

# TABLE OF CONTENTS

<b>1</b>	<b>STRUCTURAL TOOLKIT 2016 .....</b>	<b>3</b>
<b>2</b>	<b>SCOPE .....</b>	<b>3</b>
<b>3</b>	<b>LIMITATIONS .....</b>	<b>3</b>
3.1	SUPPORTED ELEMENTS.....	3
3.2	SUPPORTED CROSS SECTIONS .....	4
3.3	DESIGNS SUPPORTED FOR COLUMNS .....	4
3.4	DESIGNS SUPPORTED FOR BEAMS .....	4
3.5	DESIGNS SUPPORTED FOR FLOORS, AND SLAB FOUNDATIONS .....	4
3.6	DESIGNS SUPPORTED FOR WALLS .....	4
<b>4</b>	<b>ASSUMPTIONS.....</b>	<b>5</b>
<b>5</b>	<b>DEFINITIONS .....</b>	<b>5</b>
<b>6</b>	<b>DESIGN PROCESS .....</b>	<b>5</b>
6.1	ELEMENT .....	5
6.2	CROSS SECTION.....	6
<b>7</b>	<b>A NEW CODE CHECKING CONCRETE PROJECT.....</b>	<b>8</b>
<b>8</b>	<b>NOTATION AND REMARKS .....</b>	<b>10</b>
<b>9</b>	<b>ELEMENT DESIGN .....</b>	<b>11</b>
9.1	RCTYPES CLASS .....	11
9.2	LABELCOLUMN CLASS.....	12
9.3	LABELBEAM CLASS .....	13
9.4	LABELFLOOR CLASS .....	15
9.5	LABELWALL CLASS .....	17
9.6	DEFLECTION CALCULATION .....	20
9.7	MODIFICATION OF INPUT FORCES .....	21
9.8	MODIFICATION OF DESIGN PROCESS SCENARIO .....	25
<b>10</b>	<b>CRACKING AND STIFFNESS.....</b>	<b>27</b>
10.1	MOMENT OF INERTIA FOR CRACKING SECTION.....	27
10.2	RATIO ACTING FORCES TO FORCES GIVING THE FIRST CRACKING .....	28
<b>11</b>	<b>CROSS SECTION DESIGN .....</b>	<b>29</b>
11.1	CLASS MEMBERS.....	29
11.2	NEW CLASS MEMBERS .....	30
11.3	MAIN DESIGN METHOD.....	32
11.4	VERIFICATION OF INPUT DATA .....	34
11.5	REDUCTION OF THE SET OF FORCES.....	35
11.6	LONGITUDINAL AND TRANSVERSAL REINFORCEMENT CALCULATION .....	38
11.7	CHECKING OF ZERO VALUE OF FORCES .....	40
11.8	MATERIAL PROPERTIES ACCORDING TO LIMIT STATE .....	41
11.9	CROSS SECTION STIFFNESS CALCULATION.....	42
11.10	LONGITUDINAL REINFORCEMENT - ULS AND SLS .....	43
11.11	MINIMUM REINFORCEMENT FOR LONGITUDINAL REINFORCEMENT .....	45
11.12	TRANSVERSAL REINFORCEMENT CALCULATION FOR ULS .....	47
11.13	SIMPLIFIED LONGITUDINAL REINFORCEMENT DESIGN .....	47
11.14	LONGITUDINAL REINFORCEMENT FOR UNIAXIAL/BIAXIAL BENDING .....	49
11.14.1	CalculateLongitudinalReinforcementUniaxial.....	49

11.14.2	CalculateLongitudinalReinforcementBiaxial .....	50
11.15	CROSS SECTION STIFFNESS CALCULATION.....	51
11.16	SYMMETRICAL AND UNSYMMETRICAL REINFORCEMENT FOR UNIAXIAL BENDING .....	52
11.17	SEARCH FOR THE OPTIMAL LONGITUDINAL REINFORCEMENT.....	53
11.18	FINAL REINFORCEMENT ADJUSTMENTS .....	57
11.19	BASIC TRANSVERSE REINFORCEMENT CALCULATION.....	58
11.20	COMPLEX CALCULATION FOR TRANSVERSE REINFORCEMENT.....	60
11.21	MAXIMUM STIRRUPS SPACING .....	61
11.22	ADJUSTMENT OF THE REINFORCEMENT INCREMENT.....	62
12	SUMMARY .....	62

# 1 STRUCTURAL TOOLKIT 2016

No changes have been introduced in the version 2016.

