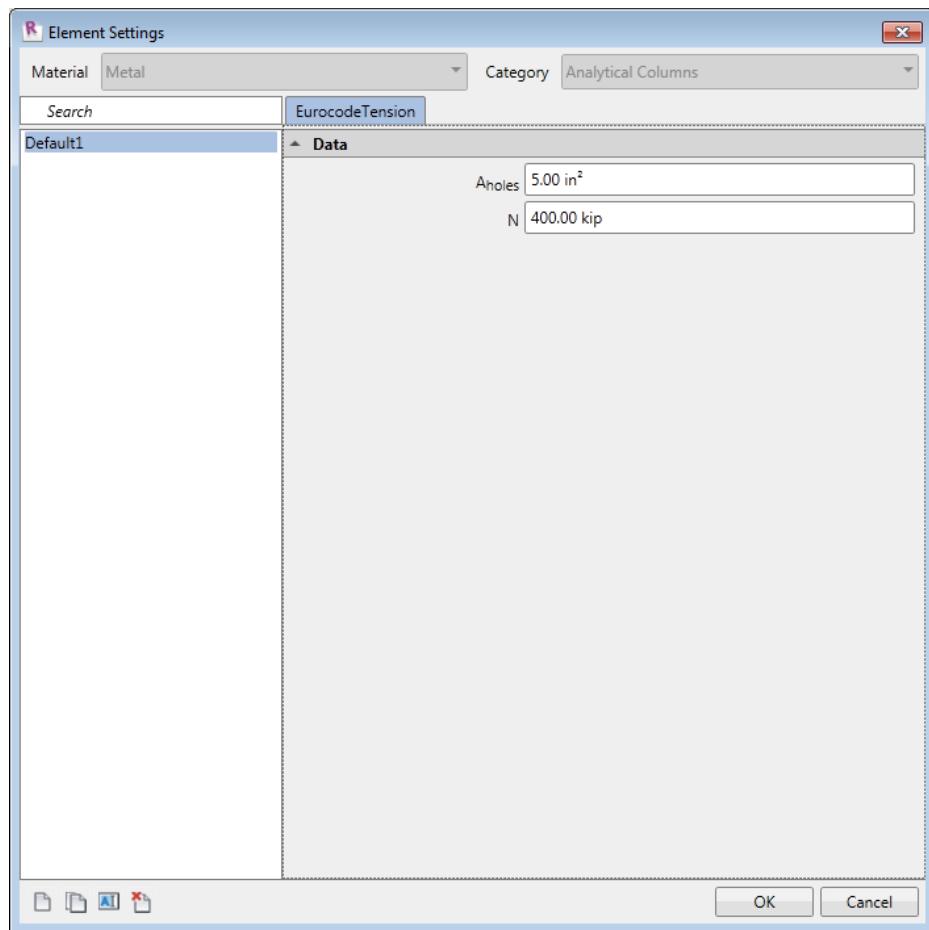
    if (parameterA != null)
    {
        result.A = UnitUtils.ConvertFromInternalUnits(parameterA.AsDouble(),
DisplayUnitType.DUT_SQUARE_METERS);
        result.Anet = result.A - label.Aholes;
        result.Nplrd = result.A * result.fy / parameters.PartialFactor0;
        result.Nurd = 0.9 * result.Anet * result.fu / parameters.PartialFactor2;
        result.Nrd = Math.Min(result.Nplrd, result.Nurd);
        result.Ratio = label.N / result.Nrd;

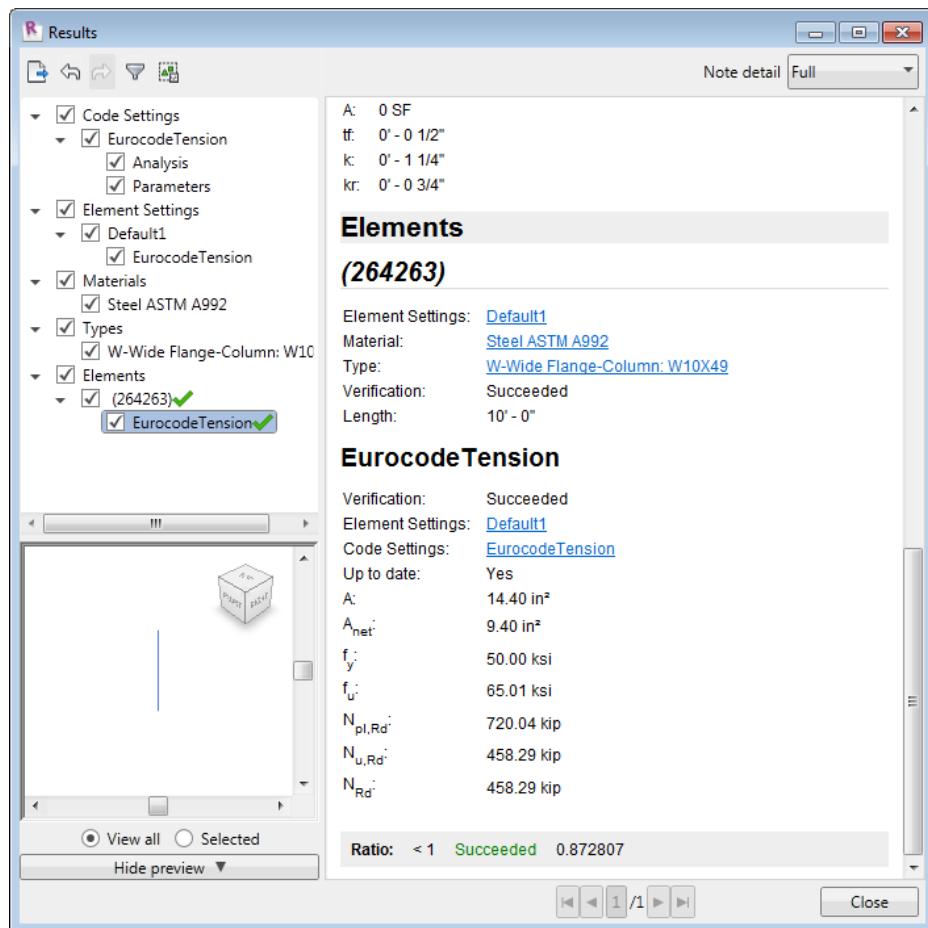
        ResultStatus status = new ResultStatus(ID);
        status.SetStatusRatioBased(result.Ratio);

        manager.SetResult(result.GetEntity(), element, status);
    }
}
```

As a final result we have whole code implemented. Now we can use it in Revit.







3.9.8 Load cases, combinations and result builder

At the very end we can enable features we gave up at the beginning. First let's enable load cases and combinations:

Code region

```
public override bool LoadCasesAndCombinationsSupport()
{
    return true;
}
```

As result code settings dialog contains an additional category:



Now we can use it in our code:

Code region

```
List<ElementId> loadCasesAndCombinations = storageDocument.CalculationParamsManager.CalculationParams.GetLoadCasesAndCombinations(ID);

foreach (ElementId id in loadCasesAndCombinations)
{
    Element loadCaseCombinationElement = data.Document.GetElement(id);

    LoadCase loadCase = loadCaseCombinationElement as LoadCase;

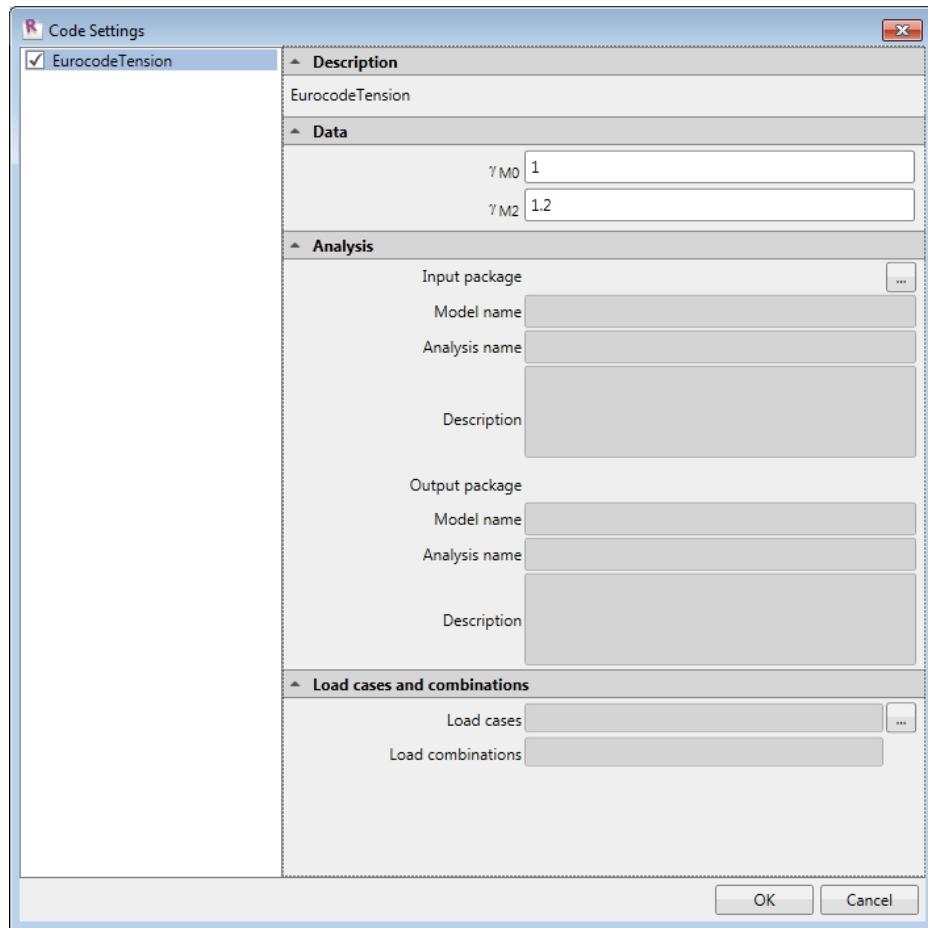
    LoadCombination loadCombination = loadCaseCombinationElement as LoadCombination;
}
```

Now it is time for using ResultsBuilder component. Till now our server wasn't based on any external package that provides input data (e.g. static analysis results). This resulted in definition of one output package in the project. Now we will switch to second scenario by:

Code region

```
public override bool ResultBuilderPackagesAsInputData()
{
    return true;
}
```

As a result our dialog looks as follows:



In order to get input package set by the user following code should be implemented:

Code region

```
ResultsPackage inputPackage = storageDocument.CalculationParamsManager.
CalculationParams.GetActivePackage(ID);

var results = inputPackage.GetLineGraphs(data.Selection.Select(o => o.Id).ToList(),
loadCasesAndCombinations, new List<LinearResultType> {LinearResultType.Fx});
```

If we want to add some data to Results Builder manually (i.e. for Results Explorer visualization) we have to get appropriate output package set by the user:

Code region

```
ResultsPackageBuilder builder = storageDocument.CalculationParamsManager.
CalculationParams.GetOutputResultPackageBuilder(ID);
//do something here
builder.Finish();
```

Note: It is important to remember to close Results Builder after modification.

We have to remember as well about returning appropriate information about our output packages via server methods:

Code region

```

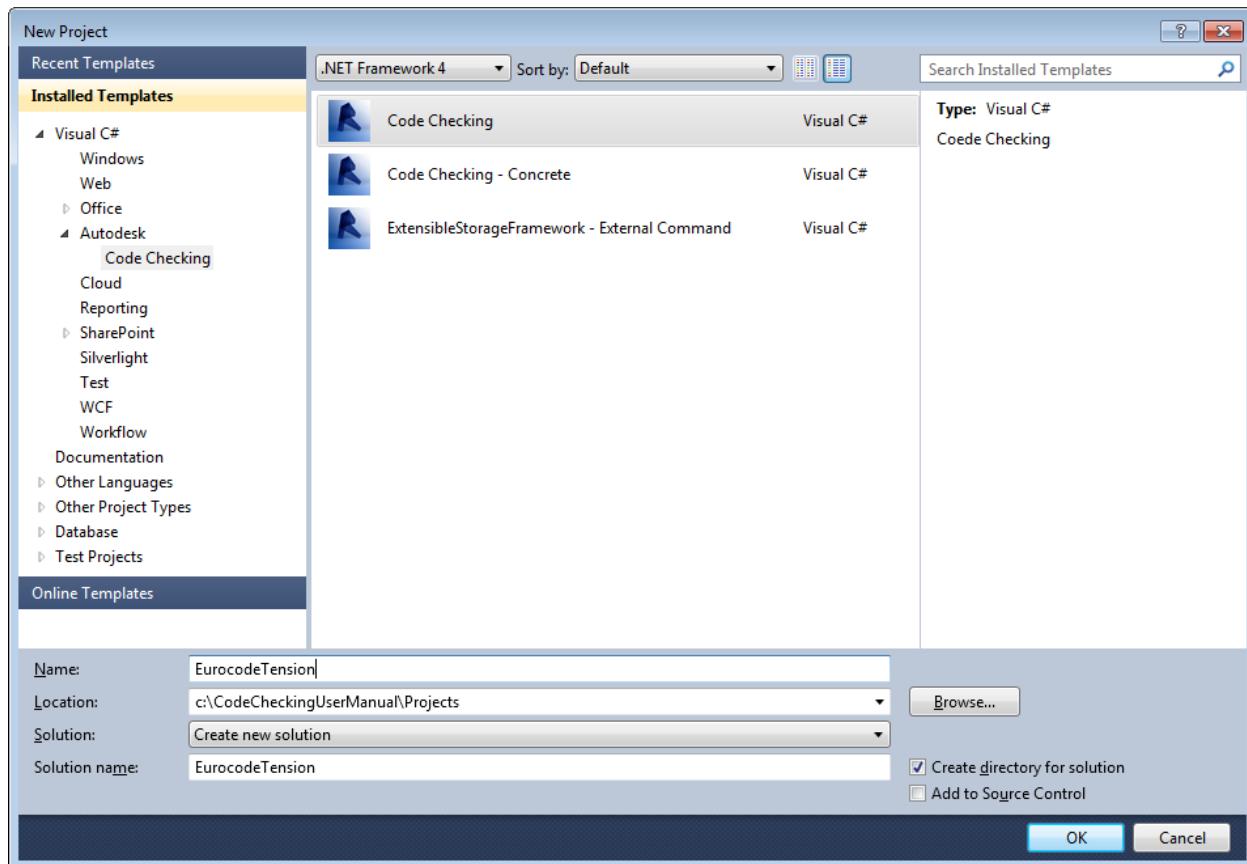
public ResultsPackageTypes GetOutputPackageResultTypes()
{
    return ResultsPackageTypes.RequiredReinforcement;
}

public UnitsSystem GetOutputPackageUnitSystem()
{
    Return UnitsSystem.Metric;
}

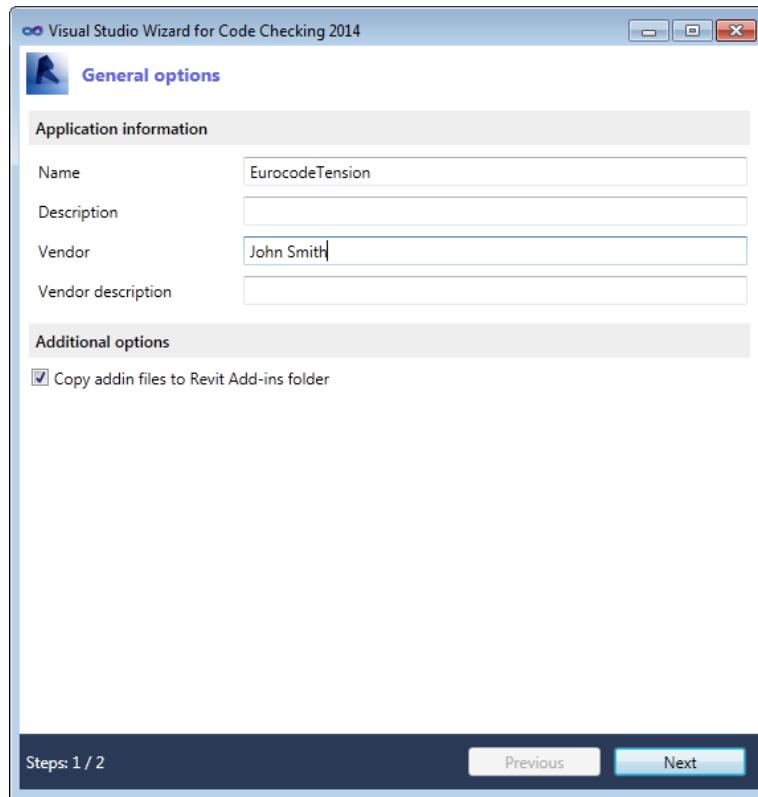
```

3.9.9 Wizard

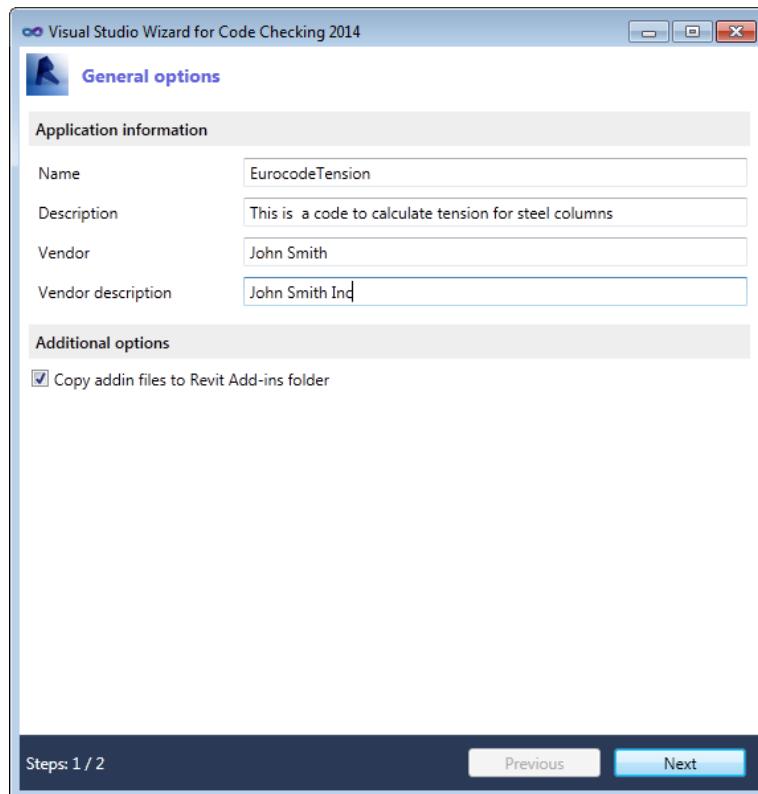
The whole example was made manually from the start till the end. There is a way to make it faster using dedicated wizard for Visual Studio.

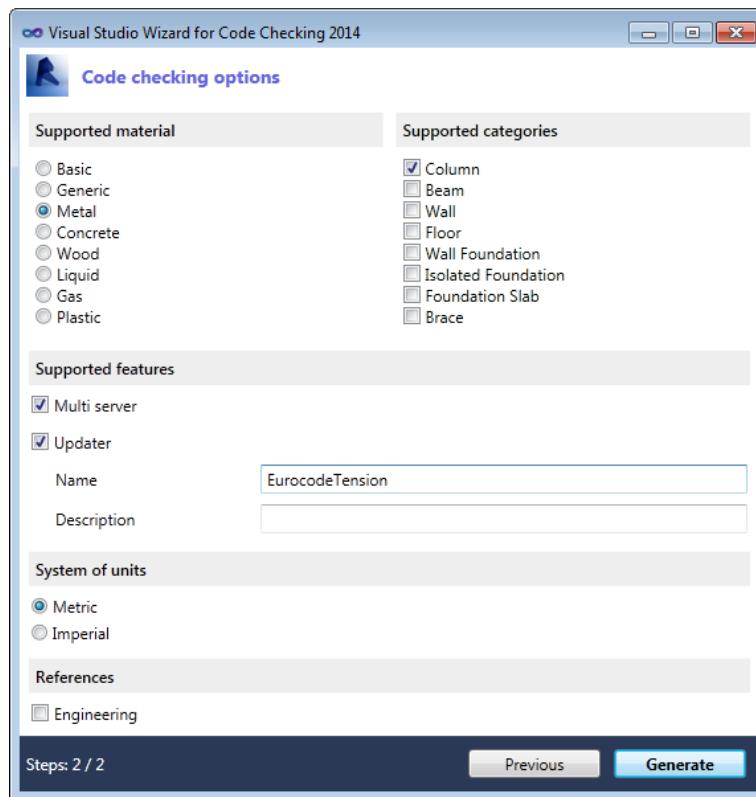


After selecting a Code Checking template, the wizard dialog is opened:



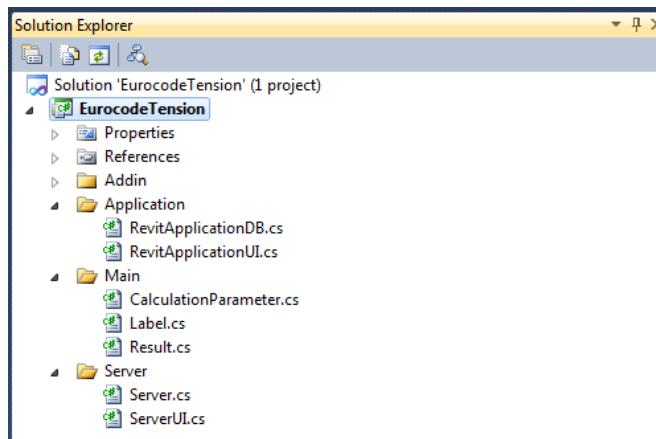
We fill information we need:





And click Generate button.

As a result the whole project will be generated with all files and appropriate references.



In case of EurocodeTension project the developers have to:

- Fill Label, Result, CalculationParameter structures with appropriate properties and attributes
- Fill Verify method inside Server class.

And that's it!

Rest actions we took manually are made by the wizard, even the automatic copy of the manifest file to Revit Addins folder.

4 CODE CHECKING FRAMEWORK CONCRETE

As part of the Code Checking SDK, a concrete template and associated wizard is provided. Following sections describe how to use Code Checking Concrete template to create Code Checking application dedicated to Concrete Design.

4.1 Description

Code Checking Concrete template and wizard are provided to assist developer in creating Code Checking application for Reinforcement Concrete design.

Developer could choose some options which control the content of the generated project. Thus by switching different options developer can decide which elements will be included in the project and which won't, and consequently which he will have to be developed by himself.

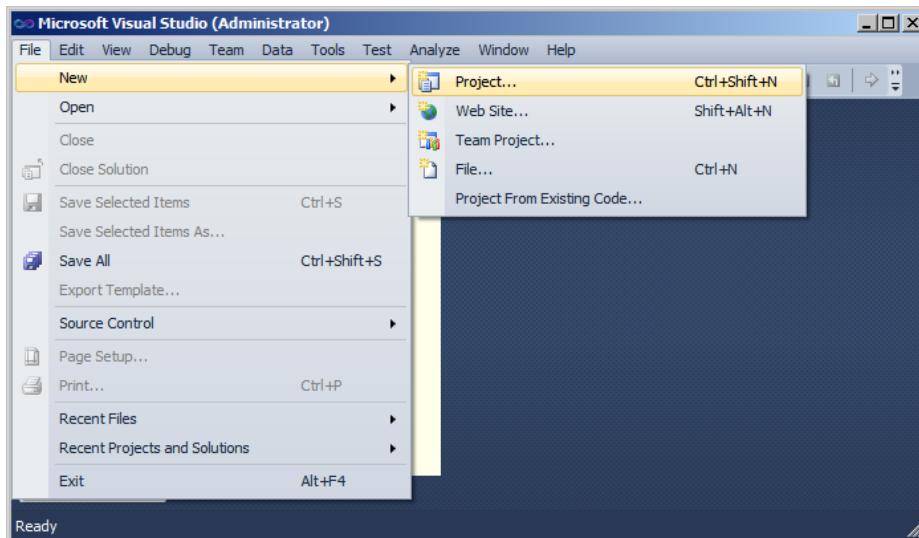
Options are divided into three categories:

- General options
- Code Checking options
- General concrete options

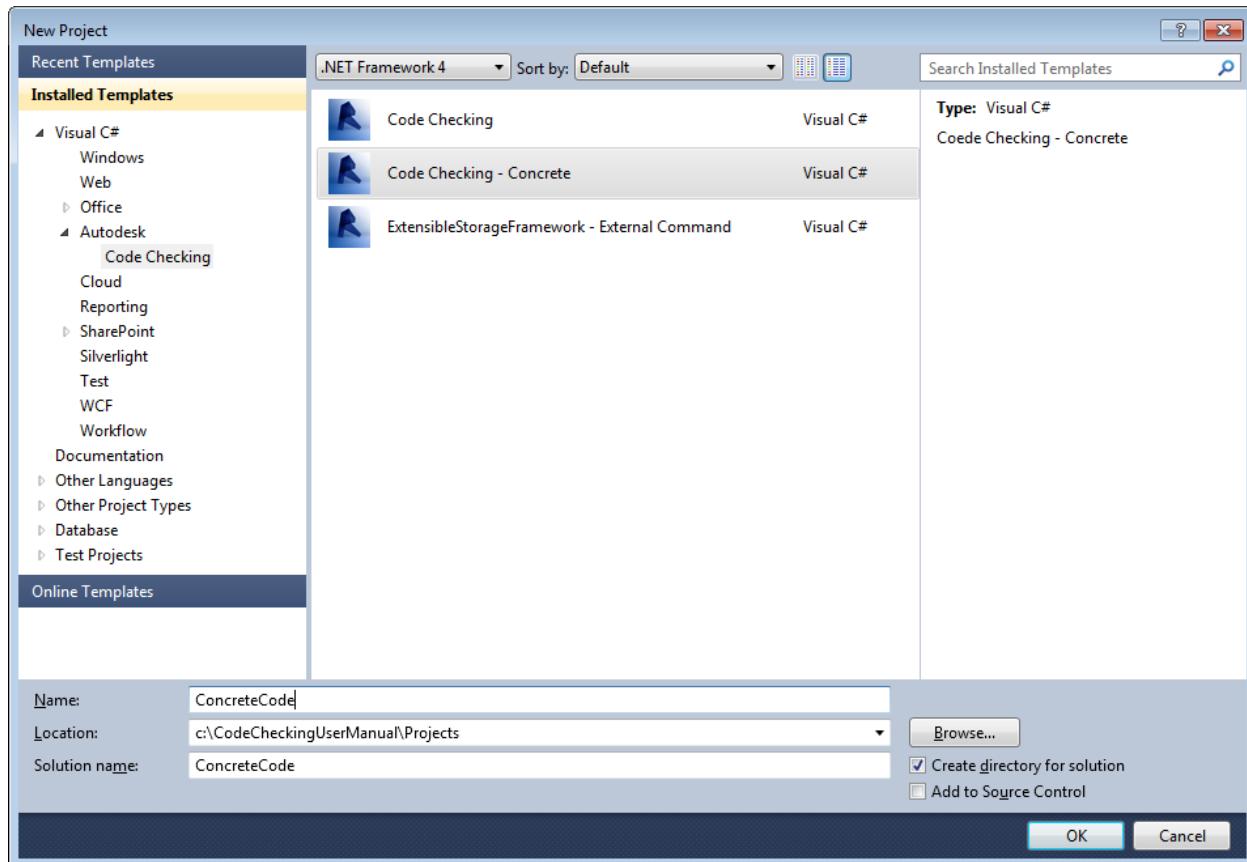
The first two groups apply to all types of application using the Code Checking Framework while the third one is dedicated to required reinforcement calculation (beam and column) project.

4.2 Project Creation

After a proper installation of the Code Checking SDK, the “Code Checking – Concrete” template is available as a C# Visual studio template.



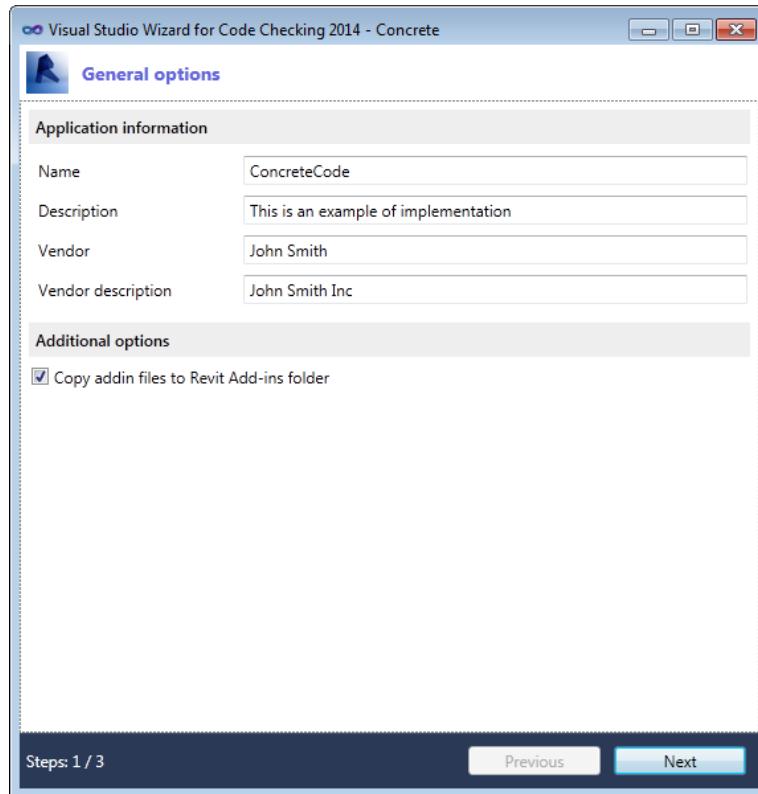
After choosing the new Project the standard Visual Studio window is displayed and the Code Checking-Concrete template is available in Visual C# \ Autodesk\Code Checking branch.



After setting the right project type and setting project name, project location and solution name the wizard is run and the developer is guided through the rest of the process to choose appropriate option that fit his need.

4.2.1 Setting general options

General Options dialog is the first dialog to be displayed after project creation.



As described earlier on this document, general options include two things:

- Revit Application information (manifest file, application guid,..)
- Option to copy the manifest files during project creation to appropriate folder.

Application information options are a collection of strings that are directly copied into the code where needed and inside manifest files.

Code region