In the version 2015 of the Structural Toolkit, an implementation of required reinforcement for surface elements has been introduced. These types of objects are: Analytical concrete floor, Analytical concrete wall and concrete slab foundation.

All new parts of code or changes have been made in the step by step example was tagged:

Code region

```
/// <structural_toolkit_2015>  
Something new or changing  
/// </structural_toolkit_2015>
```

The same tag was used in the C# code shown in “Code region” in this document.

In this document, chapters where changes in the commentary have been made was tagged "(v2015)". Headers of “Code region” where changes have been made are flagged “Updated version 2015” or “New version 2015”.Goal (v2015)

This example shows a basic implementation of a required reinforcement concrete code for beams, – columns, floor, wall and slab foundation. It presents the concept of concrete design and an easy way to create a required reinforcement concrete code based on the Code Checking Framework and SDK features. This implementation is not connected to any national regulation, but is a simplified compilation of Eurocode, ACI and many others.

## 2 SCOPE

This example presents a reinforced concrete beams, columns, floor, wall and slab foundation design for typical situation described in regulations. The result of this example is a longitudinal and transversal required reinforcements developed according to acting loads, geometry of elements and the properties of materials.

The base of this example is a C# project generated using the Code Checking Concrete template.

## 3 LIMITATIONS

As it is only an example of implementation, this code has some limitations relating to elements, cross sections and designs type. In the final implementation of a national standard these limitations could be maintained or not in accordance to local market requirements.

### 3.1 Supported elements

- Analytical concrete beams
- Analytical concrete columns
- Analytical concrete floor
- Analytical concrete wall
- Concrete slab foundation

## 3.2 Supported cross sections

- Rectangular section
- T section (interaction beams/slab)

## 3.3 Designs supported for columns

- Ultimate Limit State: Calculation of longitudinal reinforcement for uniaxial and biaxial bending with and without axial force.
- Serviceability Limit State: Calculation of longitudinal reinforcement for uniaxial and biaxial bending with and without axial force with limits of stress in steel and concrete.
- Slender effects to increase bending moments along element.
- Calculation of transverse reinforcement for shear, biaxial shear and torsion forces, for all combinations.

## 3.4 Designs supported for beams

- Ultimate Limit State: Calculation of longitudinal required reinforcement for uniaxial bending with and without axial force.
- Serviceability Limit State: Calculation of longitudinal required reinforcement for uniaxial bending with and without axial force with limits of stress in steel and concrete.
- Modification of the bending moments over supports base on the maximum value of moment along the element.
- Serviceability Limit State: Deflection taking into account cracking and creep effects.
- Calculation of required transverse reinforcement for shear, biaxial shear and torsion forces, for all combinations.

## 3.5 Designs supported for floors, and slab foundations

- Calculations of primary and secondary longitudinal required reinforcement are independent for both directions.
- Main and secondary directions are orthogonal. Main direction corresponding to the “span direction” and X-direction in the LCS of the analytical element.
- Calculations are based on nodes results that have been saved in the Result Builder.
- Ultimate Limit State: Calculation of longitudinal required reinforcement for uniaxial bending with and without axial force.
- Serviceability Limit State: Calculation of longitudinal required reinforcement for uniaxial bending with and without axial force with limits of stress in steel and concrete.
- All calculations are based on this set of internal forces:
  - $M_{xx}$ ,  $N_{xx}$  or  $M_{xx}$  and  $N_{xx}$  for primary reinforcement
  - $M_{yy}$ ,  $N_{yy}$  or  $M_{yy}$  and  $N_{yy}$  for secondary reinforcement

## 3.6 Designs supported for walls

- Calculations of horizontal and vertical longitudinal required reinforcement are independent for both directions.
- The vertical direction is corresponding to X-direction in the LCS of the analytical element.
- Calculations are based on the node results that have been saved in the Result Builder.
- Ultimate Limit State: Calculation of longitudinal required reinforcement for axial force.
- Serviceability Limit State: Calculation of longitudinal required reinforcement for force with limits of stress in steel and concrete.
- All calculations are based on this set of internal forces:

- Nxx for vertical reinforcement
- Nyy for horizontal reinforcement

## 4 ASSUMPTIONS

This example is based on the metric units system (SI). This means that all internal equations and functions are calibrated according to this units system.

For this reason, input data are converted to SI and results are stores accordingly.

Only mandatory data are stored and only the required reinforcement is calculated.