```
#region IExternalServer Members

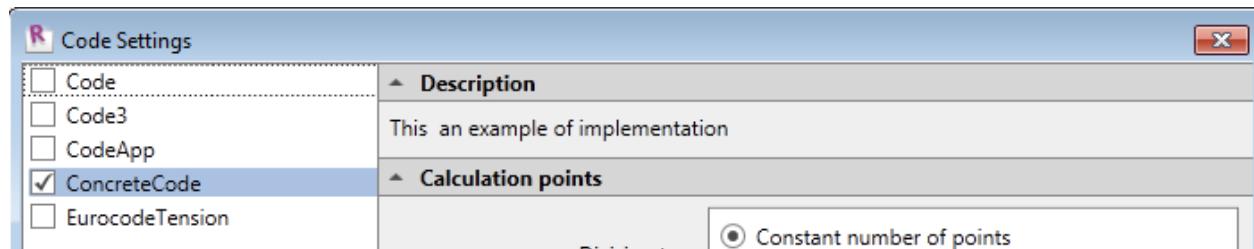
    public override string GetDescription()
    {
        return "This is an example of implementation";
    }

    public override string GetName()
    {
        return "ConcreteCode";
    }

    public override Guid GetServerId()
    {
        return ID;
    }

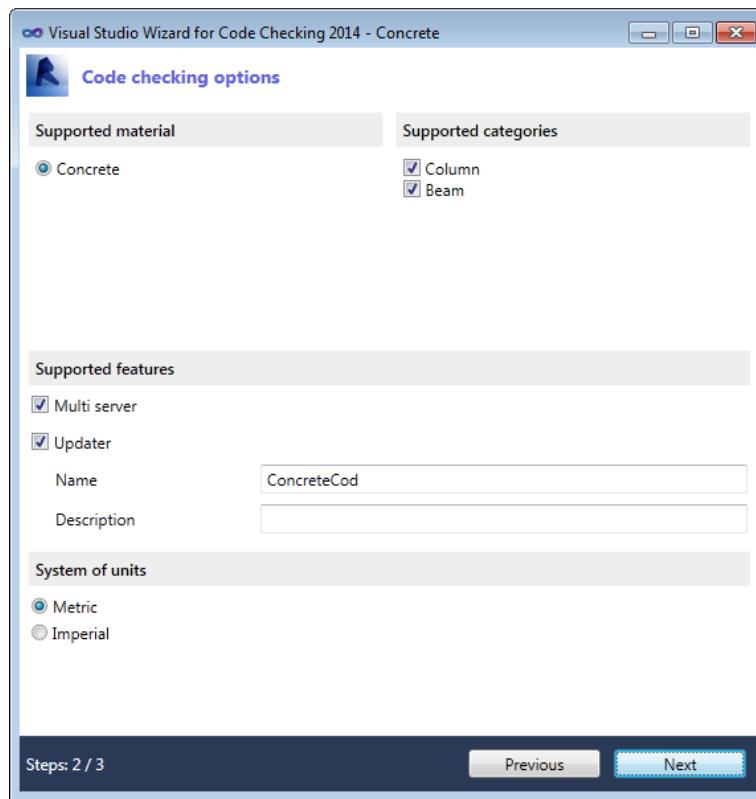
    public override string GetVendorId()
    {
        return "John Smith";
    }
```

Those strings are later used by the Code Checking Framework to expose to Revit user some application related information on the UI (both on dialogs and in the note).



4.2.2 Setting Code Checking options

Code Checking options determine the general scope of the application and are independent of the domain. So they apply to Concrete Calculations as much as to other calculation types.



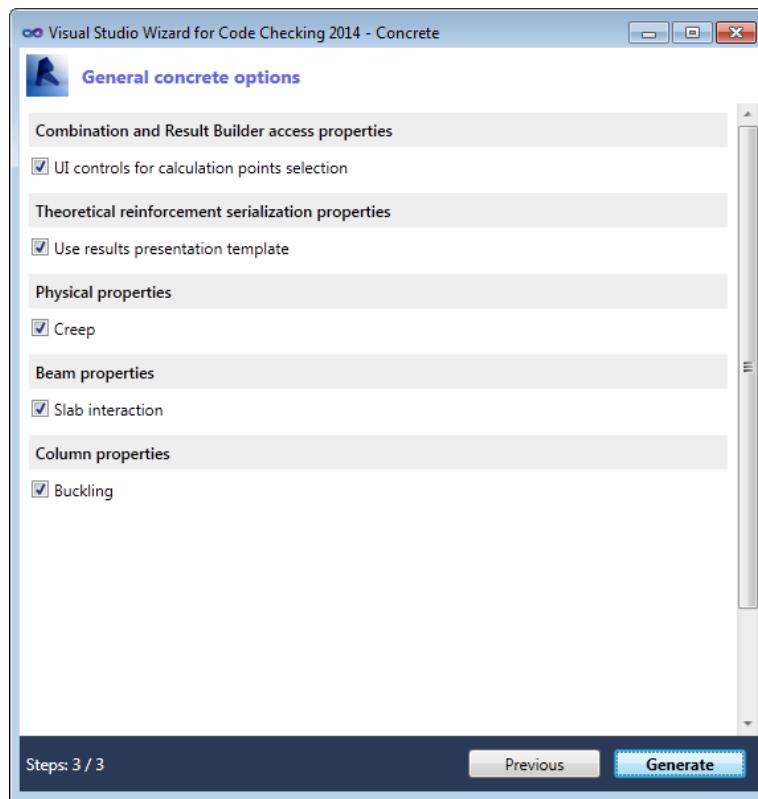
Following options are available:

- Supported material – filters structural material supported by the application. For the concrete template, only concrete is available.
- Supported categories – filters structural analytical model element that can be calculated by the Code Checking application. For this template, only beam and column are supported.
- Multi server - Let developer decides if the MultiServer option should be used or not
- Updater – Let developer decides if an entry point for a user defined updater should be created.
- System of units – sets units system (Metric/Imperial) for internal component calculations and for storing data using Extensible Storage.

4.2.3 Setting General concrete options

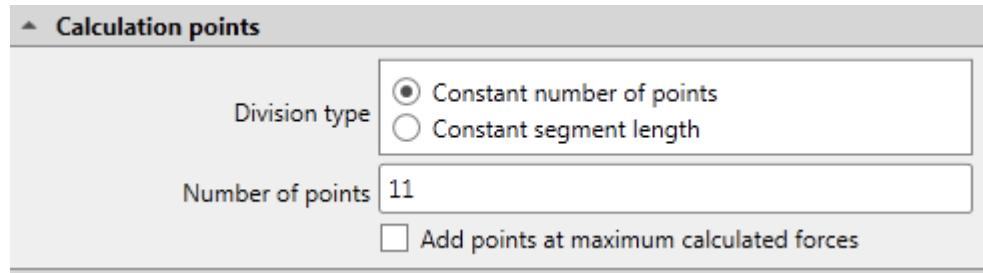
General concrete options enable the user to include in their project different functionalities that are available in the template.

These options are strictly related to Concrete design.



- UI controls for calculation points selection

The calculation points control is a UI component integrating few controls. The intention to this group of control is to facilitate the selection of sections in which calculations will be carried out for linear elements.



Both layout and logic in the code behind are simple and therefore intuitive.

There is arbitrary number of sections for each element. Sections can be distributed equally along element or the element can be divided into segments of constant length – therefore a set of two radio buttons. Below this radio buttons group, the number of divisions could be set by the Revit user using the edit box.

An additional option is exposed to Revit users to let them decide if for elements to calculate, the list of sections should be extended by those where forces are extreme.

Switching this option results in including specialized UI classes in the project and integrating them into the main calculation loop and Calculation Parameters Label (these topics will be described later on this document).

- Use results presentation template

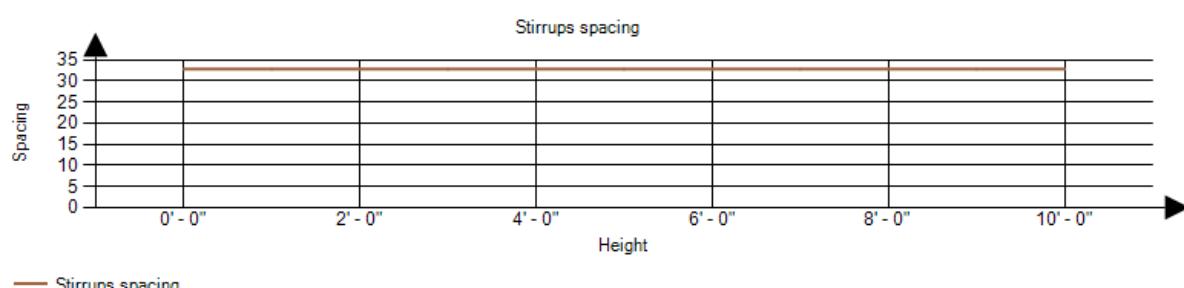
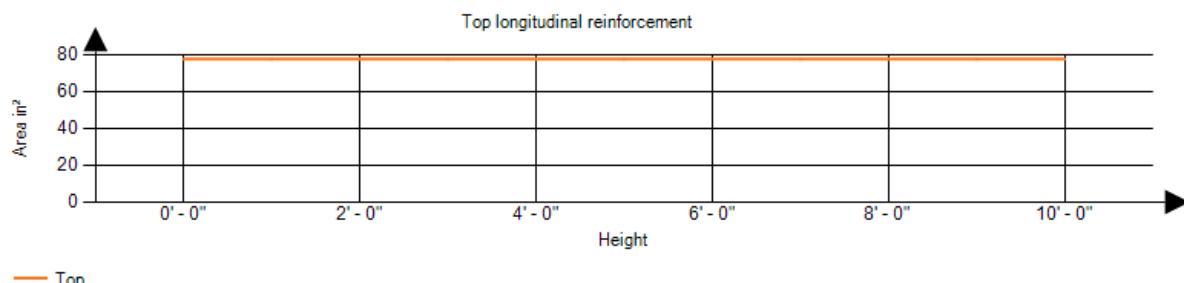
By setting this option developers may include in his project part of code that helps him to presents calculation results on the report. There are three sections in results: Summary, result tables and charts.

Summary

Mean longitudinal reinforcement density: 77.50 in²
 Mean transverse reinforcement density: 0.00 in²/ft
 Maximum longitudinal reinforcement: 77.50 in²
 Minimum longitudinal reinforcement: 77.50 in²
 Maximum stirrups spacing: 32' - 9 11/16"
 Minimum stirrups spacing: 32' - 9 11/16"

Reinforcement

Distance	Top	Stirrups spacing
0' - 0"	77.50 in ²	32' - 9 11/16"
1' - 0"	77.50 in ²	32' - 9 11/16"
2' - 0"	77.50 in ²	32' - 9 11/16"
3' - 0"	77.50 in ²	32' - 9 11/16"
4' - 0"	77.50 in ²	32' - 9 11/16"
5' - 0"	77.50 in ²	32' - 9 11/16"
6' - 0"	77.50 in ²	32' - 9 11/16"
7' - 0"	77.50 in ²	32' - 9 11/16"
8' - 0"	77.50 in ²	32' - 9 11/16"
9' - 0"	77.50 in ²	32' - 9 11/16"
10' - 0"	77.50 in ²	32' - 9 11/16"



If this option is not set, no calculation results will be presented in the note and developer will need to implement this part by hands (as for an application based on the Code Checking template).

- Creep

Creep option determines what creep value for concrete will be used in calculations.
 If the option is not set, the default value of 2.0 will be taken and used in the code.

Otherwise a special control will be added to the Element label thus giving the Revit user the possibility to set a value of their choice (Element labels such as Beam Label or Concrete Label are discussed later in this document).



Creep coefficient 2

- Slab interaction

Slab interaction option is available only for beam elements.

It gives the developer the choice of including support for beam-slab interaction in their code.

If the option is checked a couple of toggle buttons for switching off and on analysis of beam-slab interaction will be added to the Beam Label.



Slab-beam interaction

Additionally some extra code will be added to the main calculation loop. This code, if Slab-beam interaction button is press, will analyze Slab-beam interaction during calculation. Analysis will be based on actual connection between beam and slab connection from Revit model.

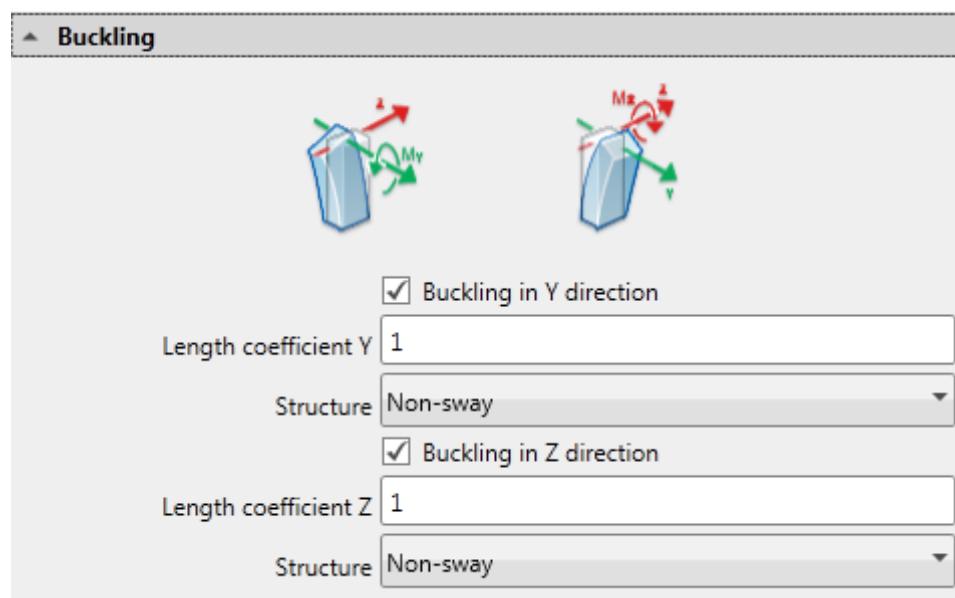
If this option is not checked, no Slab-beam interaction will be taken into account.

- Buckling

Buckling option is available only for column elements.

If checked it adds a set of controls for buckling parameters to the Column label.

Then Revit user can use them to parameterize buckling effects during calculations.



If this option is switched off, no buckling effects will be taken into account while calculating columns.

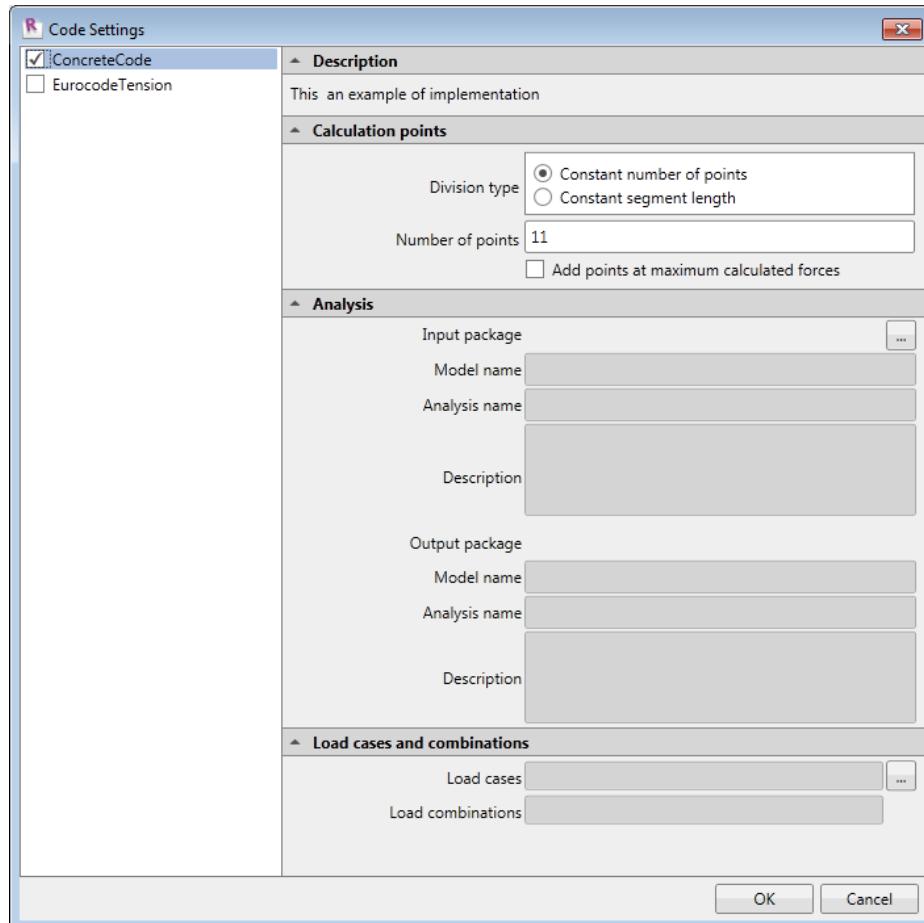
4.3 Default Data and controls

Apart from options described above, there are some other data and associated UI controls that are added to a project generated using the template.

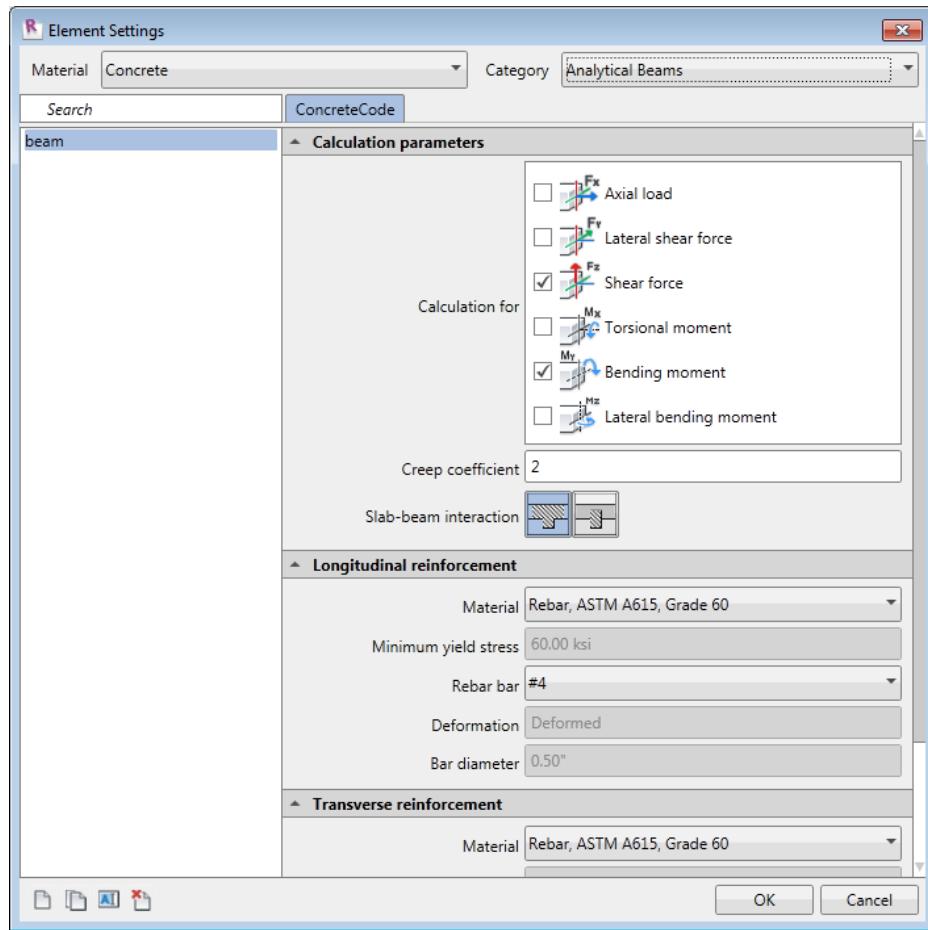
Following two sections contain a short description of them.

4.3.1 Code Settings

Code Settings dialog is common for all analytical elements and Code Checking application. Using the concrete template, Developer has control only on the “Calculation points” option. The support of the structural analysis results packages and load cases/combinations selection have been added automatically on the code settings layout.



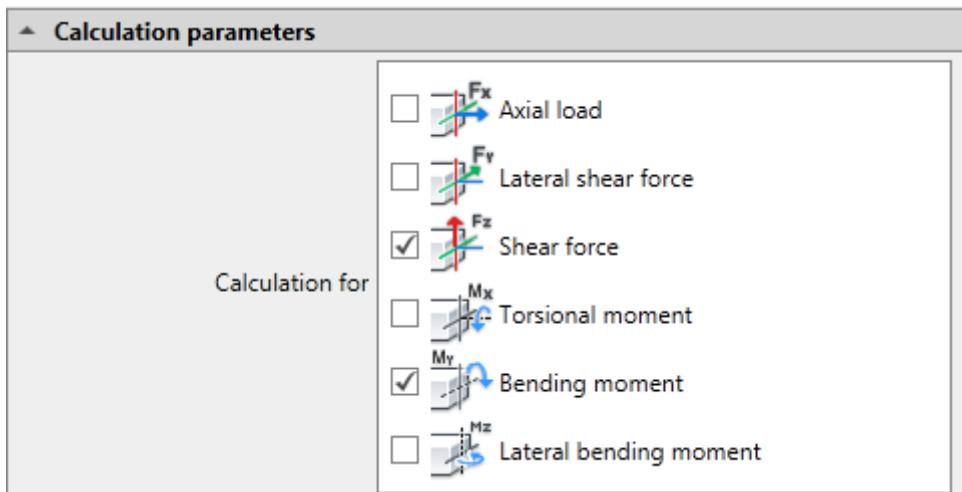
4.3.2 Element Settings



Element Settings are common for both columns and beams, however some controls are displayed differently depending on the type of element.

Element settings layout contains following sections:

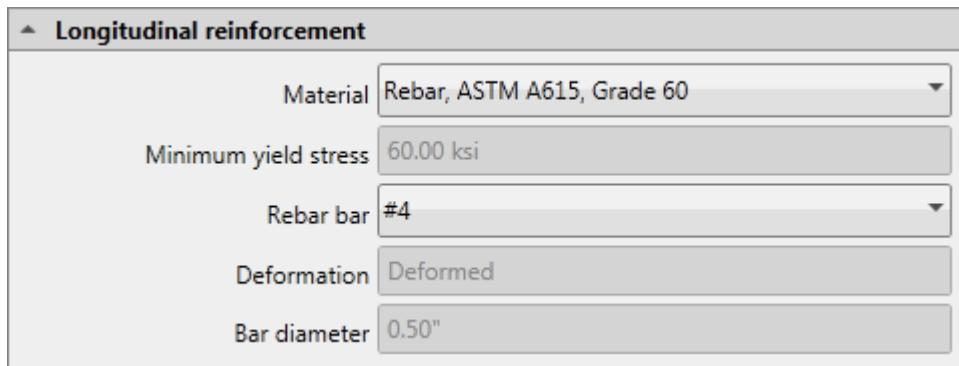
- Calculation parameters



The first element in this section is a set of checkboxes to let user select forces that will be taken into account during calculations.

The two remaining can be controlled by wizard and were described earlier in this document (creep and slab interaction).

- Longitudinal reinforcement

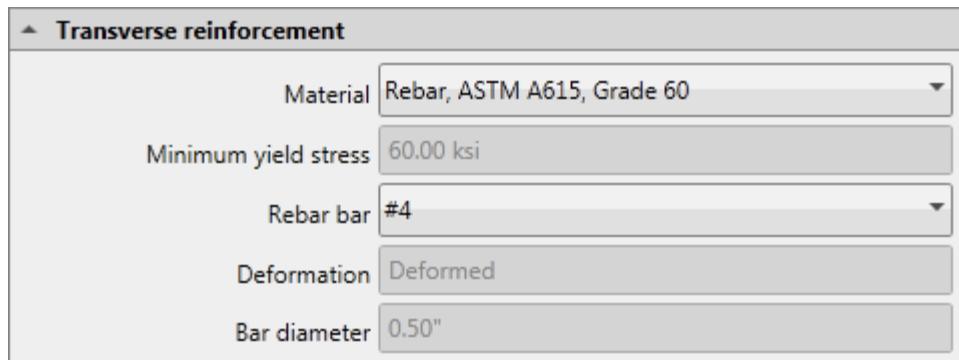


Longitudinal reinforcement is a reinforcement steel selection tool. From the five controls, only two are enabled for edition:

- Material ComboBox – displays all steel types that are available in Revit project and enables a choice of one element from the list. Available means assigned to rebar present on the Revit project.
- Rebar bar ComboBox – displays all rebar types available for selected material and enables a choice of one element from the list

The remaining three edit boxes are used only for presentation purposes only and values exposed are coming from parameters of selected elements.

- Transverse reinforcement



Transverse reinforcement is similar to the longitudinal reinforcement controls.

5 CONCRETE PROJECT - MAIN CALCULATION LOOP

As described in the previous section, a template could be used to generate a C# project dedicated to required reinforcement for concrete beams and columns.

Inside this project, several C# files have been created and the goal of this section is to explain the data flow and main part of the code exposed by this specific template.

This section is dedicated to part of code which implements the main calculation loop. Other parts of code are coming from the base Code Checking template or are related to the management of data specific to options chosen using the wizard.

Notes:

- It is assumed that Extensible Storage Framework and Code Checking concept are understood
- Following part could be read in parallel with the step by step document describing the implementation of a concrete code example.

The main loop provides a general calculation algorithm for beam and column elements and could be extended to other types of elements if needed.

This includes:

- Engine – the general algorithm
- IEngineData – the interface for general algorithm
- Element objects and calculation objects – implementations of base classes and interfaces which are operated by Engine.

5.1 Engine

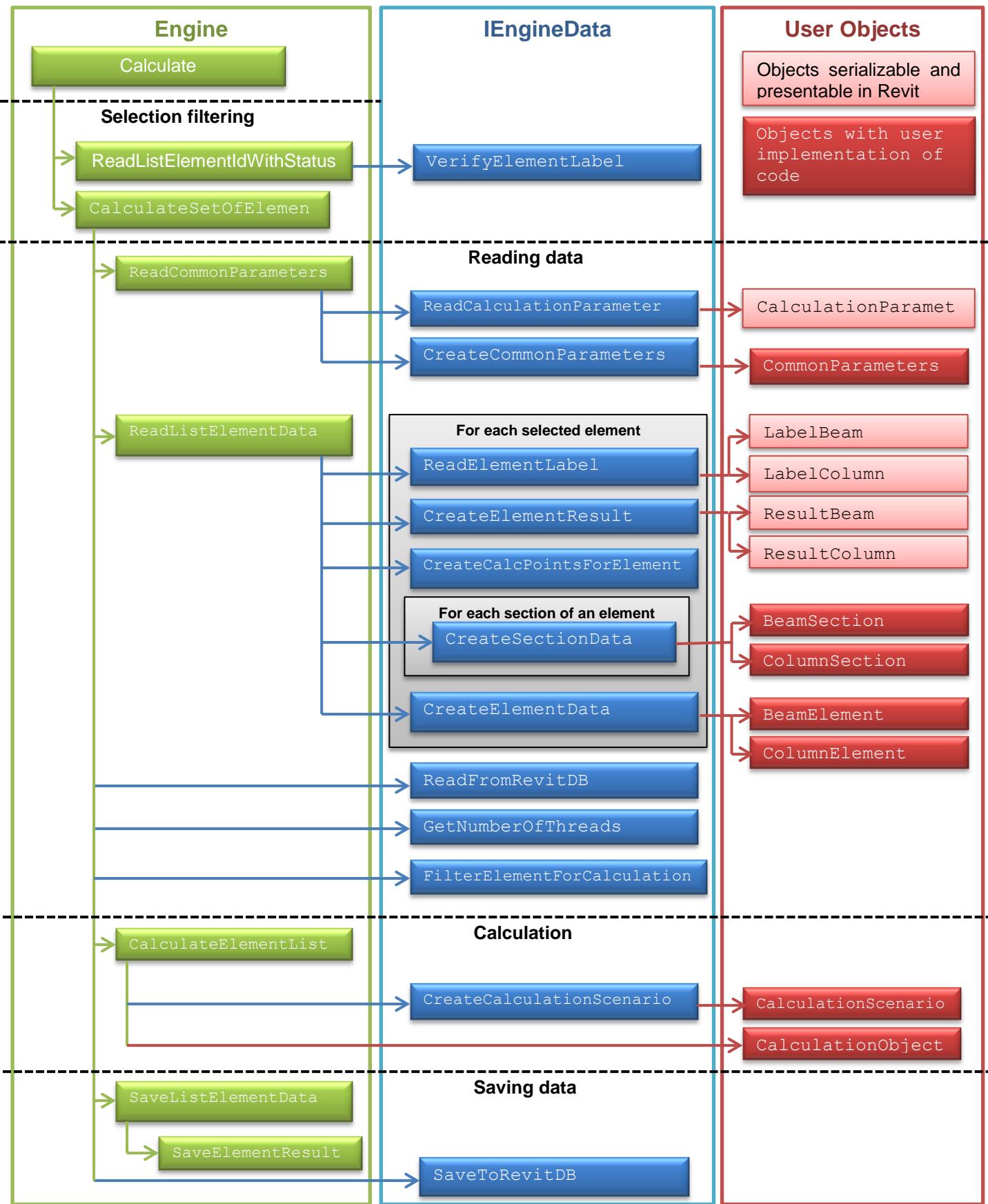
Engine class represents the main calculation loop. Engine is divided into three separated parts presented on Figure 1.

The first part is an internal algorithm, which need no changes by developer. On the diagram presented below this part is represented by a green chart. This part should be “code” agnostic and fitting majority of use cases.

The second part is an IEngineData - interface of main loop. IEngineData need to be implemented by developer and could be different according to codes. Interface of IEngineData is represented by a blue chart.

The third part is presented by red chart and it represents developer objects. Objects like labels, results, calculation parameters are provided by Code Checking Framework in Server and they are serializable and presentable in Revit (light red boxes). The rest of elements are used to implement specific code (red boxes).

Figure 1



5.1.1 Running main loop

To run calculations user need only create instance of Engine object, initialize Engine with an implementation of `IEngineData` and run `Engine.Calculate()` method.

The entry point for running the main loop is the method `Verify` from the Server.

Code region

```
public override void Verify(Autodesk.Revit.DB.CodeChecking.ServiceData data)
{
    Main.Calculation.EngineData enginData = new Main.Calculation.EngineData();
    Engine.Engine engine = new Engine.Engine(enginData);
    engine.Calculate(this, data);
}
```

5.1.2 Internal algorithm of main loop

The order of each method presented in Diagram 1 represents sequence of calling main loop methods.

Methods	Description
<code>public Engine(IEngineData engineData)</code>	Initializes a new instance of the <code>Engine</code> class.
<code>public void Calculate(Server server, ServiceData data)</code>	Runs calculation for elements served in user implementation of <code>IEngineData</code> .
<code>void CalculateSetOfElements(List<Tuple<ElementId, ResultStatus>> listElementStatus, Server server, ServiceData data)</code>	Reads from Revit information about selected elements, next Calculates the list of elements, at the end saves calculation results in Revit Data Base.
<code>void CalculateElementList(List<ObjectDataBase> listElementData, CommonParametersBase parameters, int maxNumberOfThreads)</code>	Calculates list of elements according to calculation scenario returned by <code>CreateCalculationScenario</code> . If the number of threads is greater than one than parallel calculation will be run.
<code>CommonParametersBase ReadCommonParameters(List<Tuple<ElementId, ResultStatus>> listElementStatus, Server server, ServiceData data)</code>	Read from Revit parameters common for all selected elements and stores them in <code>CommonParametersBase</code> .
<code>List<Tuple<ElementId, ResultStatus>> ReadListElementIdWithStatus(Server.Server server, ServiceData data)</code>	Reads from Revit selected elements ids, creates result statuses for each element and collect them in the list but only that elements which has appropriate label, material and category.
<code>List<ObjectDataBase> ReadListElementData(ServiceData data, List<Tuple<ElementId, ResultStatus>> listElementStatus, CommonParametersBase parameters)</code>	Reads from Revit information about selected elements and store it in the list with elements data.
<code>void SaveListElementData(ServiceData data, List<ObjectDataBase> listElementData)</code>	Saves in Revit data base information collected in the list elements data.
<code>void SaveElementResult(ElementId elementId, SchemaClass resultSchema, ResultStatus status, ServiceData data)</code>	Saves in Revit the result object of an element.

5.2 Interface IEngineData

Interface `IEngineData` is required by `Engine` constructor. Developer should implement all methods to provide general functionality for his Code Checking application.

The algorithm of main loop operates with a set of base classes and interfaces. Developer implementation of `IEngineData` gives possibility to make all main loop operations on own objects in own manner.

5.2.1 GetInputDataUnitSystem

`GetInputDataUnitSystem` method gives to developer possibility to set type of internal unit system. There are two possibilities: `DisplayUnit.METRIC` and `DisplayUnit.IMPERIAL`. Developer could decide which unit system will be used by additional external components and internally inside his Code Checking application. Unit system should be chosen at the beginning of creating a new Code Checking project.

Code region

```
public Autodesk.Revit.DB.DisplayUnit GetInputDataUnitSystem ()
{
    DisplayUnit displayUnit = DisplayUnit.IMPERIAL;
    if (Server.Server.UnitSystem == Autodesk.Revit.DB.ResultsBuilder.UnitsSystem.Metric)
        displayUnit = DisplayUnit.METRIC;
    return displayUnit;
}
```

5.2.2 GetNumberOfThreads

`GetNumberOfThreads` method gives a developer possibility to change a mode of calculation. If a developer wants to use the full power of a machine, this method should return number of available threads or processors. But during debugging process it is better to temporally change returned value onto 1.

Code region

```
public int GetNumberOfThreads(Autodesk.Revit.DB.CodeChecking.ServiceData data)
{
    return Environment.ProcessorCount;
}
```

5.2.3 CreateCalculationScenario

In `CreateCalculationScenario` method should be created new instance of user's object which implements `ICalculationScenario` interface. In an instance of `ICalculationScenario` developers can describe own calculation algorithm.

Code region

```
public ICalculationScenario CreateCalculationScenario()
{
    return new CalculationScenario();
}
```

5.2.4 CreateCommonParameters

In `CreateCommonParameters` method should be created new instance of user's object which is derived from class `CommonParametersBase`. Class `CommonParametersBase` provides few predefined parameters which are needed by the main loop. But in user's object, derived from `CommonParametersBase`, could be added additional parameters which are common for all calculated elements.

Code region

```
public CommonParametersBase CreateCommonParameters(ServiceData data, CommonParametersBase
parameters)
{
    CommonParameters commonParameters = new CommonParameters(data, parameters);

    return commonParameters;
}
```

5.2.5 CreateCalcPointsForElement

Method `CreateCalcPointsForElement` should return a list of calculation points for given element Id. `ElementId` is an id of Revit element. Using the `CodeChecking` Engineering component, `ResultCache` can return appropriate list of calculation points (`CalcPoint`) for a given `elementId` due to settings in `CalculationPointSelector` stored in `CalculationParameter` object.

User could return his own list of `CalcPoint` objects or use functionality of `ResultCache` component. Each `CalcPoint` could be type of `CalcPointLinear` for linear elements or could be type of `CalcPointSurface` for surface elements.

Each `CalcPoint` object should describe position of point on calculated element. Number of `CalcPoint` returned by this method determines how many object describing an element's sections will be created and calculated.

Code region

```
public List<CalcPoint> CreateCalcPointsForElement(ServiceData data, CommonParametersBase
parameters, ElementId elementId)
{
    List<CalcPoint> calculationPoints = new List<CalcPoint>();
    CommonParameters commonParameters = parameters as CommonParameters;
    if (commonParameters != null)
        calculationPoints = commonParameters.ResultCache.GetCalculationPoints(elementId);
    return calculationPoints;
}
```

5.2.6 CreateSectionData

Method `CreateSectionData` returns type of user section object of given element's category and material. Each user section object should be derived from `SectionDataBase` class and should be initialized with given as parameter `SectionDataBase` object.

In user section objects (`ColumnSection`, `BeamSection`) developer can add own parameters which describe a section and which are needed for calculation.

Main loop gives user possibility to create different type of section objects for one type of element's category and material, than information which type of section return should be gets from parameters stored in `sectionDataBase.CalcPoint`.

Code region

```
public SectionDataBase CreateSectionData(SectionDataBase sectionDataBase)
{
    switch (sectionDataBase.Material)
    {
        case StructuralAssetClass.Concrete:
            switch (sectionDataBase.Category)
            {
                default:
                    break;
                case Autodesk.Revit.DB.BuiltInCategory.OST_ColumnAnalytical:
                    return new ColumnSection(sectionDataBase);
                case Autodesk.Revit.DB.BuiltInCategory.OST_BeamAnalytical:
                    return new BeamSection(sectionDataBase);
            }
            break;
        case StructuralAssetClass.Metal:
            break;
    }

    return sectionDataBase;
}
```

5.2.7 CreateElementData

Method `CreateElementData` returns type of user element object of given element's category and material.

Each user element object should be derived from `ElementDataBase` class and should be initialized with given as parameter `elementDataBase` object.

In user element objects (`ColumnElement`, `BeamElement`) developer can add own parameters which describe an element and which are needed for calculation.

Main loop gives user possibility to create different type of element objects for one type of element's category and material, than information which type of element return should be gets from parameters stored in `elementDataBase` object, e.g. from a `Label` object.

Code region

```
public ElementDataBase CreateElementData(ElementDataBase elementDataBase)
{
    switch (elementDataBase.Material)
    {
        case StructuralAssetClass.Concrete:
            switch (elementDataBase.Category)
            {
                default:
                    break;
                case Autodesk.Revit.DB.BuiltInCategory.OST_ColumnAnalytical:
                    return new ColumnElement(elementDataBase);
                case Autodesk.Revit.DB.BuiltInCategory.OST_BeamAnalytical:
                    return new BeamElement(elementDataBase);
            }
            break;
        case StructuralAssetClass.Metal:
            break;
    }
    return elementDataBase;
}
```

5.2.8 CreateElementResult

Method `CreateElementResult` creates and returns new instance of user `Result` object for given element's category and material.

Rules of creating and editing `Result` objects derived from `SchemaClass` are described previously on this manual.

Code region

```
public SchemaClass CreateElementResult(BuiltInCategory category,
                                       StructuralAssetClass material)
{
    switch (material)
    {
        case StructuralAssetClass.Concrete:
            switch (category)
            {
                default:
                    break;
                case Autodesk.Revit.DB.BuiltInCategory.OST_BeamAnalytical:
                    return new ResultBeam();
                case Autodesk.Revit.DB.BuiltInCategory.OST_ColumnAnalytical:
                    return new ResultColumn();
            }
            break;
        case StructuralAssetClass.Metal:
            break;
    }
    return null;
}
```

5.2.9 ReadCalculationParameter

Method `ReadCalculationParameter` should returns user `CalculationParameter` object copied from Revit database. Rules of creating and editing `CalculationParameter` object derived from `SchemaClass`.

Code region

```
public SchemaClass ReadCalculationParameter(Autodesk.Revit.DB.CodeChecking.ServiceData data)
{
    StorageService service = CodeChecking.Storage.StorageService.GetService();
    Storage.StorageDocument storageDocument = service.GetStorageDocument(data.Document);
    CalculationParameter calculationParameter = storageDocument.CalculationParamsManager
        .CalculationParams.GetEntity<CalculationParameter>(data.Document);
    return calculationParameter;
}
```

5.2.10 ReadElementLabel

Method `ReadElementLabel` should return user `Label` object for given element's category and material. This object is copied from Revit database. Rules of creating and editing `Label` object derived from `SchemaClass`.

Code region

```
public SchemaClass ReadElementLabel(BuiltInCategory category,
    StructuralAssetClass material, CodeChecking.Storage.Label label, ServiceData data)
{
    if (label != null)
    {
        switch (material)
        {
            case StructuralAssetClass.Concrete:
                switch (category)
                {
                    default:
                        break;
                    case BuiltInCategory.OST_ColumnAnalytical:
                        return label.GetEntity<LabelColumn>(data.Document);
                    case BuiltInCategory.OST_BeamAnalytical:
                        return label.GetEntity<LabelBeam>(data.Document);
                }
                break;
            case StructuralAssetClass.Metal:
                break;
        }
    }
}
```

5.2.11 VerifyElementLabel

Method `VerifyElementLabel` verifies parameters stored in user's element `Label`. User could add errors to a status of element when any parameter from a label doesn't fulfill requirements.

Code region