## 5 DEFINITIONS

- ULS: Ultimate Limit State (dimensioning of the state of structural damage).
- SLS: Serviceability Limit State (dimensioning of the state of loss of functionality/serviceability)
- Set of forces: Set of internal forces ( $F_x$ ,  $F_y$ ,  $F_z$ ,  $M_x$ ,  $M_y$ ,  $M_z$ ) for linear elements (beam and column) or ( $M_{xx}$ ,  $M_{yy}$ ,  $N_{xx}$ ,  $N_{yy}$ ,  $Q_{xx}$ ,  $Q_{yy}$ ) for surface elements (floor, wall, slab foundation) which are generated by a simple load case or a load combination
- Sets of forces list: List of sets of forces based on the list of load combinations or simple load cases
- Effect of slenderness: Additional action effects caused by structural deformations (second order effects)

## 6 DESIGN PROCESS

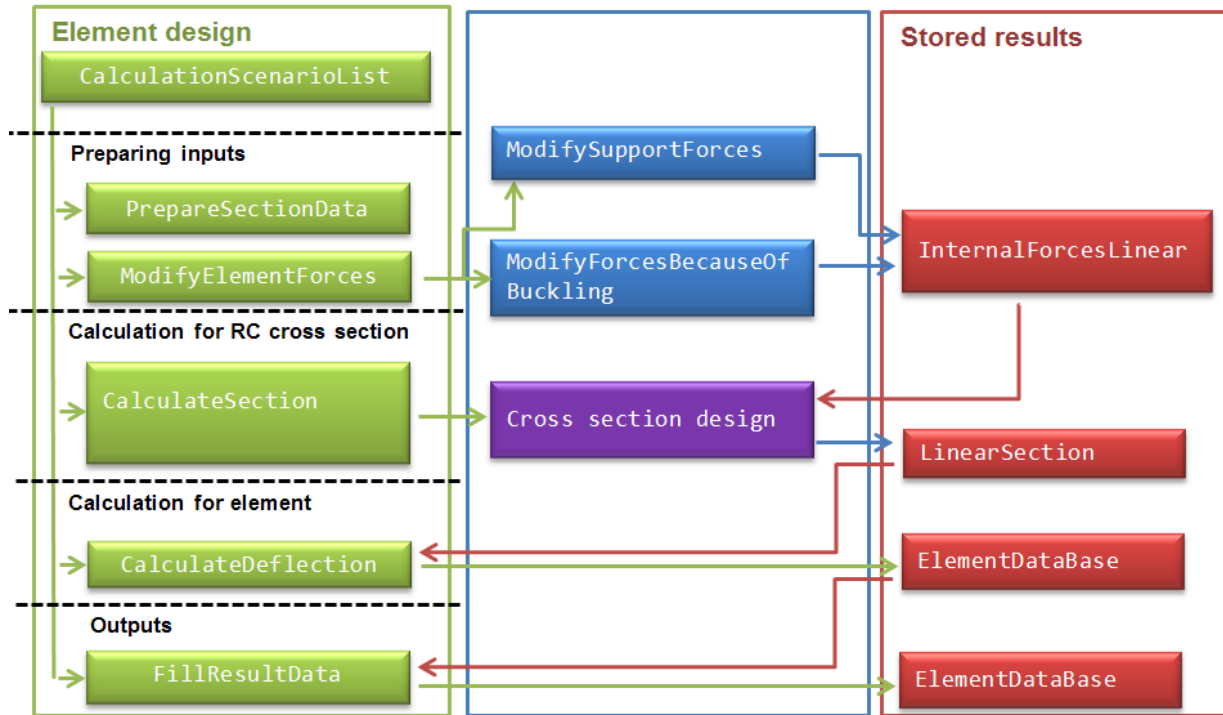
The design process can be divided into two parts related to element and cross section:

- The part related to element allows modification of bending moments taken from static analysis for beams, columns, floors, walls and slabs foundation and calculation of deflection for beams and floors.
- The part related to cross section allows calculation of longitudinal and transversal required reinforcement. In this part the stiffness of the cross section is calculated as well.

### 6.1 Element

At the element level, two options are included in the example:

- First option modifies the internal forces (bending moment) along the element:
  - For beams, over the supports base on the maximum value of moment along element
  - For columns, along the whole element length according to the slenderness effect
- Second option is the deflection of for beams. This kind of calculation is based on stiffness of cross section and the static displacement.