```
public void VerifyElementLabel(BuiltInCategory category,
                               StructuralAssetClass material, SchemaClass label,
                               ref Autodesk.Revit.DB.CodeChecking.Storage.ResultStatus status)
{
    if (label != null)
    {
        switch (material)
        {
            case StructuralAssetClass.Concrete:
                switch (category)
                {
                    default:
                        break;
                    case Autodesk.Revit.DB.BuiltInCategory.OST_ColumnAnalytical:
                    {
                        LabelColumn labelCol = label as LabelColumn;
                        if (labelCol != null)
                        {
                            if (labelCol.EnabledInternalForces.Count == 0)
                                status.AddError("ErrNoChosenInternalForces");
                        }
                    }
                    break;
                    case Autodesk.Revit.DB.BuiltInCategory.OST_BeamAnalytical:
                    {
                        LabelBeam labelBm = label as LabelBeam;
                        if (labelBm != null)
                        {
                            if (labelBm.EnabledInternalForces.Count == 0)
                                status.AddError("ErrNoChosenInternalForces");
                        }
                    }
                    break;
                }
            break;
        }
    }
}
```

5.2.12 FilterElementForCalculation

Method `FilterElementForCalculation` should returns list of elements which user wants to calculate. User cans filter input `listElementData` and exclude from the list elements with errors.

Code region

```
public List<ObjectDataBase> FilterElementForCalculation(List<ObjectDataBase>
                                                       listElementData)
{
    List<ObjectDataBase> listElementFiltered = new List<ObjectDataBase>();
    foreach (ObjectDataBase obj in listElementData)
    {
        ElementDataBase elem = obj as ElementDataBase;
        if (elem != null)
        {
            if (!elem.Status.IsError())
                listElementFiltered.Add(elem);
        }
    }
    return listElementFiltered;
}
```

5.2.13 ReadFromRevitDB

Method `ReadFromRevitDB` gives possibility to read data from Revit project and puts them into user element's objects and into user section's objects and into user common parameters.

Code region

```
public void ReadFromRevitDB(List<ObjectDataBase> listElementData, CommonParametersBase
parameters, ServiceData data)
{
    // read additional information from Revit
}
```

5.2.14 SaveToRevitDB

Method `SaveToRevitDB` gives possibility to write data to Revit project. The method has access to the list of user's element objects and user common parameters.

Code region

```
public void SaveToRevitDB(List<ObjectDataBase> listElementData,
                         CommonParametersBase parameters, ServiceData data)
{
    // save additional information to Revit
}
```

5.3 Implementation of code

Whole implementation of code could be set in objects which represent elements, sections and calculation.

Below are five steps to implement a code for beam and column elements:

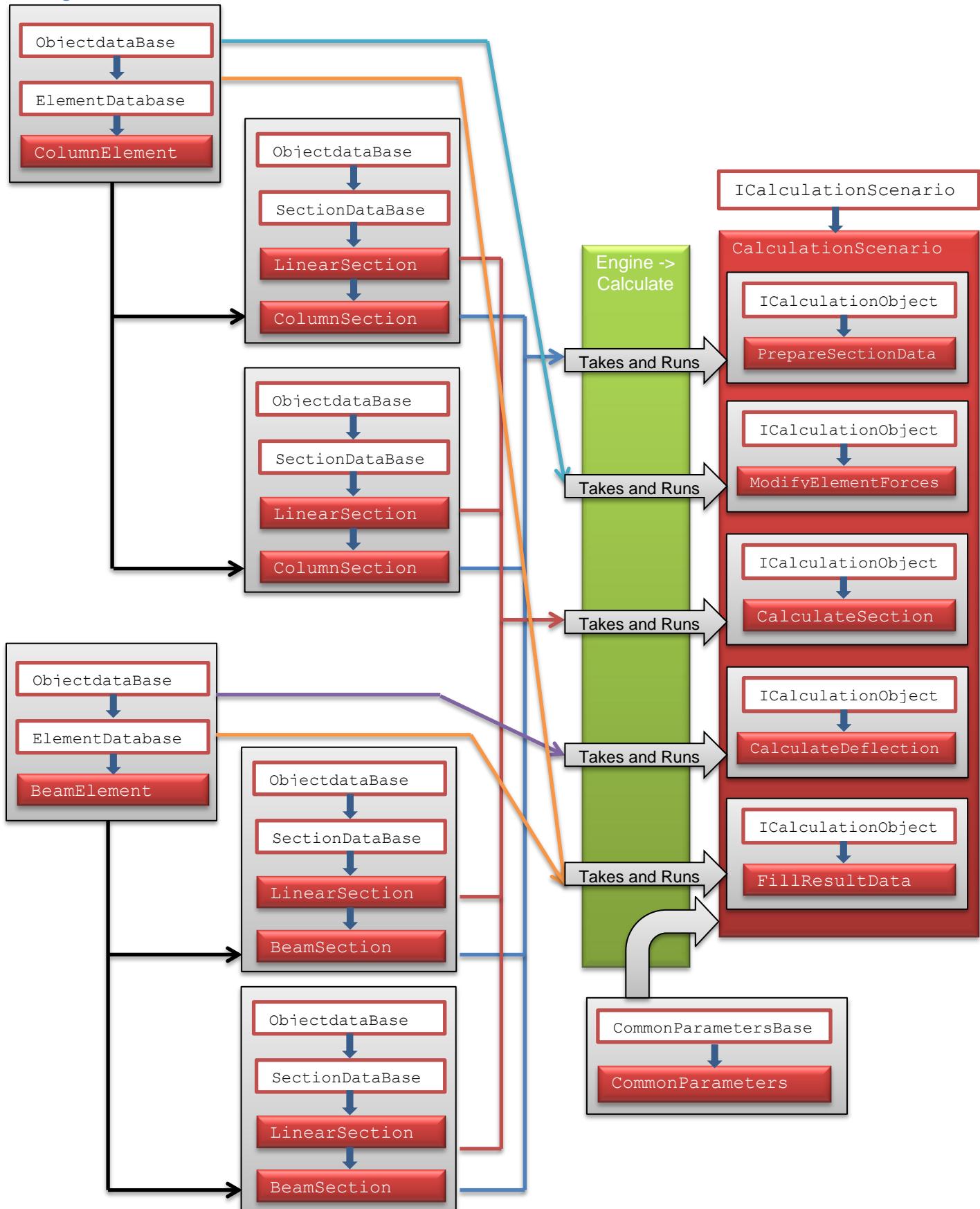
1. Implement an element object `BeamElement` for beams and an object `ColumnElement` for columns, both derived from `ElementDataBase`.
2. Implement a section object `BeamSection` for beams and an object `ColumnElement` for columns, both derived from `SectionDataBase`.
3. Implement a `CommonParameters` object derived from a class `CommonParametersBase` with parameters which are common for all elements.
4. Implement any number of calculation objects derived from `ICalculationObject`, each should represent different kind of calculations e.g. `PrepareSectionData`, `CalculateDeflection`, each the objects could be purposed exclusively to calculate sections or purposed exclusively to calculate elements and to listed categories.
5. Implement a calculation scenario object derived from `ICalculationScenario`. In the object you have to create list of your calculation objects in appropriate order for the implemented code.

How calculation for a code works?

After implementation all these objects we can run calculation. Calculation method of the Engine takes consecutively calculation objects from the list created in `CalculationScenario`, next it tries type of current calculation object, if it is type of `CalculationObjectType.Element` the Engine runs calculation for all element objects, if it is type of `CalculationObjectType.Section`, Engine runs calculation for all section objects. Additionally for each calculation objects algorithm serve `CommonParameters` object.

In a **Error! Reference source not found.** is shown general calculation algorithm for any code.

Figure 2



5.3.1 Element objects

All objects which represent elements should be derived from `ElementDataBase` class.

Each element's object correct implemented in `EngineData` interface has following parameters;

- `SchemaClass Result`: element's result schema (presentable Revit element's results for Code Checking)
- `List<CalcPoint> calcPoints`: list of calculation points for an element
- `List<SectionDataBase> listSectData`: list of sections data of the element
- `ResultStatus Status`: Element's result status (calculation errors, warnings, ...)

Inherited parameters from `ObjectDataBase` class:

- `BuiltInCategory category`: Element's category (`OST_BeamAnalytical`, `OST_ColumnAnalytical`)
- `StructuralAssetClass material`: Element's material type from Revit (concrete, metal,...)
- `ElementId elemId`: Element's identificator from Revit
- `SchemaClass label`: Element's Label schema (presentable Revit element settings for Code Checking)

All these parameters are automatically initialized by `EngineData` interface. All additional developer parameters could be added to his specific element objects.

Below are 3 steps to implement object for beam and column elements:

1. Add new class derived from `ElementDataBase` class.

Code region

```
class BeamElement : ElementDataBase
{
    public BeamElement(ElementDataBase elementDataBase) : base(elementDataBase)
    {
    }
    // Here add your parameters.

}

class ColumnElement : ElementDataBase
{
    public ColumnElement(ElementDataBase elementDataBase) : base(elementDataBase)
    {
    }
    // Here add your parameters.

}
```

2. Add specific parameters to these objects which will be needed for calculation, storing results, or storing additional parameters read from Revit project.
3. Return these types of elements objects in a method `CreateElementData` (in `EngineData` interface)

5.3.2 Section objects

All objects which represent sections should be derived from `SectionDataBase` class.
 Each section's object correct implemented in `EngineData` interface has following parameters;
`CalcPoint calcPoint`: a calculation point linked to the section,
 And inherited parameters from `ObjectDataBase` class of the element to which the section belongs:

- `BuiltInCategory category`: Element's category (`OST_BeamAnalytical`, `OST_ColumnAnalytical`)
- `StructuralAssetClass material`: Element's material type from Revit (concrete, metal,...)
- `ElementId elemId`: Element's identifier from Revit
- `SchemaClass label`: Element's label schema (presentable Revit element settings for Code Checking)

All these parameters are automatically initialized by `EngineData` interface. All additional user's parameters could be added to his specific section objects.

Below are 3 steps to implement object for beam and column sections:

1. Add new class derived from `SectionDataBase` class.

Code region

```
class BeamSection : SectionDataBase
{
    public BeamSection(SectionDataBase sectionDataBase) : base(sectionDataBase)
    {
    }
    // Here add your parameters.

}

class ColumnSection : SectionDataBase
{
    public ColumnSection(SectionDataBase sectionDataBase) : base(sectionDataBase)
    {
    }
    // Here add your parameters.

}
```

2. Add specific parameters to these objects which will be needed for calculation, storing results or storing additional parameters read from Revit project.
3. Return these types of sections objects in a method `CreateSectionData` (in `EngineData` interface)

5.3.3 CommonParameters

Developer object which represents common parameters for all elements and that should be derived from a class `CommonParametersBase`.

`CommonParameters` object implementing `IEngineData` interface has following parameters:

- `Guid activePackageGuid`: Identifier of an active package with results
- `List<ElementId> listElemId`: List of elements identifiers
- `List<ElementId> listCombId`: List of combinations identifiers
- `SchemaClass calcParams`: Reference to calculation parameters (`CalculationParameter`)

Below are three steps to implement Common Parameters:

1. Add new class derived from `CommonParametersBase` class.

Code region

```
public class CommonParameters : CommonParametersBase
{
    public CommonParameters(ServiceData data, CommonParametersBase param) : base(param)
    {
    }

    // Here add your parameters.
}
```

2. Add specific parameters to this object which will be needed for calculation
3. Return this type of `CommonParameters` objects in a method `CreateCommonParameters` (in `EngineData` interface)

5.3.4 Calculation objects

All objects which represent code calculations should be derived from `ICalculationObject` class.

Below are the three steps to implement a calculation object:

1. Add new class derived from `ICalculationObject` class.

Code region

```
public class CalculateSection : ICalculationObject
{
    public bool Run(ObjectDataBase obj)
    {
        return true;
    }

    public CommonParametersBase Parameters { get; set; }
    public CalculationObjectType Type { get; set; }
    public IList< BuiltInCategory> Categories { get; set; }
    public ErrorResponse ErrorResponse { get; set; }
}

public class CalculateDeflection : ICalculationObject
{
    public bool Run(ObjectDataBase obj)
    {
        return true;
    }

    public CommonParametersBase Parameters { get; set; }
    public CalculationObjectType Type { get; set; }
    public IList<BuiltInCategory> Categories { get; set; }
    public ErrorResponse ErrorResponse { get; set; }
}
```

2. Add your specific calculation algorithm in a `Run` method. Return a value false if you want to inform calculation algorithm that your calculation failed.
3. Add all created calculation objects in appropriate order in a `CalculationScenario` object.

5.3.5 Calculation scenario

`CalculationScenario` object represents a calculation scenario and is derived from `ICalculationScenario`.

In this object developers have to create a list that contains their calculation objects in appropriate order.

Below are three steps to implement a calculation scenario:

1. Add new class derived from `ICalculationScenario` class.

Code region

```

public class CalculationScenario : ICalculationScenario
{
    public List<ICalculationObject> CalculationScenarioList()
    {
        List<ICalculationObject> scenario = new List<ICalculationObject>();

        {
            CalculateSection calcObj = new CalculateSection();
            calcObj.Type = CalculationObjectType.Section;
            calcObj.ErrorResponse = ErrorResponse.SkipOnError;
            calcObj.Categories = new List<BuiltInCategory>()
                { BuiltInCategory.OST_BeamAnalytical,
                  BuiltInCategory.OST_ColumnAnalytical };
            scenario.Add(calcObj);
        }
        {
            CalculateDeflection calcObj = new CalculateDeflection();
            calcObj.Type = CalculationObjectType.Element;
            calcObj.ErrorResponse = ErrorResponse.SkipOnError;
            calcObj.Categories = new List<BuiltInCategory>()
                { BuiltInCategory.OST_BeamAnalytical };
            scenario.Add(calcObj);
        }
        {
            FillResultData calcObj = new FillResultData();
            calcObj.Type = CalculationObjectType.Element;
            calcObj.ErrorResponse = ErrorResponse.RunOnError;
            calcObj.Categories = new List<BuiltInCategory>()
                { BuiltInCategory.OST_BeamAnalytical,
                  BuiltInCategory.OST_ColumnAnalytical };
            scenario.Add(calcObj);
        }
        return scenario;
    }
}

```

2. Create a list of calculation objects in a CalculationScenarioList method, which represents all steps of calculation.

For an each calculation object developer should define:

- A type of calculation (CalculationObjectType.Element for objects which implements calculation for elements or set a type CalculationObjectType.Section for objects which implements calculation for sections),
- A list of categories supported by the object (the Run method of an object is called only for elements or sections which have supported category),
- An error response (If ErrorResponse.SkipOnError is set, calculation algorithm skips next Run methods when any Run method returned before value false. If ErrorResponse.RunOnError is set, calculation algorithm always calls the Run method.)

3. Return this type of CalculationScenario object in a method CreateCalculationScenario (in Engine Data interface).

6 REINFORCEMENT CONCRETE COMPONENT

Component: RCUAPINet.dll

Namespace: Autodesk.CodeChecking.Concrete

As we saw previously, the Code Checking SDK exposes a set of feature to facilitate the creation of Concrete Design application in Revit.

To go even further, the component Autodesk.CodeChecking.Concrete provides common methods for verification of reinforced Concrete cross section. The component is based on mechanical behavior of concrete and reinforcement (steel) and geometry of the cross section to analyze.

This component delivers information about reinforced cross section in the failure state, stresses and forces according to set of strains and other information useful during the reinforcement concrete design.

Note: This component is available for x86 and x64 architecture. Code Checking Framework will load and resolve correct version in Revit environment.

6.1 Units

The component is totally independent to units, but all input data must be set in a consistent system of units. It can be used with metric units (SI), United States customary units or any other systems. Of course the units of outputs are dependent of input data units. The easiest and recommended way to use this component is to define all input data in base units e.g. [m], [N] or [in], [lb] and derived units e.g. [Pa = N / (m²)] or [psi = lb / (in²)]. In that case the output data will be in the base and derived units too. The list of units for input and output in that case is presented below:

- metric units (SI):

Unit	Input	Output
length, distance	[m]	[m]
area	[m ²]	[m ²]
first moment	-	[m ³]
moment of inertia	-	[m ⁴]
force	[N]	[N]
moment	[N · m]	[N · m]
stress	[N / (m ²)]	[N / (m ²)]
angle	[Radian]	[Radian]

- United States customary units:

Unit	Input	Output
length, distance	[in]	[in]
area	[in ²]	[in ²]
first moment	-	[in ³]
moment of inertia	-	[in ⁴]
force	[lb]	[lb]
moment	[lb · in]	[lb · in]
stress	[lb / (in ²)]	[lb / (in ²)]
angle	[Radian]	[Radian]

- Other user units system:

Units	Input	Output
length, distance	[length]	[length]
Area	[length ²]	[length ²]
first moment	-	[length ³]
moment of inertia	-	[length ⁴]
Force	[force]	[force]
moment	[force · length]	[force · length]
Stress	[force / (length ²)]	[force / (length ²)]
Angle	[Radian]	[Radian]

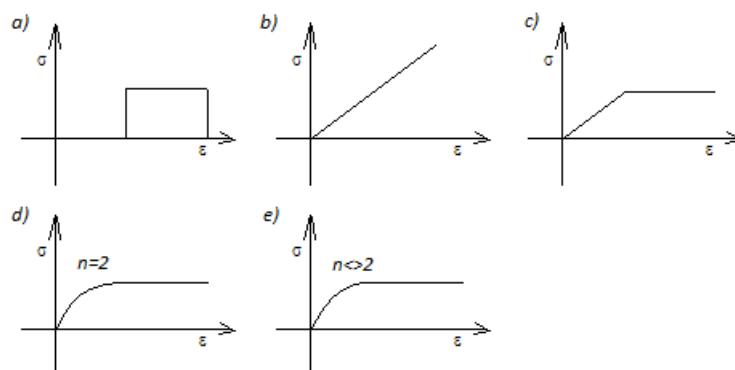
6.2 Concrete material

The concrete material is described by the following set of mechanical parameters:

- DesignStrength – strength of concrete at failure point
- StrainUltimateLimit – limit of compressive strain of concrete at failure point
- ModulusOfElasticity - modulus of elasticity (Young modulus)
- StressDiagramType - model of mechanical behavior
- Power - the value of power for power-rectangular model of stress-strain relation
- StrainRelationChangeover - value of strain where the type of stress-strain relationship is changing
- EffectiveHeightReductionFactor – reduction factor of the compression zone height

Mechanical behavior of concrete is described by the type of stress-strain relationship. The component supports 5 models of stress-strain relationship:

- a) Rectangular - simplified model,
- b) Linear – linear function,
- c) Bilinear - bilinear (linear-rectangular),
- d) ParabolicRectangular - parabolic-rectangular,
- e) PowerRectangular - power-rectangular,



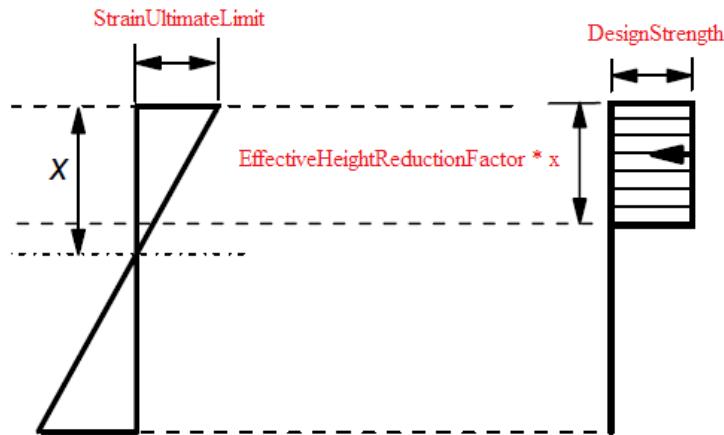
List of necessary properties is different for each stress-strain relationship. Not used properties are ignored. Below list of dependencies:

Properties	Diagram Type
------------	--------------

	Rectangular	Linear	Bilinear	ParabolicRectangular	PowerRectangular
DesignStrength	X	X	X	X	X
StrainUltimateLimit	X	X	X	X	X
ModulusOfElasticity	X	X	X	X	X
StrainRelationChangeover			X	X	X
Power					X
EffectiveHeightReductionFactor	X				

6.2.1 Rectangular model

This is a simplified model with uniform stress block. The maximum stress is limited by the DesignStrength. The maximum strain is limited by the StrainUltimateLimit. The height of the equivalent stress block is reduced by EffectiveHeightReductionFactor.



To define concrete according to this model the dedicated method `SetStrainStressModelRectangular` can be used:

Code region

```
public void CreateConcrete()
{
    // ACI 318-08 3000 psi concrete with rectangular stress distribution
    double betha1 = 0.85;
    double fc = 4000;
    double Ec = 57000 * Math.Sqrt(fc);
    Concrete concrete = new Concrete();
    concrete.SetStrainStressModelRectangular( 0.85*fc, 0.003, Ec, betha1);
}
```

Alternatively all parameters of concrete can be set one by one:

Code region

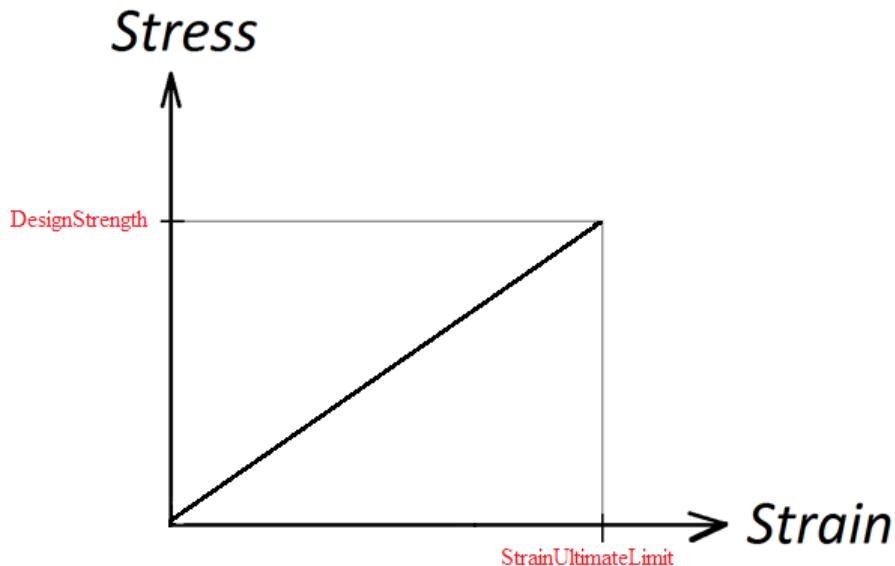
```
public void CreateConcrete()
{
    // ACI 318-08 3000 psi concrete with rectangular stress distribution
    double betha1 = 0.85;
    double fc = 4000;
    double Ec = 57000*Math.Sqrt(fc);
    Concrete concrete = new Concrete();
    concrete.StrainStressModel = StressDiagramType.Rectangular;
    concrete.DesignStrength =0.85*fc;
    concrete.StrainUltimateLimit = 0.003;
    concrete.EffectiveHeightReductionFactor = betha1;
    concrete.ModulusOfElasticity = Ec;
}
```

6.2.2 Linear model

This model is typically used for Serviceability Limit State (SLS) calculation or for Working Stress Method (WS) calculation. The compressive stress is proportional to the strain. This model is described by following equation:

$$\text{stress} = \text{strain} \cdot \text{DesignStrength} / \text{StrainUltimateLimit};$$

The maximum stress is limited by the DesignStrength. The maximum strain is limited by the StrainUltimateLimit.



To define concrete according to that model the dedicated method `SetStrainStressModelLinear` can be used:

Code region

```
public void CreateConcrete()
{
    // EN 1992-1-1:2004 C40/50 concrete for SLS long term effect calculation
    double fck = 40e6;
    double creepCoefficient = 2.1;
    double Ec = 35e9;
    double Eceff = Ec/(1.0 + creepCoefficient);
    double ecu = Eceff/fck;
    Concrete concrete = new Concrete();
    concrete.SetStrainStressModelLinear(fck, ecu, Eceff);
}
```

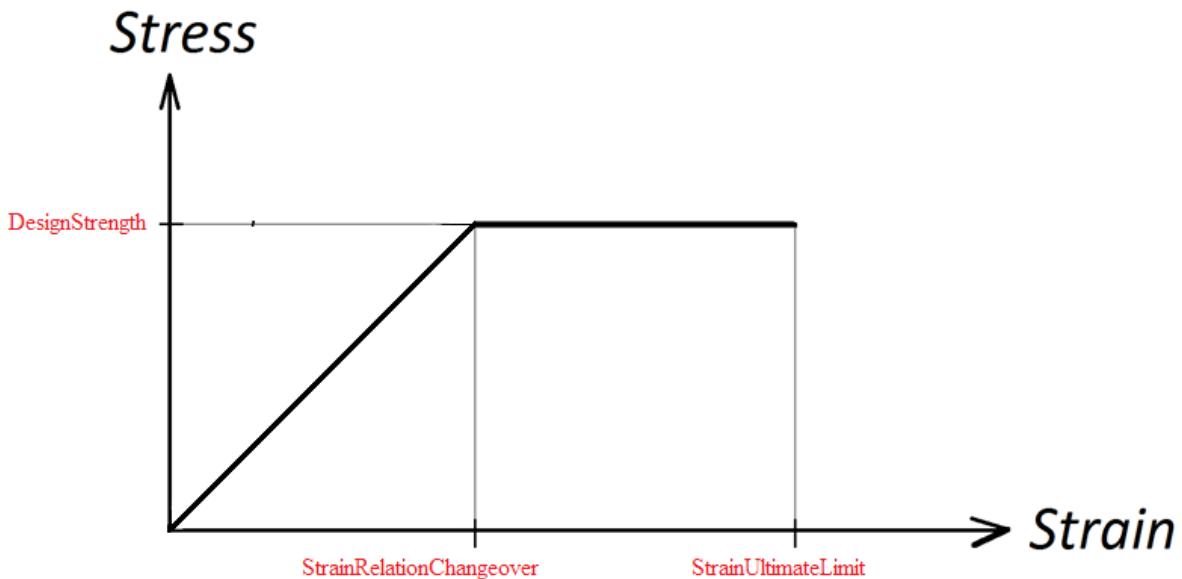
As for the rectangular model, all parameters to properly define the concrete can be set one by one.

6.2.3 Bilinear model

This model is used for Ultimate Limit State (ULS) calculation alternatively to parabolic-rectangular model. The compressive stress for strains between 0 and the `StrainRelationChangeover` is described by the linear function and between `StrainRelationChangeover` and `StrainUltimateLimit` by a constant function. This model is described by following equations:

```
when strain < StrainRelationChangeover then
    stress = strain · DesignStrength / StrainUltimateLimit;
when strain > StrainRelationChangeover then
    stress = DesignStrength
```

The maximum stress is limited by the `DesignStrength` and the maximum strain is limited by the `StrainUltimateLimit`.



To define concrete according to this model the dedicated method `SetStrainStressModelBiLinear` can be used:

Code region

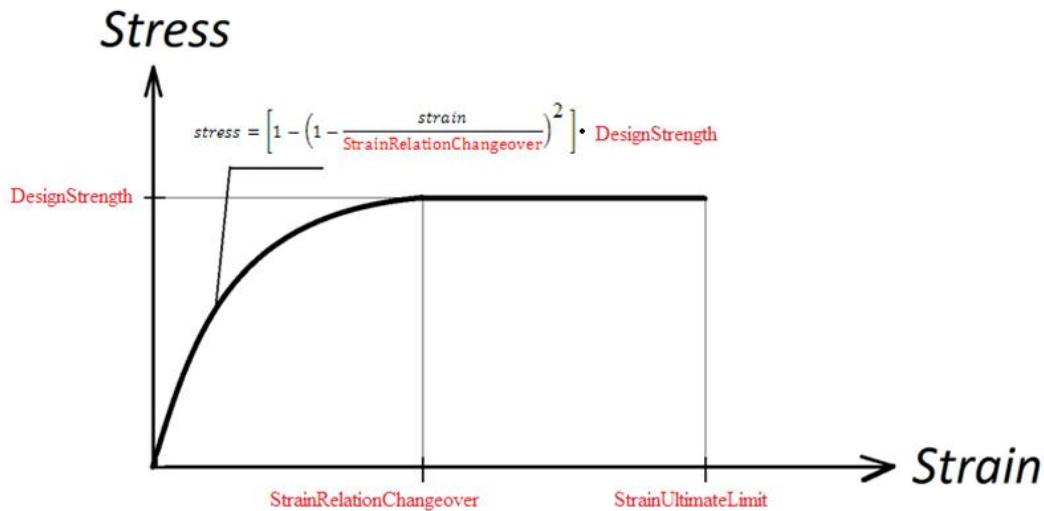
```
public void CreateConcrete()
{
    // EN-1992-1-1:2004 C30/37 bilinear model
    double fcd = 30e6/1.5;
    double Ecm = 32e9;
    double EpsCU3 = 1.75e-3;
    double EpsC3 = 3.5e-3;
    Concrete concrete = new Concrete();
    concrete.SetStrainStressModelBilinear(fcd, EpsCU3, Ecm, EpsC2);
}
```

6.2.4 Parabolic-Rectangular model

This is a frequently used model of concrete for Ultimate Limit State (ULS) calculation for normal strength concrete. The compressive stress for strains between 0 and `StrainRelationChangeover` is described by the parabolic function and between `StrainRelationChangeover` and `StrainUltimateLimit` by a constant function. This model is described by following equations:

```
when strain < StrainRelationChangeover
    stress = [1.0 - (1.0 - strain / StrainRelationChangeover)2] · DesignStrength
when strain > StrainRelationChangeover
    stress = DesignStrength
```

The maximum stress is limited by the `DesignStrength` and the maximum strain is limited by the `StrainUltimateLimit`.



To define concrete according to this model the dedicated method `SetStrainStressModelParabolicRectangular` can be used:

Code region

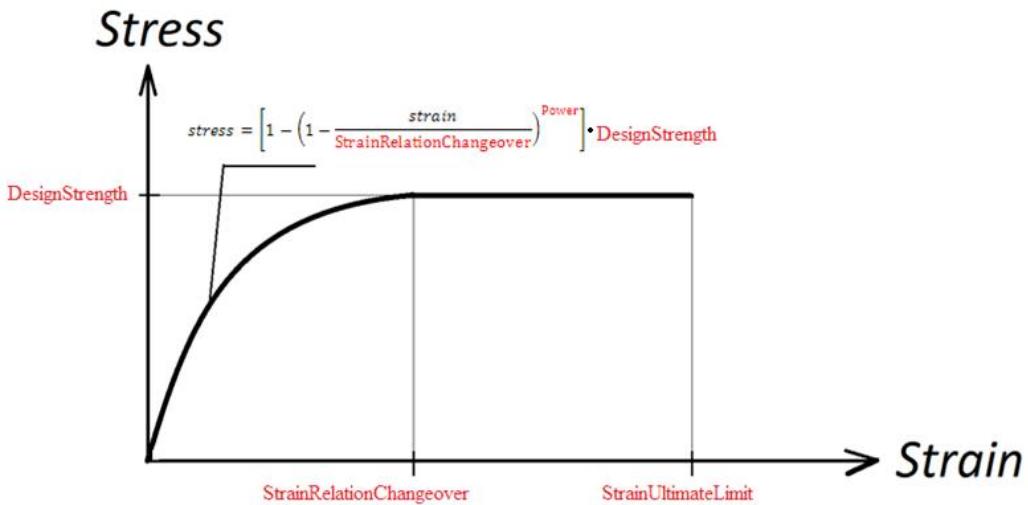
```
public void CreateConcrete()
{
    // IS:456-2000 M35 concrete parabolic-rectangular
    double fck = 35e6;
    double Ec = 5000*Math.Sqrt(fck*1e-6)*1e6;
    double ecu= 0.0035;
    Concrete concrete = new Concrete();
    concrete.SetStrainStressModelParabolicRectangular( 0.4416*fck,ecu, Ec, 0.002);
}
```

6.2.5 Power-Rectangular model

This model is typically used for Ultimate Limit State (ULS) calculation for high strength concrete. The compressive stress for strains between 0 and `StrainRelationChangeover` is described by the power function where power is less than 2.0 and between `StrainRelationChangeover` and `StrainUltimateLimit` by constant function. This model is described by equations:

```
when strain < StrainRelationChangeover
    stress = [1.0 - (1.0 - strain / StrainRelationChangeover)Power] · DesignStrength
when strain > StrainRelationChangeover
    stress = DesignStrength
```

The maximum stress is limited by the `DesignStrength` and the maximum strain is limited by the `StrainUltimateLimit`.



To define concrete according to this model the dedicated method SetStrainStressModelPowerRectangular can be used:

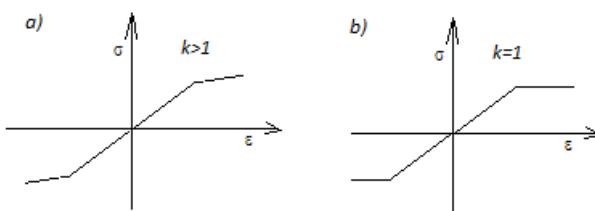
Code region

```
public void CreateConcrete()
{
    // EN-1992-1-1:2004 C80/95 power-rectangular
    double fcd = 90e6/1.5;
    double Ecm = 42e9;
    double n = 1.4;
    double EpsCU2 = 2.6e-3;
    double EpsC2 = 2.5e-3;
    Concrete concrete = new Concrete();
    concrete.SetStrainStressModelPowerRectangular(fcd, EpsCU2, Ecm, EpsC2, n);
}
```

6.3 Steel Material

The reinforcement steel is described by the following set of mechanical parameters:

- DesignStrength – strength of reinforcement steel at failure point (design yield strength)
- StrainUltimateLimit – limit of strain at failure point
- ModulusOfElasticity - modulus of elasticity (Young modulus)
 - a) HardeningFactor – defining model of mechanical behavior of steel: for k = 1.0 it is model with horizontal branch, for k > 1.0 with inclined branch. See picture below ((a) inclined branch model , (b) horizontal branch model)



To define steel with hardening (inclined branch) the dedicated method `SetModelWithHardening` can be used:

Code region

```
public void CreateSteel()
{
    // EN-1992-1-1:2004 steel 500A class A ANNEX C
    Steel steel = new Steel();
    double fyd = 500e6 / 1.15;
    double Es = 200e9;
    double hardening = 1.05;
    double strainLim = 0.025;
    steel.SetModelWithHardening(fyd, strainLim, Es, hardening);
}
```

To define steel without hardening (inclined branch) the dedicated method `SetModelIdealElastoPlastic` can be used:

Code region

```
public void CreateSteel()
{
    // ACI 318-08 steel A615 Grade 40
    Steel steel = new Steel();
    double fy = 40e3;
    double Es = 29e6;
    double strainLim = 0.02;
    steel.SetModelIdealElastoPlastic(fyd, strainLim, Es);
}
```

Alternatively all parameters to define the steel material model could be set by hand as following:

Code region

```
public void CreateSteel()
{
    // GB 50010 - 2002 HRB400
    Steel steel = new Steel();
    steel.DesignStrength = 360e6;      // fy' for fy = 400e6 Pa
    steel.ModulusOfElasticity = 200e9; // Es
    steel.StrainUltimateLimit = 0.01;
    steel.HardeningFactor = 1.0;      // without hardening
}
```

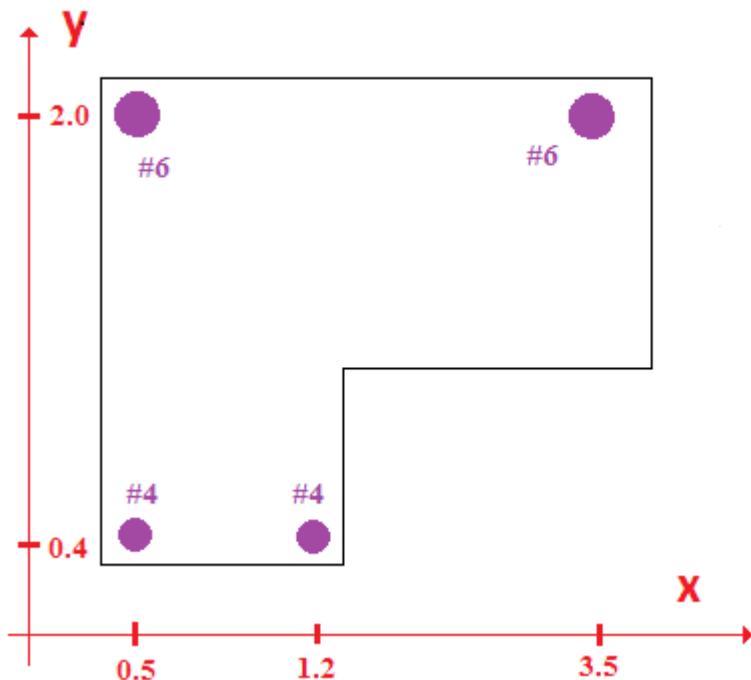
6.4 Rebar Definition

A single bar is described by position (x, y) and area as following:

Code region

```
public void CreateSingleRebar()
{
    // 0.01 diameter bar on the position x=y=0.05
    double diameter = 0.01;
    double area = diameter * diameter * Math.PI / 4.0;
    double x = 0.05;
    double y = 0.05;
    Rebar bar = new Rebar(x, y, area);
}
```

All rebars in a specific cross section are described by a list of `Rebar`. This list of `Rebar` is set by using `SetRebars` method:



Code region

```
public void CreateRebars(Geometry geometry)
{
    List<Rebar> rebars = new List<Rebar>();
    rebars.Add(new Rebar(0.5, 0.5, 0.2 )); // #4 area = 0.2 in2
    rebars.Add(new Rebar(1.2, 0.5, 0.2 ));
    rebars.Add(new Rebar(3.5, 2.0, 0.44 )); // #6 area = 0.44 in2
    rebars.Add(new Rebar(0.5, 2.0, 0.44 ));
    RCSolver solver = RCSolver.CreateNewSolver(geometry);
    solver.SetRebars(rebars);
}
```

6.5 Geometry

The concrete cross-section is described by a polyline. It is a list of points defined point by point. Component doesn't support arcs and other curves. Component does not support internal holes as well. Typical geometry (rectangular, T-section and circle) can be defined by dedicated constructors:

Code region

```
public void CreateTypicalGeometrys()
{
    //rectangular section
    double b = 0.2;
    double h = 0.5;
    Geometry geometryRectangular = new Geometry(h,b);

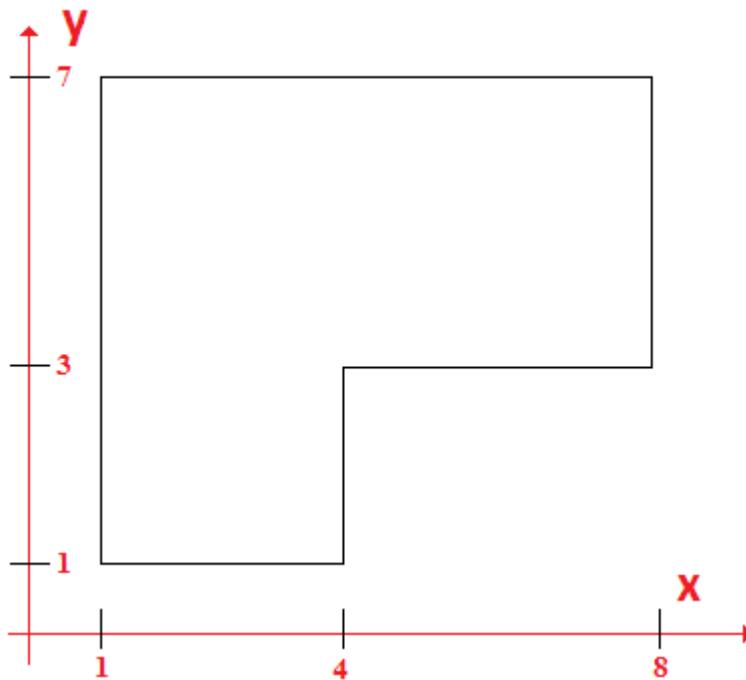
    //T-section
    double flangeH = 0.2;
    double flangeB = 0.7;
    Geometry geometryTSection = new Geometry(h,flangeH, b, flangeB);

    //circular section
    double diameter = 0.9;
    Geometry geometryCircular = new Geometry(diameter);
}
```

For rectangular and T-section bottom left corner is in the center of coordinate system x=0, y=0 and the contour has a counterclockwise orientation.

Internal representation of the circular cross-section is a regular 18-gon (octadecagon).The area of the polygon is equal to area of a circle: $\pi \cdot \text{diameter}^2 / 4$. The center of the circle is in the center of coordinate system x=0, y=0 and the contour has a counterclockwise orientation.

Other section can be defined point by point:



Code region

```
public void CreateGeometry ()
{
    Geometry geometry = new Geometry();
    geometry.Add(1.0, 1.0);
    geometry.Add(4.0, 1.0);
    geometry.Add(4.0, 3.0);
    geometry.Add(8.0, 3.0);
    geometry.Add(8.0, 7.0);
    geometry.Add(1.0, 7.0);
}
```

6.6 RCSolver

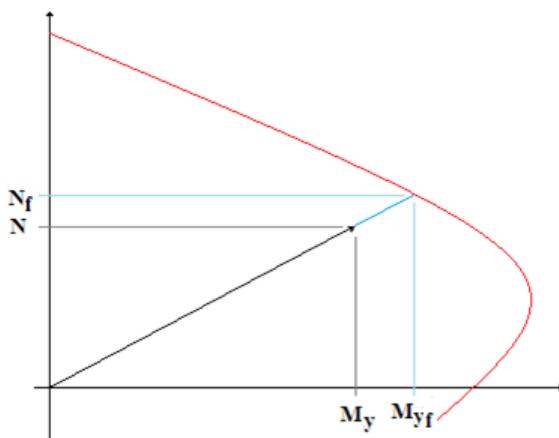
The RCSolver provides methods to analyze RC cross section. This analysis is basing on definition of Geometry, Rebars, Steel and Concrete.

6.6.1 SolveResistance - section resistance

This method finds state of strain and stress at the failure. As parameters this method takes the acting forces N , M_x , M_y . Forces N_f , M_{xf} , M_{yf} at the failure are related to acting forces N , M_x , M_y following the rule:

- $N_f = \alpha \cdot N$
- $M_{xf} = \alpha \cdot M_x$
- $M_{yf} = \alpha \cdot M_y$.

where α is a coefficient called "safety factor".



Code region

```
public void Resistance(RCSolver solver)
{
    double N = 100e3;
    double Mx = 0.0;
    double My = 50e3;
    solver.SolveResistance(N, Mx, My);
    SetOfForces forces = solver.GetInternalForces(ResultType.Section);
    double alpha = N / forces.AxialForce;
}
```

6.6.2 SolveResistanceF - section resistance for fixed moments

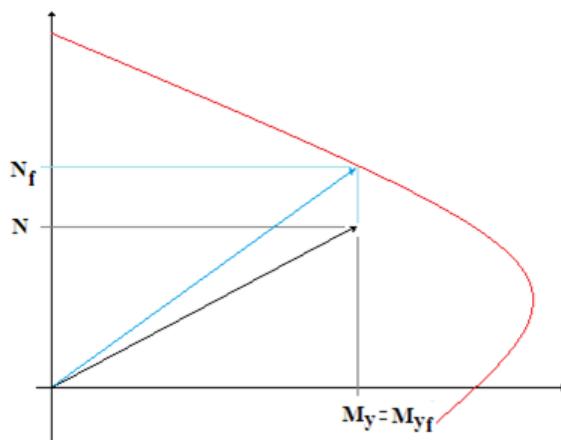
This method finds state of strain and stress at the failure. As a parameters this method takes the value of moment, definition of axis around the moment is acting and definition the sign of result forces. Forces N_f , M_{xf} , M_{yf} at the failure are related to acting forces N , M_x , M_y following the rules:

- N_f
- $M_{xf} = M_x$
- $M_{yf} = 0$

or (depending on the definition of the axes)

- N_f
- $M_{xf} = 0$
- $M_{yf} = M_y$

The α coefficient describes how much axial force must increase up (or decrease) to failure the section.



Code region

```
public void Resistance(RCSolver solver)
{
    double M = 50e3;
    solver.SolveResistanceF(M, Axis.y, true);
    SetOfForces forces = solver.GetInternalForces(ResultType.Section);
    double alpha = M / forces.MomentY;
}
```

6.6.3 SolveResistanceM - section resistance for fixed axial force

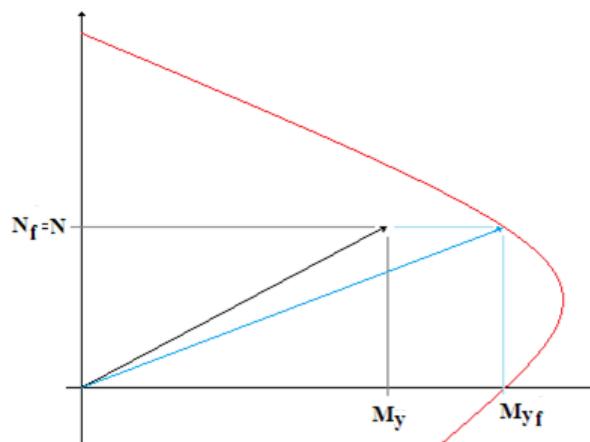
This method finds state of strain and stress at the failure. As a parameters this method takes the value of axial forces, definition of axis around the result moment is acting and definition the sign of result forces. Forces N_f , M_{xf} , M_{yf} at the failure are related to acting forces N , M_x , M_y following the rules:

- $N_f = N$
- $M_{xf} = 0$
- $M_{yf} = 0$

or (depending on the definition of the axes)

- $N_f = N$
- $M_{xf} = 0$
- $M_{yf} \neq 0$

The α coefficient describes how much the moment must increase up (or decrease) to failure the section.

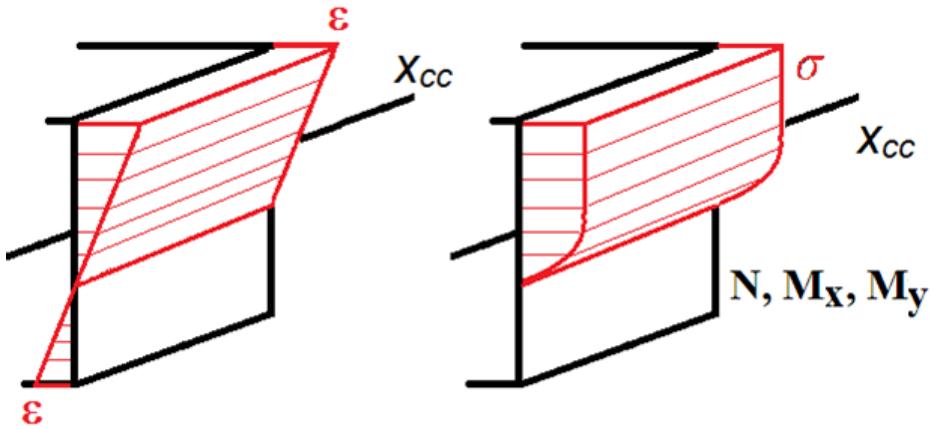


Code region

```
public void Resistance(RCSolver solver)
{
    double N = 100e3;
    solver.SolveResistanceM(N, Axis.y, true);
    SetOfForces forces = solver.GetInternalForces(ResultType.Section);
    double alpha = N / forces.AxialForce;
}
```

6.6.4 SolveForces - forces at a given strain

This method finds forces N , M_x , M_y taking into account given maximum and minimum strains of the cross-section. This method could be used for the whole RC section or only for rebar or concrete.



Code region

```
public void Resistance(RCSolver solver)
{
    double topStrain = 0.0035;
    double bottomStrain = -0.01;
    solver.SolveForces(ResultType.Section, topStrain, bottomStrain);
    SetOfForces forces = solver.GetInternalForces(ResultType.Section);
}
```

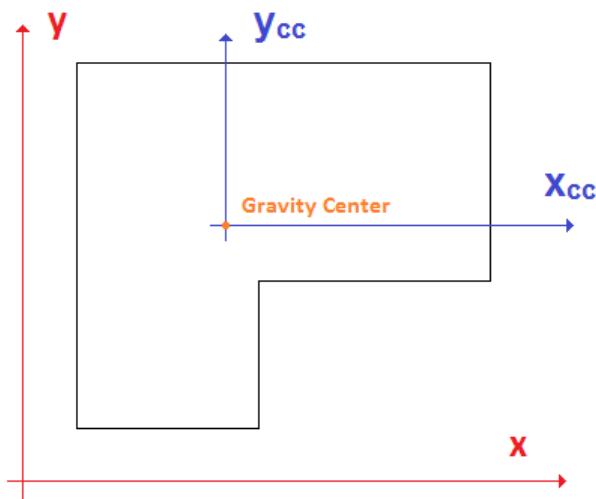
If bending axis is not in x direction the additional parameter is necessary. This parameter is an angle between the vector perpendicular to neutral axis and the x-axis:

Code region

```
public void Resistance(RCSolver solver)
{
    double rightStrain = 0.0035;
    double leftStrain = -0.01;
    solver.SolveForces(ResultType.Section, rightStrain, leftStrain, Math.PI);
    SetOfForces forces = solver.GetInternalForces(ResultType.Section);
}
```

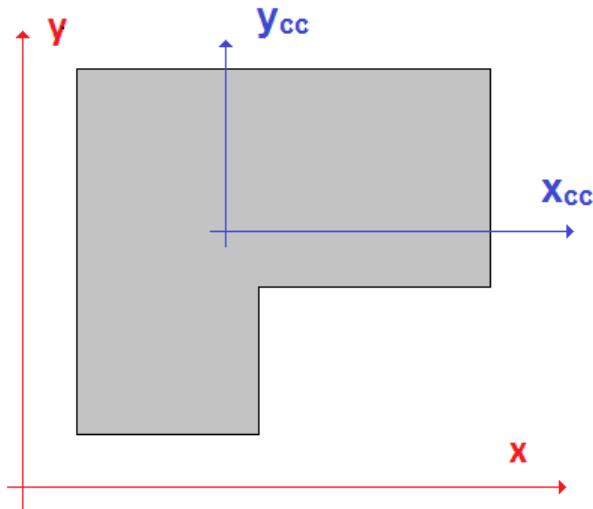
6.6.5 Simple output

These properties are available at any time and are independent of the calculation. These results are: area, moments of inertia, first moments of inertia, and center of inertia. These results are available for pure concrete cross section, only rebars and the RC cross section. The moments are calculated around axis x_{cc} and y_{cc} with origin (0, 0) in the gravity center of the pure concrete cross section.



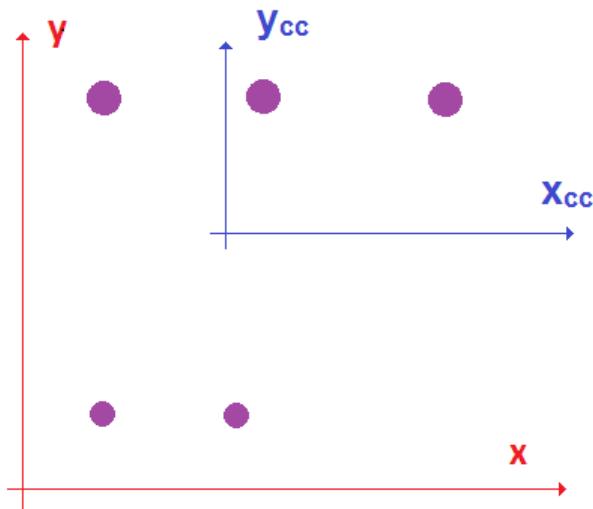
- For pure concrete – ResultType.Concrete
 - GetArea
 - GetMomentOfInertiaX

- o GetMomentOfInertiaY
- o GetCenterOfInertia
- o GetMomentFirstX – for every section is equal 0.0 (center of gravity of concrete)
- o GetMomentFirstY – for every section is equal 0.0 (center of gravity of concrete)



- For pure rebars – ResultType.Rebars

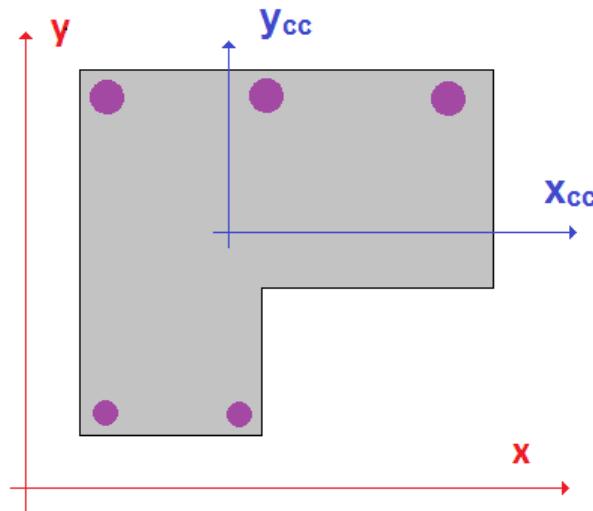
- o GetArea
- o GetMomentOfInertiaX
- o GetMomentOfInertiaY
- o GetCenterOfInertia
- o GetMomentFirstX
- o GetMomentFirstY



- For equivalent section (concrete + reinforcement bars) – ResultType.Section

- o GetArea
- o GetMomentOfInertiaX
- o GetMomentOfInertiaY
- o GetCenterOfInertia

- o GetMomentFirstX
- o GetMomentFirstY



Code region

```
public void SimpleOutput(RCSolver solver)
{
    // result for concrete
    double Ac = solver.GetArea(ResultType.Concrete);
    Point2D Cc = solver.GetCenterOfInertia(ResultType.Concrete);
    double Icx = solver.GetMomentOfInertiaX(ResultType.Concrete);
    double Icy = solver.GetMomentOfInertiaY(ResultType.Concrete);
    // result for rebars
    double As = solver.GetArea(ResultType.Rebars);
    Point2D Cs = solver.GetCenterOfInertia(ResultType.Rebars);
    double Isx = solver.GetMomentOfInertiaX(ResultType.Rebars);
    double Issy = solver.GetMomentOfInertiaY(ResultType.Rebars);
    double Ssx = solver.GetMomentFirstX(ResultType.Rebars);
    double Ssy = solver.GetMomentFirstY(ResultType.Rebars);
    // result for reduced
    double Aeff = solver.GetArea(ResultType.Section);
    Point2D Ceff = solver.GetCenterOfInertia(ResultType.Section);
    double Ieffx = solver.GetMomentOfInertiaX(ResultType.Section);
    double Ieffy = solver.GetMomentOfInertiaY(ResultType.Section);
    double Sx = solver.GetMomentFirstX(ResultType.Section);
    double Sy = solver.GetMomentFirstY(ResultType.Section);
}
```

6.6.6 Calculation results

This outputs are available only after use one of the "calculation methods". Results are depending on the last calculation.

- GetInternalForces – returns the internal forces generated by stresses.
Internal forces are calculated by integration of stress field. These results are available for pure concrete cross section, only rebars and the RC cross section.

- AxialForce - value of axial force:

$$N = \int \sigma dA + \sum \sigma_i \cdot A_i$$

- MomentX - moment around x-axis basing on stress field in the section.

$$M_x = \int \sigma \cdot y_{cc} dA + \sum \sigma_i \cdot y_{cc,i} \cdot A_i$$

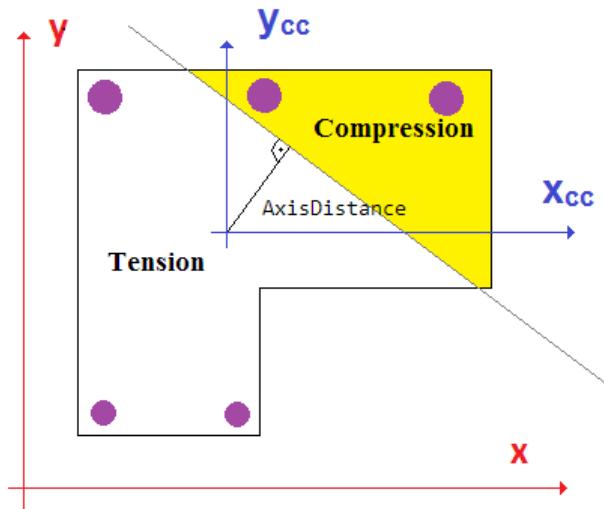
- MomentY - moment around y-axis basing on stress field in the section.

$$M_y = \int \sigma \cdot x_{cc} dA + \sum \sigma_i \cdot x_{cc,i} \cdot A_i$$

Code region

```
public void SolverForces(RCSolver solver)
{
    SetOfForces forces = solver.GetInternalForces(ResultType.Section);
    double N = forces.AxialForce;
    double Mx = forces.MomentX;
    double My = forces.MomentY;
}
```

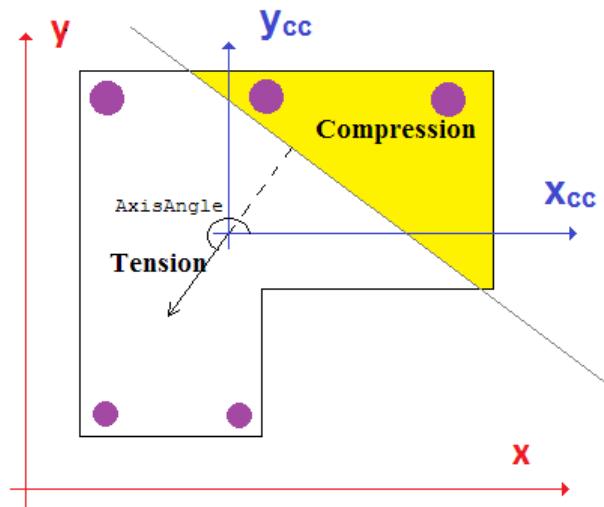
- GetNeutralAxisDistance - distance from gravity center of contour to a neutral axis. Returns value of AxisDistance, see picture below. Neutral axis is a line where the strains are equal zero. This line divides compression and tension part.



Code region

```
public void AxisDistance(RCSolver solver)
{
    double angle = solver.GetNeutralAxisDistance();
}
```

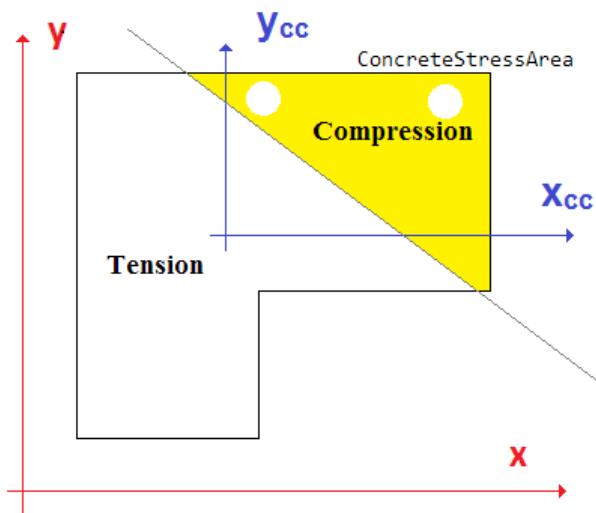
- `GetNeutralAxisAngle` - Angle between the vector perpendicular to the neutral axis and the x-axis.
Returns value of "Angle", see picture below. Neutral axis is a line where the strains are equal zero. This line divides compression and tension part. The sense of the vector indicates direction to the most tensioned fiber.



Code region

```
public void AxisAngle(RCSolver solver)
{
    double angle = solver.GetNeutralAxisAngle();
}
```

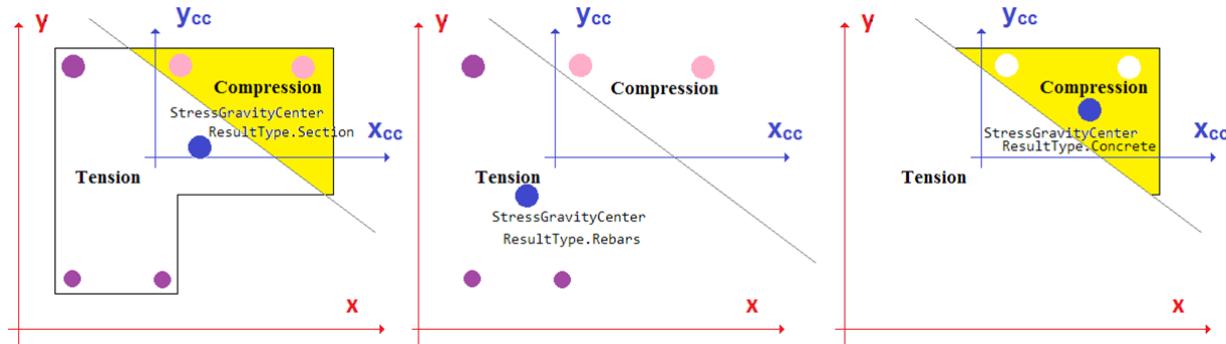
- GetConcreteStressArea – Area of concrete with compressive stresses. Represent value of area of contour under compressive stresses taken into account in the calculation. For concrete with rectangular model compressive stress area is reduced by EffectiveHeightReductionFactor.



Code region

```
public void ConcreteCompressionArea(RCSolver solver)
{
    double angle = solver.GetConcreteStressArea();
}
```

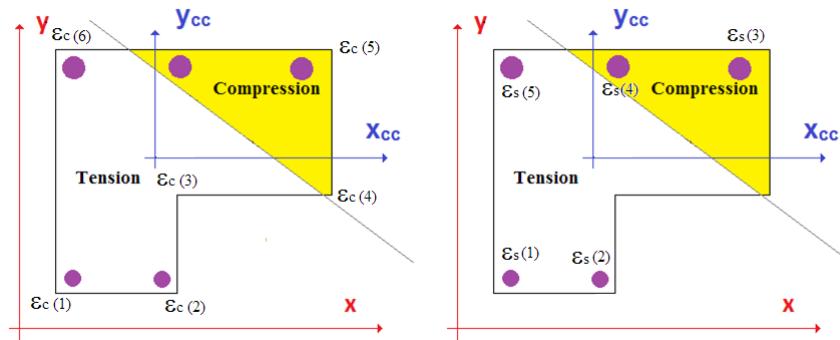
- GetStressGravityCenter – returns center of gravity of stress block. These results are available for pure concrete cross section, pure rebar and the RC cross section.



Code region

```
public void StressGravityCenter(RCSolver solver)
{
    Point2D Gcc = solver.GetStressGravityCenter(ResultType.Concrete);
    Point2D Gcr = solver.GetStressGravityCenter(ResultType.Rebars);
    Point2D Gcs = solver.GetStressGravityCenter(ResultType.Section);
}
```

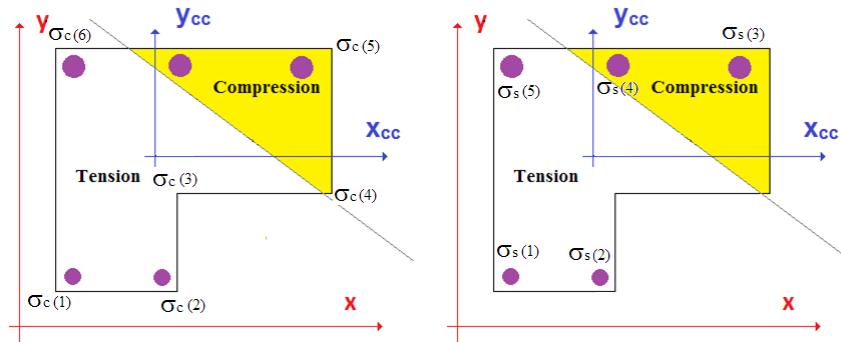
- GetStrain – returns the strains in the cross-section or in the rebar. Compressive strain is positive, tensioning strain is negative.



Code region

```
public void FindExtremeStrain(RCSolver solver)
{
    int rebarNo = solver.GetRebars().Count();
    double strainRebarMin = Double.MaxValue;
    double strainRebarMax = Double.MinValue;
    double strainRebar = 0.0;
    if (rebarNo < 1)
        strainRebarMin = strainRebarMax = 0;
    for (int i = 0; i < rebarNo; i++)
    {
        strainRebar = solver.GetStrain(ResultType.Rebars, i);
        strainRebarMax = Math.Max(strainRebar, strainRebarMax);
        strainRebarMin = Math.Min(strainRebar, strainRebarMin);
    }
    int contourNo = solver.GetGeometry().Count;
    double strainConcreteMin = Double.MaxValue;
    double strainConcreteMax = Double.MinValue;
    double strainConcrete = 0.0;
    if (rebarNo < 1)
        strainConcreteMin = strainConcreteMax = 0;
    for (int i = 0; i < contourNo; i++)
    {
        strainConcrete = solver.GetStrain(ResultType.Concrete, i);
        strainConcreteMax = Math.Max(strainConcrete, strainConcreteMax);
        strainConcreteMin = Math.Min(strainConcrete, strainConcreteMin);
    }
}
```

- GetStress – returns the stress in the cross-section or in the rebar.



Code region

```
public void FindExtremeStrain(RCSolver solver)
{
    int rebarNo = solver.GetRebars().Count();
    double stressRebarMin = Double.MaxValue;
    double stressRebarMax = Double.MinValue;
    double stressRebar = 0.0;
    if (rebarNo < 1)
        stressRebarMin = stressRebarMax = 0;
    for (int i = 0; i < rebarNo; i++)
    {
        stressRebar = solver.GetStress(ResultType.Rebars, i);
        stressRebarMax = Math.Max(stressRebar, stressRebarMax);
        stressRebarMin = Math.Min(stressRebar, stressRebarMin);
    }
    int conturNo = solver.GetGeometry().Count;
    double stressConcreteMin = 0.0;
    double stressConcreteMax = 0.0;
    double stressConcrete = 0.0;
    if (rebarNo < 1)
        stressConcreteMin = stressConcreteMax = 0;
    for (int i = 0; i < conturNo; i++)
    {
        stressConcrete = solver.GetStress(ResultType.Concrete, i);
        stressConcreteMax = Math.Max(stressConcrete, stressConcreteMax);
        stressConcreteMin = Math.Min(stressConcrete, stressConcreteMin);
    }
}
```

6.8 Verification examples

Complete engineering descriptions for the examples shown below are described inside the “Manual Verifications Concrete” document. Implementation examples are also provided with the Concrete Calculations Example part of the SDK solution. On this example, geometry and rebars are defined by end to demonstrate feature however the final goal is to read this data from Revit model.

6.8.1 Case 1

Description:

Cross section characteristics calculation

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel and return cross section characteristics.

6.8.2 Case 2

Description:

Internal forces calculation for given state of strain in a section and rectangular model for concrete.

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and neutral axis angle and return internal forces.

6.8.3 Case 3

Description:

Internal forces calculation for given state of strain in a section and linear model for concrete.

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and neutral axis angle and return internal forces.

6.8.4 Case 4

Description:

Internal forces calculation for given state of strain in a section and bilinear model for concrete.

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and neutral axis angle and return internal forces.

6.8.5 Case 5

Description:

Internal forces calculation for given state of strain in a section and parabolic-rectangular model for concrete.

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and neutral axis angle and return internal

forces.

6.8.6 Case 6

Description:

Internal forces calculation for given state of strain in the section and power-rectangular model for concrete.

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and neutral axis angle and return internal forces.

6.8.7 Case 7

Description:

Internal forces calculation for given state of strain in the section and inclined branch model for steel.

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and return internal forces.

6.8.8 Case 8

Description:

Internal forces calculation in an asymmetric section for given state of strain with horizontal neutral axis

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and return internal forces.

6.8.9 Case 9

Description:

Internal forces calculation in a symmetric section for given state of strain with inclined neutral axis

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, extreme strains in the section and return internal forces.

6.8.10 Case 10

Description:

Calculation of capacity state in a symmetric section for bending moment (M_x)

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (0, M_x , 0) and return state of strain and stress at the failure.

6.8.11 Case 11

Calculation of capacity state in symmetric section for bending moment (M_y)

Goal:

Read cross section geometry , positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (0, 0, My) and return state of strain and stress at the failure.

6.8.12 Case 12

Description:

Calculation of capacity state in a symmetric section for bending moment Mx with axial force

Goal:

Read cross section geometry, positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (N, Mx, 0) and return state of strain and stress at the failure.

6.8.13 Case 13

Description:

Calculation of capacity state in a symmetric section for bending moment My with axial force.

Goal:

Read cross section geometry , positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (N, 0, My) and return state of strain and stress at the failure.

6.8.14 Case 14

Description:

Calculation of capacity state in a symmetric section for bidirectional bending with axial force

Goal:

Read cross section geometry , positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (N, Mx, My) and return state of strain and stress at the failure.

6.8.15 Case 15

Description:

Calculation of capacity state in an asymmetric section for bidirectional bending with axial force

Goal:

Read cross section geometry , positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (N, Mx, My) and return state of strain and stress at the failure.

6.8.16 Case 16

Description:

Calculation of capacity state in a symmetric section for fixed axial force.

Goal:

Read cross section geometry , positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section and return state of strain and stress at the failure for fixed axial force and Mx or My=0.

6.8.17 Case 17

Description:

Calculation of capacity state in a symmetric section for fixed moment

Goal:

Read cross section geometry , positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (Mx or My) and return state of strain and stress at the failure for fixed moment (Mx or My) and the other moment=0.

6.8.18 Case 18

Description:

Calculation of capacity state in symmetric section for fixed negative (tensioning) axial force.

Goal:

Read cross section geometry , positions of reinforcing bars inside the section, parameters of concrete and steel, loads applied to the section (negative N) and return state of strain and stress at the failure for fixed axial force N and Mx or My=0.

7 ENGINEERING COMPONENT

Component: CodeChecking.Engineering.dll

Namespace: Autodesk.Revit.DB.CodeChecking.Engineering

The goal of this component is to expose specific information from a Revit project useful for the Code Checking process.

Different kinds of information are covered here as the category, the geometry and the cover assigned to a Revit element. In case of Framing and column, this component exposes some helpers to get dimensions and mechanical characteristics of the start and end cross section. In addition for slabs, some additional data are provided.

The Code Checking process is based on structural analysis results and for this purposes some helpers are available to get easily this kind of results when they are available for some specific Revit element.

7.1 Element Analyzer

To get Revit element information useful to perform a code check, the `ElementAnalyser` object should be used. This object exposes a method `Analyse` called with as a parameter the Revit element to analyze. This object returns an `ElementInfo` object.

7.1.1 Units

Using the default constructor for the `ElementAnalyser` object, all data will be returned in Revit API units.

The `displayUnitType` enum could be used as well as parameter of the `ElementAnalyzer` constructor and in this case data will be returned in SI or base imperial unit system as described below:

- Metric (SI) units means following units:

Revit Unit Type	Units
UT Length	[m]
UT Section Property	[m]
UT Section Dimension	[m]
UT Section Area	[m ²]
UT Moment of Inertia	[m ⁴]
UT Mass per Unit Length	[kg/m]
UT Stress	[N/m ²]
UT UnitWeight	[kN/m ³]
UT Structural Velocity	[m/s]

- Imperial units means following unit:

Revit Unit Type	Units
UT Length	[in.]
UT Section Property	[in.]
UT Section Dimension	[in.]
UT Section Area	[in. ²]
UT Moment of Inertia	[in. ⁴]
UT Mass per Unit Length	[lb/ft]
UT Stress	[lbf/ft ²]
UT UnitWeight	[lbf /ft ³]
UT Structural Velocity	[ft/s]

Code region

```
Autodesk.Revit.DB.CodeChecking.Engineering.Tools.ElementAnalyser
revitElementAnalyser = null;
// Revit API units
revitElementAnalyser = new ElementAnalyser();
// SI units
revitElementAnalyser = new ElementAnalyser(DisplayUnit.METRIC);
// Imperial units
revitElementAnalyser = new ElementAnalyser(DisplayUnit.IMPERIAL);
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
```

7.1.2 Element

An `ElementInfo` object is returned by the method `Analyse(element)`. Information returned here is related to:

- Material (for metal, wood and concrete material physical asset)
- Cover (for elements where the physical material type is set to concrete)
- Cross section geometry at the start and the end of the element (for beam and columns)
- Cross section parameters at the start and the end of the element (for beam and columns)
- Slab geometry, helpers methods and object to support concrete beam and slab interactions

Following class members are available:

Members	Description
<code>public MaterialInfo Material</code>	Exposes physical material asset data assigned to an element.
<code>public RebarCoversInfo RebarCovers</code>	Exposes cover data assigned to an element
<code>public ElementSectionsInfo Sections</code>	Exposes geometrical information for start and end cross section of a linear element (beam and column)
<code>public ElementSectionsParamsInfo SectionsParams</code>	Exposes dimensions and mechanical characteristics for start and end cross section of a linear element (beam and column)
<code>public ElementSlabsInfo Slabs</code>	Exposes data related to the slab element and interaction with concrete beam
<code>public ElementType Type</code>	Gets the type of an element according a specific classification
<code>public double GeomLength();</code>	Gets the length of the element (beam and column)

7.1.3 Material

The object `MaterialInfo` contains a set of information related to the physical material asset assigned to element.

Properties	Description
<code>public MaterialCategory Category</code>	Gets the category of the material
<code>public MaterialCharacteristics Characteristics</code>	Gets characteristic of the material useful for Code Checking purposes as for example the Young modulus
<code>public MaterialProperties Properties</code>	Gets some addition information as the name of the material

Code region

```
Autodesk.Revit.DB.CodeChecking.Engineering.Tools.ElementAnalyser revitElementAnalyser =
new ElementAnalyser() ;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);

MaterialInfo materialInfo = elementInfo.Material;
MaterialCategory materialCategory = materialInfo.Category;
MaterialCharacteristics materialCharacteristics = materialInfo.Characteristics;
MaterialProperties materialProperties = materialInfo.Properties;
```

7.1.3.1 Material Category

Materials are classified in category. The enum is defined as following:

Enum	Description
MaterialCategory.Undefined	This means that no material is assigned to the element or the Structural Asset Class is Undefined (StructuralAssetClass.Undefined)
MaterialCategory.Metal	This means that the StructuralAssetClass is of type Metal
MaterialCategory.Timber	This means that the StructuralAssetClass is of type Timber
MaterialCategory.Concrete	This means that the StructuralAssetClass is of type Concrete or Precast
MaterialCategory.Other	This means that the StructuralAssetClass is not Metal, Timber, Concrete or undefined

7.1.3.2 Material Characteristics

MaterialCharacteristics object contains main characteristics of the material.

Note: All these values could be taken directly from the Revit API StructuralAssetClass as well.
With this direct access are exposed:

Properties	Description
public double DampingRatio	The damping Ratio of the material
public XYZ PoissonRatio	The Poisson coefficient of the material in 3 directions
public XYZ ShearModulus	The shear module of the material in 3 directions
public object Specific	In case of Metal, Concrete and Wood, this object will be not null and will of the type MaterialMetalCharacteristics, MaterialTimberCharacteristics, MaterialConcreteCharacteristics
public double UnitWeight	The unit weight of the material (Density from Structural Asset multiply by acceleration)
public XYZ YoungModulus	The Young modulus of the material in 3 directions

Code region

```
Autodesk.Revit.DB.CodeChecking.Engineering.Tools.ElementAnalyser revitElementAnalyser =
new ElementAnalyser() ;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);

MaterialInfo materialInfo = elementInfo.Material;
MaterialCategory materialCategory = materialInfo.Category;
MaterialCharacteristics materialCharacteristics = materialInfo.Characteristics;

double dampingRatio = materialCharacteristics.DampingRatio;
XYZ poissonRatio = materialCharacteristics.PoissonRatio;
XYZ shearModulus = materialCharacteristics.ShearModulus;
XYZ thermalExpansionCoefficient = materialCharacteristics.ThermalExpansionCoefficient;
object specific = materialCharacteristics.Specific;
double unitWeight = materialCharacteristics.UnitWeight;
XYZ youngModulus = materialCharacteristics.YoungModulus;
```

7.1.3.2.1 Metal Characteristics

Additional data related to the `StructuralAssetClass` when the material is metal:

Properties	Revit Structural Asset Class
MinimumYieldStress	MinimumYieldStress
MinimumTensileStrength	MinimumTensileStrength
ReductionFactor	MetalReductionFactor
MetalResistanceCalculationStrength	MetalResistanceCalculationStrength

Code region

```
Autodesk.Revit.DB.CodeChecking.Engineering.Tools.ElementAnalyser revitElementAnalyser =
new ElementAnalyser() ;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);

MaterialInfo materialInfo = elementInfo.Material;
MaterialCategory materialCategory = materialInfo.Category;
MaterialCharacteristics materialCharacteristics = materialInfo.Characteristics;

if (materialCategory == MaterialCategory.Metal)
{
    MaterialMetalCharacteristics materialMetalCharacteristics = (MaterialMetalCharacteristics) materialCharacteristics.Specific;
    double minimumTensileStrength= materialMetalCharacteristics.MinimumTensileStrength;
    double minimumYieldStress= materialMetalCharacteristics.MinimumYieldStress ;
    double reductionShearFactor = materialMetalCharacteristics.ReductionShearFactor;
}
```

7.1.3.2.2 Timber Characteristics

Additional data related to the `StructuralAssetClass` when the material is Wood:

Properties	Revit Structural Asset Class
ParallelCompressionStrength	WoodParallelCompressionStrength
ParallelShearStrength	WoodParallelShearStrength
BendingStrength	WoodBendingStrength
PerpendicularShearStrength	WoodPerpendicularShearStrength
PerpendicularCompressionStrength	WoodPerpendicularCompressionStrength

Two additional fields not present on the Structural Asset have been added, AverageModulusG and BurningVelocity.

Code region

```
Autodesk.Revit.DB.CodeChecking.Engineering.Tools.ElementAnalyser revitElementAnalyser =
new ElementAnalyser() ;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);

MaterialInfo materialInfo = elementInfo.Material;
MaterialCategory materialCategory = materialInfo.Category;
MaterialCharacteristics materialCharacteristics = materialInfo.Characteristics;

if (materialCategory == MaterialCategory.Timber)
{
    MaterialTimberCharacteristics materialTimberCharacteristics = (MaterialTimberCharacteristics) materialCharacteristics.Specific;
    double averageModulusG = materialTimberCharacteristics.AverageModulusG;
    double bendingStrength = materialTimberCharacteristics.BendingStrength;
    double burningVelocity = materialTimberCharacteristics.BurningVelocity;
    double parallelCompressionStrength = materialTimberCharacteristics.ParallelCompressionStrength;
    double perpendicularCompressionStrength = materialTimberCharacteristics.PerpendicularCompressionStrength;
    double perpendicularShearStrength = materialTimberCharacteristics.PerpendicularShearStrength;
}
```

7.1.3.2.3 Concrete Characteristics

Additional data related to the StructuralAssetClass when the material is Concrete:

Properties	Revit Structural Asset Class
Compression	ConcreteCompression
BendingReinforcement	ConcreteBendingReinforcement
ShearReinforcement	ConcreteShearReinforcement
ShearStrengthReduction	ConcreteShearStrengthReduction

Code region

```
Autodesk.Revit.DB.CodeChecking.Engineering.Tools.ElementAnalyser revitElementAnalyser =
new ElementAnalyser() ;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);

MaterialInfo materialInfo = elementInfo.Material;
MaterialCategory materialCategory = materialInfo.Category;
MaterialCharacteristics materialCharacteristics = materialInfo.Characteristics;

if (materialCategory == MaterialCategory.Concrete )
{
    MaterialConcreteCharacteristics materialConcreteCharacteristics = (MaterialConcreteCharacteristics) materialCharacteristics.Specific;
    double bendingReinforcement = materialConcreteCharacteristics.BendingReinforcement;
    double compression = materialConcreteCharacteristics.Compression;
    double shearReinforcement = materialConcreteCharacteristics.ShearReinforcement;
    double shearStrengthReduction = materialConcreteCharacteristics.ShearStrengthReduction;
}
```

7.1.3.3 Material Properties

`MaterialProperties` object contains some additional properties of the material as the physical Asset name.

Properties	Description
<code>public string Name</code>	Get the name of the physical asset

7.1.4 Cover

`RebarCoverInfo` object allow developers to check if a cover definition is assigned to an element , to get the cover name and to get the cover value. If the material assigned to the Revit element analyzed is not Concrete, `RebarsCoversInfo` object is null.

`RebarCoverInfo` class exposes following methods:

Methods	Description
<code>public bool Exist(BuiltInParameter coverType);</code>	Check if a specific type of cover exist for an element
<code>public double GetDistance(BuiltInParameter coverType);</code>	Get cover distance
<code>public string GetName(BuiltInParameter coverType);</code>	Get cover name

Code region

```
ElementAnalyser revitElementAnalyser = new ElementAnalyser();
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
RebarCoversInfo rebarCoversInfo = elementInfo.RebarCovers;
if(rebarCoversInfo.Exist(BuiltInParameter.CLEAR_COVER))
{
    String coverName = rebarCoversInfo.GetName(BuiltInParameter.CLEAR_COVER);
    double coverDistance = rebarCoversInfo.GetDistance(BuiltInParameter.CLEAR_COVER);
}
```

Note: Revit API exposes following list of parameters to describe the cover:

- BuiltInParameter.CLEAR_COVER;
- BuiltInParameter.CLEAR_COVER_BOTTOM;
- BuiltInParameter.CLEAR_COVER_TOP;
- BuiltInParameter.CLEAR_COVER_OTHER;
- BuiltInParameter.CLEAR_COVER_EXTERIOR;
- BuiltInParameter.CLEAR_COVER_INTERIOR;

7.1.4.1 Element Sections and Section

ElementSectionsInfo and SectionsInfo objects allow developer to retrieve a geometrical description, dimensions and mechanical characteristics for a cross section at the start and the end of a Revit beam or column.

Members	Description
<code>public SectionsInfo AtTheBeg</code>	Gets the sections at the beginning of the element.
<code>public SectionsInfo AtTheEnd</code>	Gets the sections at the end of the element
<code>SectionsInfo AtThePoint(double relativePoint);</code>	Gets the sections at the point along element length.

The method AtThePoint() return some interpolated values based on the SectionInfo at the beginning and the end.

Code region

```
ElementAnalyser revitElementAnalyser = new ElementAnalyser();
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
ElementSectionsInfo elementSectionsInfo = elementInfo.Sections ;
SectionsInfo sectionsInfoAtTheBeg = elementSectionsInfo.AtTheBeg;
SectionsInfo sectionsInfoAtTheEnd = elementSectionsInfo.AtTheEnd;
SectionsInfo sectionsInfoAtHalf = elementSectionsInfo.AtThePoint(0.5);
```

7.1.4.2 Section

Section object will return the geometry of a cross section. Developers will be able with this object to get on a local system of coordinates (with origin the start or the end point of the element location line) the Shape of the section (contour), internal holes described as a list of shapes and boundaries of the cross sections.

Members	Description
<code>public Shape Contour</code>	List of points (XYZ) describing a cross section
<code>public List<Shape> Holes</code>	List of shapes that representing some holes on a cross section
<code>public void AddHole(List<XYZ> hole);</code>	Method to Add a new hole to the list of shapes that representing holes of inside a cross section.
<code>public XYZ GetMaximumBoundary();</code>	Gets maximum coordinates of upper-right corner of a cross section
<code>public XYZ GetMinimumBoundary();</code>	Gets minimum coordinates of lower-left corner of a cross section

7.1.4.3 Coordinate System

The local coordinate system of a cross section based on the location line of the element. The `CoordinateSystem` object could be used to applied some additional transformation and get for example a cross section description on the global system of coordinates.

7.1.5 Element Sections Parameters and Sections Dimensions

One of the most interesting features of this component is that developers could get for a specific section dimensions and mechanical characteristics based on geometry.

Members	Description
<code>public SectionCharacteristics</code>	Gets the characteristics of section. For double sections or compound sections gets the characteristics for a whole section
<code>public List<SectionCharacteristics> ComponentsCharacteristics</code>	Gets the characteristics of section components (double sections or compound sections)
<code>public List<SectionDimensions> ComponentsDimensions</code>	Gets the dimensions of section components (double sections or compound sections)
<code>public SectionDimensions Dimensions</code>	Gets the dimensions of section. For double sections or compound sections gets the dimensions for a whole section

`SectionsParamsInfo` object is based on the `SectionProfile` base class that exposes following members:

Members	Description
<code>public List<SectionShapeType> ComponentsShapeType</code>	Gets or sets the list of components types of the shapes for double or compound
<code>public SectionShapeType ShapeType</code>	Gets or sets the type of the shape of section
<code>public bool Tapered</code>	Gets or sets a value indicating whether the profile is tapered

7.1.5.1 Shape Type

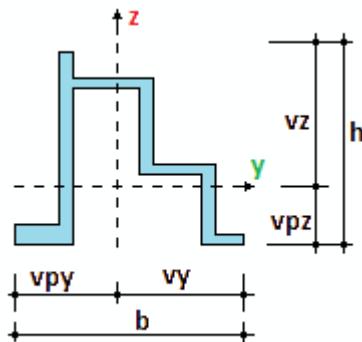
`SectionShapetype` allows developers to classify beam and column cross section according predefined patterns.

The Revit element geometry is analyzed at the start and end points of the location line with the level of details set to medium and the best matching pattern is returned.

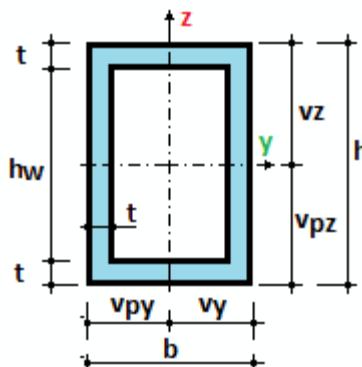
When the section is compound, the `ComponentsShapeType` list is not null

Different types of section are available:

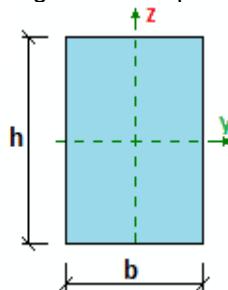
- **Unusual** – the Section doesn't match any patterns



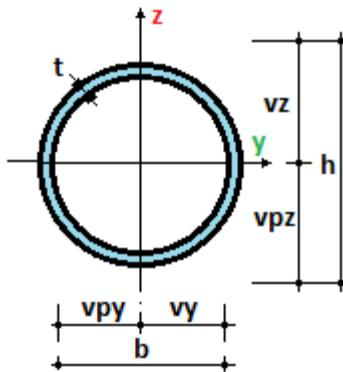
- **RectangularHollowConstant** - the section is recognized as a rectangular tube section where the thickness of the tube is constant



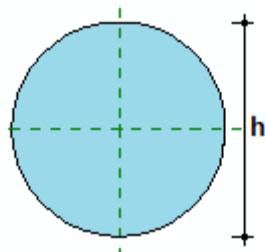
- **RectangularBar** - the section is recognized as a plain rectangular section



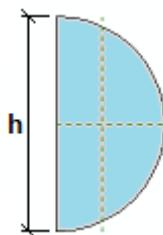
- **RoundTube** - the section is recognized as a circular tube cross section



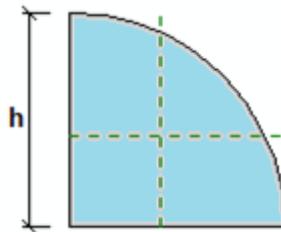
- **RoundBar** - the section is recognized as a plain circular cross section



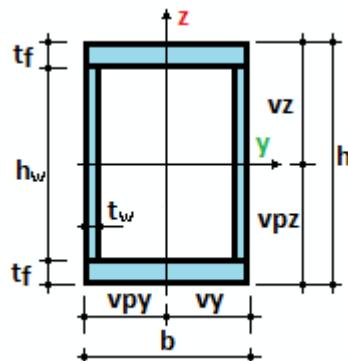
- **HalfRoundBar** - the section is recognized as a plain half-circular cross section



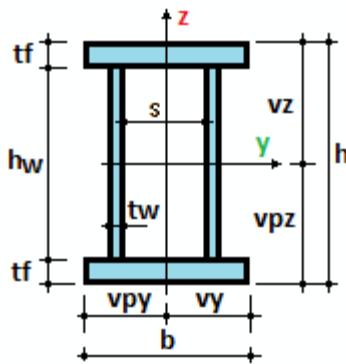
- **QuarterRoundBar** - the section is recognized as a plain quarter-circular cross section



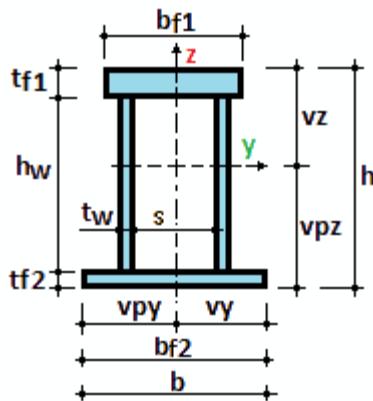
- **RectangularHollowNotConstant** - the section is recognized as a rectangular tube section where thickness of the flange and weldde are different



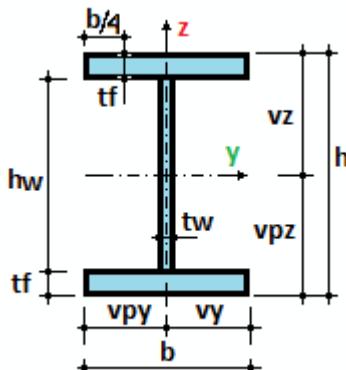
- **BoxBiSymmetrical** - the section is recognized as a bi symmetrical (y and Z axis) box where the flanges are beyond the outline of webs



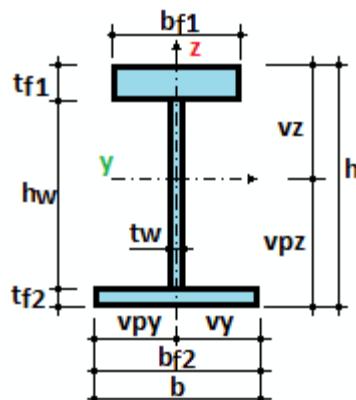
- **BoxMonoSymmetrical** - the section is recognized as a symmetrical (z axis) box where the flanges are beyond the outline of webs



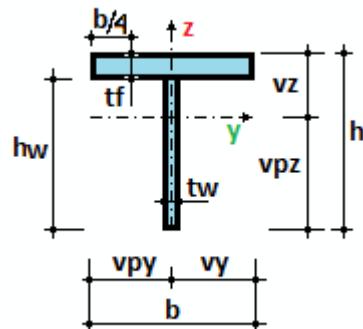
- I - the section is recognized as a Bi symmetrical (y and z axis) I section



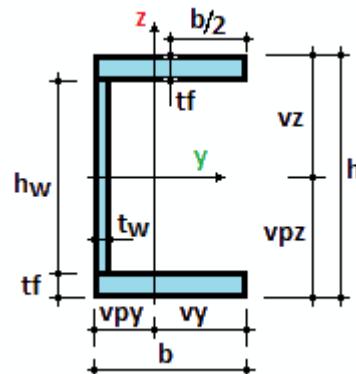
- **IASymmetrical** - the section is recognized as an Asymmetrical (symmetrical around z axis) I section



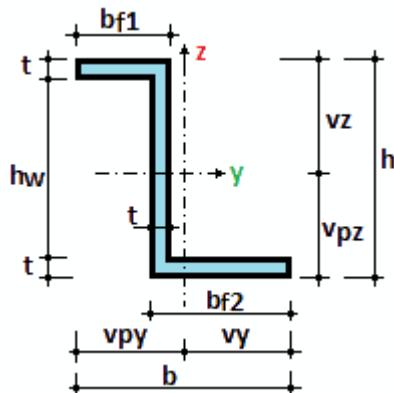
- **T** - the section is recognized as a T section



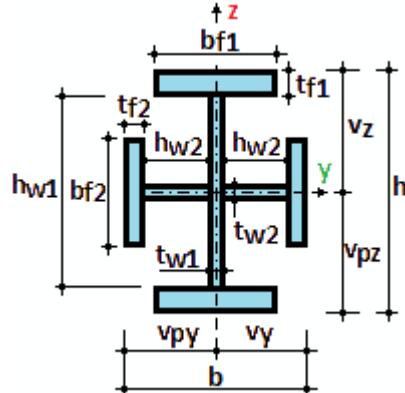
- **C** - the section is recognized as a C section (symmetrical around y axis)



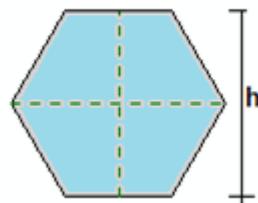
- **Z** - the section is recognized as a Z section



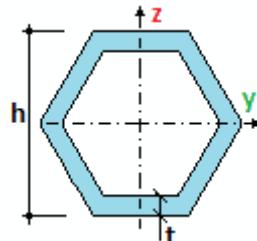
- **CrossBiSymmetrical** - the section is recognized as a bi symmetrical cross



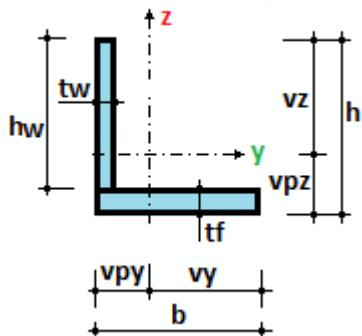
- **PolygonalBar** - the section is recognized as a plain polygonal section



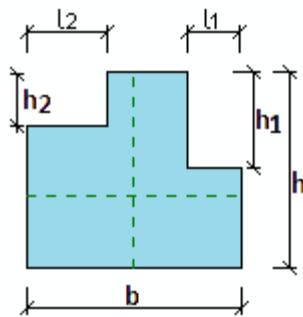
- **PolygonalHollow** - the section is recognized as a polygonal tube cross section. Thickness is constant in this case.



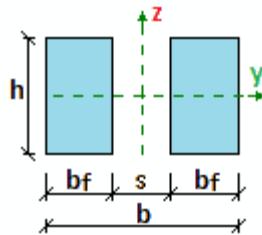
- **L** - the section is recognized as a L section



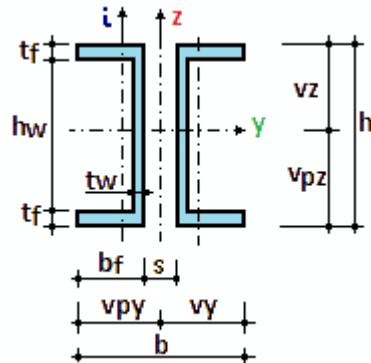
- **TAsymmetrical** - the section is recognized as an Asymmetrical T.



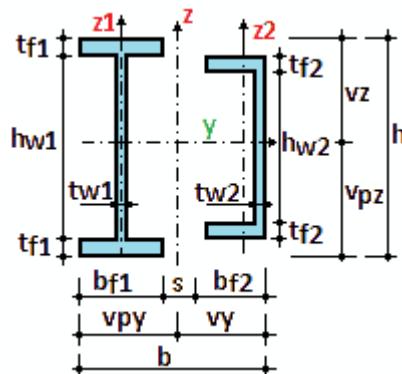
- **DoubleRectangularBar** - the section is recognized as double section composed of 2 plain Rectangular section



- **DoubleSection** - the section is recognized as double section composed of 2 similar cross section



- **CompoundSection** - the section is recognized as a compounded section of n section that could be recognized by patterns



7.1.5.2 Section Characteristics

Main mechanical characteristics are calculated based on the geometry of the Revit element.

Properties	Description
<code>public double A</code>	Area of the cross section area
<code>public double Iy</code>	Moment of inertia around the Y axis
<code>public double Iz</code>	Moment of inertia around the Z axis
<code>public double mass</code>	Nominal weight per unit length
<code>public double ry</code>	Radius of gyration around the Y axis
<code>public double rz</code>	Radius of gyration around the Z axis

7.1.5.3 Section Dimensions

Below is the list of the properties exposed by the `SectionDimensions` class.

Properties and methods	Description
<code>public double b</code>	Width of a cross-section
<code>public double bf</code>	Flange width of a cross-section
<code>public double bf1</code>	Top flange width in case of a symmetric section
<code>public double bf2</code>	Bottom flange width in case of a symmetric section
<code>public double h</code>	Depth of a cross-section
<code>public double h1</code>	Dimension of the main vertical cutout
<code>public double h2</code>	Dimension of the secondary vertical cutout
<code>public double hw</code>	Height of the web
<code>public double hw1</code>	Height of the main web
<code>public double hw2</code>	Height of the secondary web
<code>public double l1</code>	Dimension of the main horizontal cutout
<code>public double l2</code>	Dimension of the secondary horizontal cutout
<code>public double s</code>	Distance between section components
<code>public double t</code>	Thickness of for hollow sections and Z shape
<code>public double tf</code>	Thickness of flanges when constant
<code>public double tf1</code>	Main flange thickness
<code>public double tf2</code>	Secondary flange thickness
<code>public double tw</code>	Web thickness
<code>public double tw2</code>	Secondary web thickness
<code>public double vpy</code>	Horizontal distance from the negative edge of section to the center of gravity
<code>public double vpz</code>	Vertical distance from the negative edge of section to the center of gravity
<code>public double vy</code>	Horizontal distance from the positive edge of section to the center of gravity
<code>public double vz</code>	Vertical distance from the positive edge of section to the center of gravity

7.1.6 Element Slabs

The goal of the `ElementSlabInfo` class is to provide some information related to a Revit floor element (geometry, layers) and some helpers to properly manage interactions between a slab and concrete beams (T section).

Members	Description
<code>public SlabInfo AsElement</code>	if element is a slab gets <code>SlabInfo</code> object
<code>public ElementTSectionInfo TSection</code>	if element is a beam gets T - section if possible

Two object are exposed, SlabInfo, created when the element to analyzed is from the BuiltInCategory.OST_Floor category and ElementTSectionInfo, created when the element to analyzed is from the BuiltInCategory.OST_StructuralFraming category.

When developer would like to analyze the floor based on the analytical model and not the physical model, the boolean TSectionAnalysis should be set to true.

Code region

```
ElementAnalyser revitElementAnalyser = new ElementAnalyser();
revitElementAnalyser.UseAnalyticalModelForSlabs = true;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
ElementSlabsInfo elementSlabInfo = elementInfo.Slabs;
SlabInfo slabInfo = elementSlabInfo.AsElement;
```

When developer would like to analyze a beam as a T section, the boolean TSectionAnalysis should be set to true.

Code region

```
ElementAnalyser revitElementAnalyser = new ElementAnalyser();
revitElementAnalyser.TSectionAnalysis = true;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
ElementSlabsInfo elementSlabInfo = elementInfo.Slabs;
ElementTSectionInfo elementTSectionInfo = elementSlabInfo.TSection;
```

7.1.6.1 Slab

The SlabInfo class describes a Revit slab element by some contours (external and internal) based on a list of edges and a list of nodes. Some additional properties as the family name, the type name and the thickness are available.

Below is the list of members for this class:

Members	Description
public int ContextId	Get the Revit ElementId
public List<SlabContour> Contours	Gets the list of contours describing the slab
public List<SlabEdge> Edges	Gets the list of edges describing the slab
public string FamilyName	Gets the family name
public string LevelName	Gets the level name
public XYZ MaximumBoundary	Gets maximum coordinates (upper-right-front corner) of the slab boundary box
public XYZ MinimumBoundary	Gets minimum coordinates (lower-left-rear corner) of the slab boundary box
public List<SlabNode> Nodes	Gets the list of nodes
public double SpanDirectionAngle	Gets the angle of the span direction
public CoordinateSystem Transform	Allows to transform points from local to global (and vice versa) coordinate system
public string TypeName	Gets the type name
public SlabEdge GetEdge(List<XYZ> pointList)	Returns edge of specific coordinates

<code>public SlabEdge GetEdge(XYZ point1, XYZ point2)</code>	Returns edge of specific coordinates
<code>public SlabContour GetMainContour()</code>	Returns main contour of the slab
<code>public SlabNode GetNode(XYZ point)</code>	Returns node of specific coordinates
<code>public bool IsPointInDomain(XYZ pt)</code>	Checks if a point is in domain of the slab (it has to be inside main contour and it can't be inside any hole)
<code>public double Thickness()</code>	Gets the thickness of slab.

7.1.6.1.1 SlabContour

The Slab contour is defined by his type (main or hole) and is composed of a list of edges and nodes. Some additional methods are exposed to get some additional information, see below the list of members:

Members	Description
<code>public SlabContourType ContourType</code>	Type of slab contour
<code>public List<SlabEdge> Edges</code>	List of edges
<code>public bool IsClockWise</code>	Check if contour is clockwise
<code>public bool IsClosed</code>	Checks if contour is closed
<code>public List<SlabNode> Nodes</code>	List of nodes
<code>public SlabEdge GetEdge(XYZ point1, XYZ point2);</code>	Returns edge of specific coordinates
<code>public bool IsContourInside(List<XYZ> pointList)</code>	Checks if contour is inside current one
<code>public bool IsContourInside(SlabContour contour)</code>	Checks if contour is inside current one
<code>public bool IsCrossingLine(XYZ pt1, XYZ pt2)</code>	Checks if line is crossing current contour
<code>public bool IsPointInside(XYZ pt)</code>	Checks if point is inside contour.

7.1.6.1.2 SlabEdge

Members	Description
<code>public SlabNode NodeEnd</code>	Gets end node
<code>public List<SlabNode> Nodes</code>	Gets nodes
<code>public SlabNode NodeStart</code>	Gets start node
<code>public SlabContour ParentContour</code>	Gets parent contour
<code>public bool ArePointsLikeNodes(List<XYZ> pointList)</code>	Checks if points from the list have the same coordinates as current edge
<code>public XYZ GetCenter()</code>	Returns center of edge
<code>public double GetDistanceFromPoint(XYZ point)</code>	Returns distance to specific point
<code>public bool IsLineCrossingThis(XYZ point1, XYZ point2)</code>	Checks if line is crossing current edge
<code>public bool IsPointOnEdge(XYZ pt)</code>	Checks if specific point is on edge

7.1.6.1.3 SlabNode

The Slab node is basically a XYZ object.

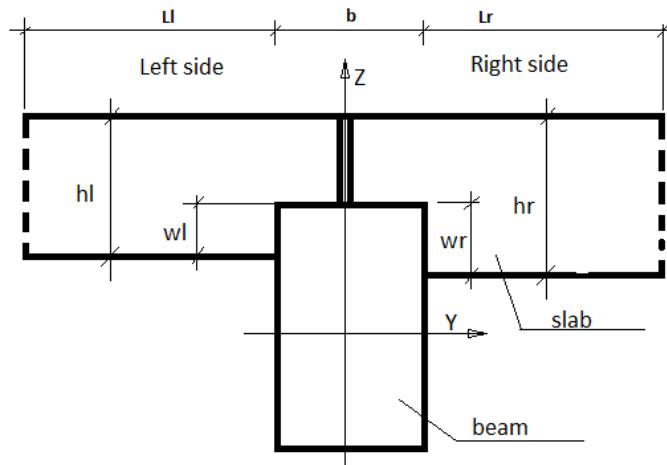
Members	Description
<code>public XYZ Coordinates</code>	Gets or local coordinates
<code>public List<SlabContour> ParentContours</code>	Gets parent contours
<code>public List<SlabEdge> ParentEdges</code>	Gets parent edges
<code>public XYZ GetGlobalCoordinates();</code>	Returns coordinates in global coordinates system

7.1.6.2 T section

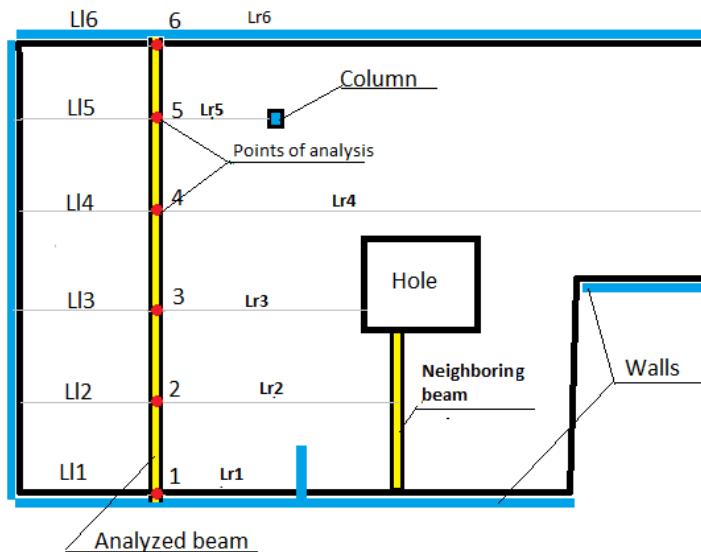
For Concrete Code Checking, it's important to analyze interaction between a concrete beam, adjoining concrete slab and neighbor to be able to calculate it as T section.

The goal of this object is to provide this kind of information. 2 types of data are exposed here:

- T section parameters



- Adjoining elements



Members	Description
<code>public List<SlabInfo> Adjoinings</code>	Gets list of SlabInfo objects that represent adjoining slabs
<code>public SectionsInfo BeamSections</code>	Gets beam section
<code>public int ContextId</code>	Beam Revit ElementId

<code>public Section Contour</code>	Gets contour of the joint section including flanges
<code>public SlabInfo AdjoiningSlabOnTheLeftSide(double relativePosition)</code>	Gets the adjoining slab on the left side of the element at the point along its length
<code>public SlabInfo AdjoiningSlabOnTheRightSide(double relativePosition)</code>	Gets the adjoining slab on the right side of the element at the point along its length.
<code>public double BeamWidth();</code>	Gets width of beam element
<code>public double DistanceToNearestObjectOnTheLeftSide(double relativePosition)</code>	Gets the distance to nearest characteristic object on the left side of the beam element. Distance is calculated at the point specified along the element length
<code>public double DistanceToNearestObjectOnTheRightSide(double relativePosition)</code>	Gets the distance to nearest characteristic object on the right side of the element. Distance is calculated at the point specified along the element length
<code>public Section GetContour(double flangeL, double flangeR)</code>	Gets contour of the joint section including flanges
<code>public double OverlappingDepthOnTheLeftSide(double relativePosition)</code>	Gets the distance from lower slab edge to upper element edge on the left side of the element. Depth is calculated at the point specified along the element length
<code>public double OverlappingDepthOnTheRightSide(double relativePosition)</code>	Gets the distance from lower slab edge to upper element edge on the right side of the element. Depth is calculated at the point specified along the element length
<code>public double SlabThicknessOnTheLeftSide(double relativePosition)</code>	Gets the thickness of the slab on the left side of the element at the point specified along its length
<code>public double SlabThicknessOnTheRightSide(double relativePosition)</code>	Gets the thickness of the slab on the right side of the element at the point specified along its length
<code>public TSectionNearestObjectType TypeOfNearestObjectOnTheLeftSide(double relativePosition)</code>	Gets type of nearest characteristic object on the left side of the element
<code>public TSectionNearestObjectType TypeOfNearestObjectOnTheRightSide(double relativePosition)</code>	Gets type of nearest characteristic object on the right side of the element

Code region

```
ElementAnalyser revitElementAnalyser = new ElementAnalyser();
revitElementAnalyser.TSectionAnalysis = true;
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
ElementInfo elementInfo = revitElementAnalyser.Analyse(activeElement);
ElementSlabsInfo elementSlabInfo = elementInfo.Slabs;
ElementTSectionInfo elementTSectionInfo = elementSlabInfo.TSection;
double position = 0.5;
SlabInfo slabLeft =
    elementTSectionInfo.AdjoiningSlabOnTheLeftSide(position);
SlabInfo slabRight =
    elementTSectionInfo.AdjoiningSlabOnTheRightSide(position);
double slabThicknessOnTheLeftSide =
    elementTSectionInfo.SlabThicknessOnTheLeftSide(position);
double slabThicknessOnTheRightSide =
    elementTSectionInfo.SlabThicknessOnTheRightSide(position);
double overlappingDepthOnTheLeftSide =
    elementTSectionInfo.OverlappingDepthOnTheLeftSide(position);
double overlappingDepthOnTheRightSide =
    elementTSectionInfo.OverlappingDepthOnTheRightSide(position);
TSectionNearestObjectType typeOfNearestObjectOnTheLeftSide =
    elementTSectionInfo.TypeOfNearestObjectOnTheLeftSide(position);
TSectionNearestObjectType typeOfNearestObjectOnTheRightSide =
    elementTSectionInfo.TypeOfNearestObjectOnTheRightSide(position);
```

7.2 Force Result Cache

`Engineering.Tools.ForceResultCache` is a class designed to store and to explore results of structural calculations in Revit (from `ResultsBuilder`). It consumes data provided by `Results Builder`, caches it internally and finally provides developer with a convenient interface to explore these results. The general idea behind this class is to make results exploration in Code Checking faster and more convenient.

API comprises `ForceResultCache` class and several helper classes used mainly for `ForceResultCache` parameterization:

- `ForceLoadCaseDescriptor`
- `ForceResultsPackageDescriptor`
- `ForceCalculationDataDescriptor`
- `ForceCalculationDataDescriptorLinear`

7.2.1 ForceLoadCaseDescriptor

`ForceLoadCaseDescriptor` is a simple class used as storage for all data related to load case or load combination in Revit.

It is created internally by `ForceResultCache` and the only public members are properties representing values stored in the object:

- `Id` is `ElementId` of Revit Element
- `Name` is a `String` containing load name
- `Type` is parameter of `LoadType` type saying whether object pointed by the `Id` is a simple load or a load combination.

It can take following values:

- `LoadType.Unknown`
- `LoadType.SimpleCase`

- LoadType.Combination.
- Finally State parameter indicates the loads limit state with possible values being:
 - ForceLimitState.Unknown
 - ForceLimitState.Uls
 - ForceLimitState.Sls
 - ForceLimitState.Sls

7.2.2 ForceResultsPackageDescriptor

`ForceResultsPackageDescriptor` is another simple container. It identifies and describes Result Builder `ResultsPackage`.

It has following public properties available for the user:

- `Name` is a String containing package name
- `Guid` is Guid of the Package

It also has two static public methods for `ForceResultsPackageDescriptor` creation:

Code region

```
static public ForceResultsPackageDescriptor
GetResultPackageDescriptor(Autodesk.Revit.DB.Document document, Guid guid)

static public List<ForceResultsPackageDescriptor>
GetAllResultPackageDescriptors(Autodesk.Revit.DB.Document document)
```

First method creates a `ForceResultsPackageDescriptor` object for a `ResultsPackage` with a known package Id, and the second creates a List of `ForceResultsPackageDescriptor` objects for all `ResultsPackages` in Revit document.

7.2.3 ForceCalculationDataDescriptor

`ForceCalculationDataDescriptor` class identifies Revit Element along with its force types for all elements for which calculation results are to be cached within `ForceResultCache`.

Objects of this class are created by the user prior to `ForceResultCache` creation (a collection of `ForceCalculationDataDescriptor` objects is passed to `ForceResultCache` constructor).

Objects of this class are created only by the public constructor where passed parameters identify Revit Element and its force types.

Code region

```
public ForceCalculationDataDescriptor(ElementId elementId, IEnumerable<ForceType>
forceTypes = null)
```

After object creation, its data can be queried by three public properties:

- `ElementId` is the Revit element Id.
- `ForceTypes` is a list of `ForceType` objects that identifies forces that have to be cached.

- `ForceTypesHashed` is a string that contains force types description in a compact form which can later be used for sorting purposes.

7.2.4 ForceCalculationDataDescriptorLinear

`ForceCalculationDataDescriptorLinear` class inherits from

`ForceCalculationDataDescriptor`.

It is meant to provide more detailed information about the way the results are to be cached for linear elements such as structural beams and columns.

Standard constructor has following syntax:

Code region

```
public ForceCalculationDataDescriptorLinear(ElementId elementId, double elementLength,
IEnumerable<double> points, bool isInputRelative, List<ForceType> forceTypes)
```

The additional parameter `IEnumerable<double> points` is a list of point linear coordinates in which results are to be read from Result Builder and then internally stored. Two other parameters carry information saying whether point coordinates are relative or absolute.

Two additional properties are exposed:

- `Points` is a list of double representing point coordinates (relative)
- `PointsHashed` – is a string containing compact representation of point coordinates that can be used when comparing two elements

7.2.5 ForceResultCache

`ForceResultCache` class usage follows a three steps pattern:

1. Caching the results – through `ForceResultCache` object creation.

The whole process of reading of results from `ResultsPackage` and storing them in memory takes place during `ForceResultCache` object creation.

After `ForceResultCache` is created no additional data can be added to the cache.
Object is instantiated by the standard constructor:

Code region

```
public ForceResultsCache(
Autodesk.Revit.DB.Document document,
ICollection<ForceCalculationDataDescriptor>elementIDs,
ICollection<ForceResultsPackageDescriptor> resultPackDescriptors,
ICollection<ElementId> loads, DisplayUnit unitSystem)
```

Where parameters have following meaning:

- `document`, is the `Autodesk.Revit.DB.Document` object that contains the model,
- `elementIDs` is a collection of `ForceCalculationDataDescriptor` objects specifying the detailed information on how to cache the data (which elements, what forces, in which calculation points).

For linear elements `ForceCalculationDataDescriptorLinear` should be used.

- `resultPackDescriptors` is a collection of `ForceResultsPackageDescriptor` objects related to results packages from which results are to be read
- `loads` is a collection of `ElementId` of Revit load cases and load combinations for which results are to be read
- `unitSystem` is the Revit `DisplayUnit` unit object specifying units in which results will be presented

2. Querying metadata of imported results

Once `ForceResultCache` object is created (and data has been imported in the process) user can read various information describing results that have been cached:

- `public List<CalcPoint> GetCalculationPoints(ElementId elementId)`
This method returns a list of `CalcPointLinear` or `CalcPointSurface` objects (both inherited from abstract class `CalcPoint`) which provides information on point coordinates of results.
`CalcPointLinear` object has two properties of type `double`, `CoordRelative` and `CoordAbsolute` and both represent linear coordinate of the point (relative and absolute values respectively) and `CalcPointSurface` has one property of `XYZ` type that specifies point position in global coordinate system.
Apart from providing point coordinates `CalcPoint` objects are used as input parameters in methods that query results.
- `public ElementResultsStatus GetElementResultsStatus(ElementId elementId)`
This method returns an `ElementResultsStatus` object for a given element describing its result status. This property can be queried by users if they need general information about element results correctness.
Possible values are:
 - `NoResults` – no results have been imported for the element
 - `ResultsNotComplete` – some, but not all requested results have been imported
 - `ResultsOk` – all requested results have been imported
- `public List<ForceLoadCaseDescriptor> GetLoadCaseDescriptors()`
Returns a list of `ForceLoadCaseDescriptor` objects. They describe all load cases and load combinations of imported results.
Apart from providing general information related to loads such as name and type, `ForceLoadCaseDescriptor` objects are used as input parameters in querying results.

3. Querying results

There are two types of results query:

- Results query for a given calculation point.

Code region

```
public List<double> GetForceForPoint(ElementId elementId, ForceResultsPackageDescriptor packageDescriptor, IEnumerable<ForceLoadCaseDescriptor> loads, CalcPoint calculationPoint, ForceType forceType)

public List<double> GetForceForPoint(ElementId elementId, IEnumerable<ForceLoadCaseDescriptor> loadCaseDescriptors, CalcPoint calculationPoint, ForceType forceType)
```

Both methods return a list of double containing results in one calculation point for several loads.

The meaning of parameters is as follows:

- ElementId elementId - Revit element id
- IEnumerable<ForceLoadCaseDescriptor> loadCaseDescriptors - collection of load case descriptors for which results are to be read. The order of elements corresponds to the order inside the result list.
- CalcPoint calculationPoint - a calculation point for which results will be read
- ForceType forceType - force type of results

The first of the two methods has an additional argument: ForceResultsPackageDescriptor packageDescriptor which determines the result package for results. The second method uses first available package.

- Results query for a given load.

Code region

```
public List<double> GetForceForCase(ElementId elementId, ForceResultsPackageDescriptor resultPackageDescriptor, ForceLoadCaseDescriptor caseDescriptor, IEnumerable<CalcPoint> calculationPoints, ForceType forceType)

public List<double> GetForceForCase(ElementId elementId, ForceLoadCaseDescriptor caseDescriptor, IEnumerable<CalcPoint> calculationPoints, ForceType forceType)
```

Two methods listed above return a list of double containing results for one load in several calculation points. The meaning of parameters is as follows:

- ElementId elementId - Revit element id
- ForceLoadCaseDescriptor loadCaseDescriptor - load case descriptors for which results are to be read.
- IEnumerable<CalcPoint> calculationPoints - a collection of calculation points for which results will be read. The order of elements corresponds to the one on the result list.
- ForceType forceType - force type of results

Analogically the first of two methods allows for a specification of the package from which results will be read.