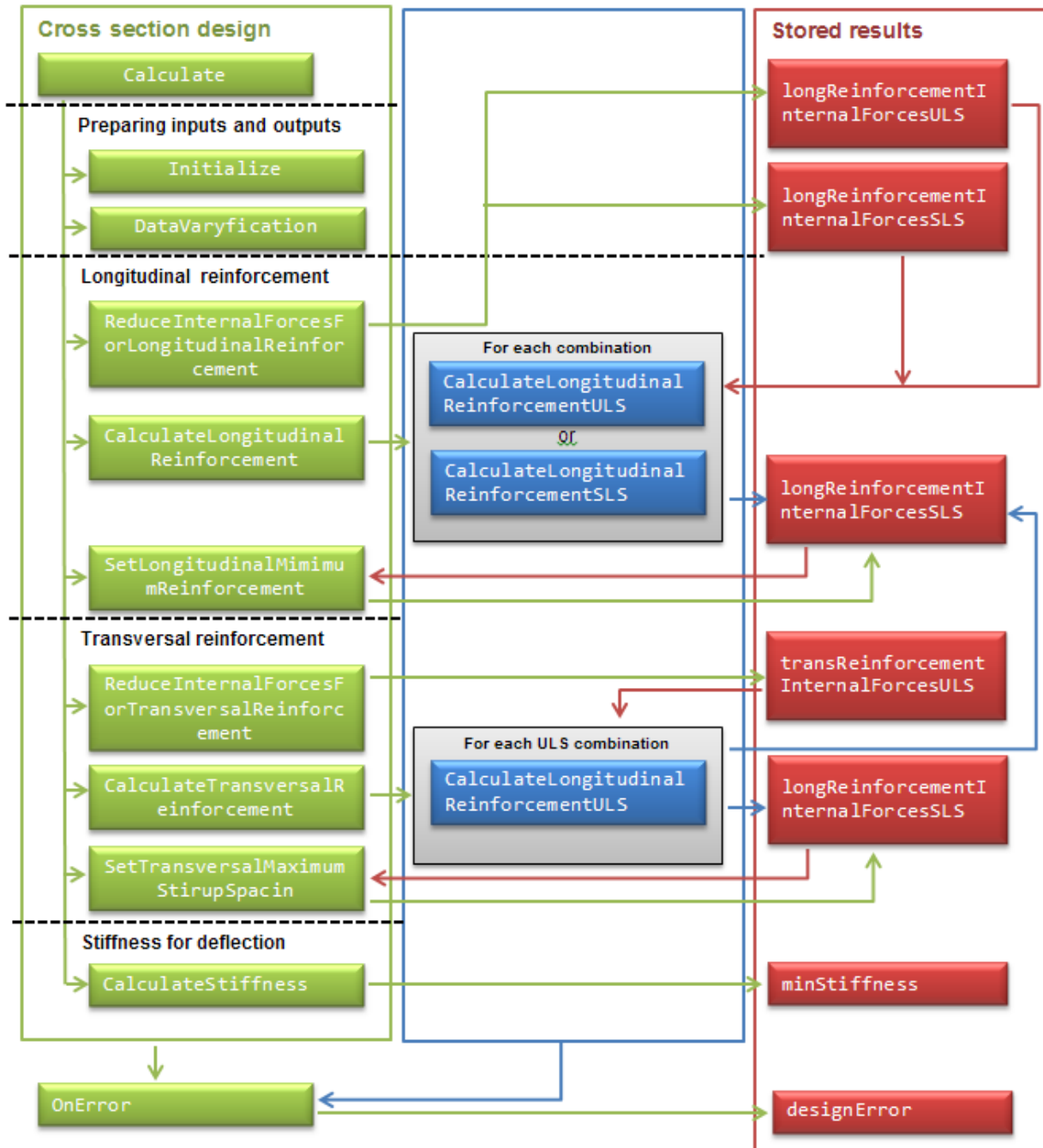
## 6.2 Cross section

The entry point for the cross sections calculation process is the method `Calculate()` expose inside the `ConcreteSectionDesign` class.

In this method the following actions are performed:

- Initializing of internal calculation objects
- Verifying input data
- Dimensioning the longitudinal reinforcement for ULS
- Dimensioning the longitudinal reinforcement for SLS
- Finding the minimum area needed for longitudinal reinforcement
- Calculating the stiffness (only for beam sections)
- Dimensioning the transverse reinforcement for ULS for beams and columns
- Finding the minimum spacing for transverse reinforcement for beams and columns

If calculations were not successful, some errors will be caught and stored at this level.

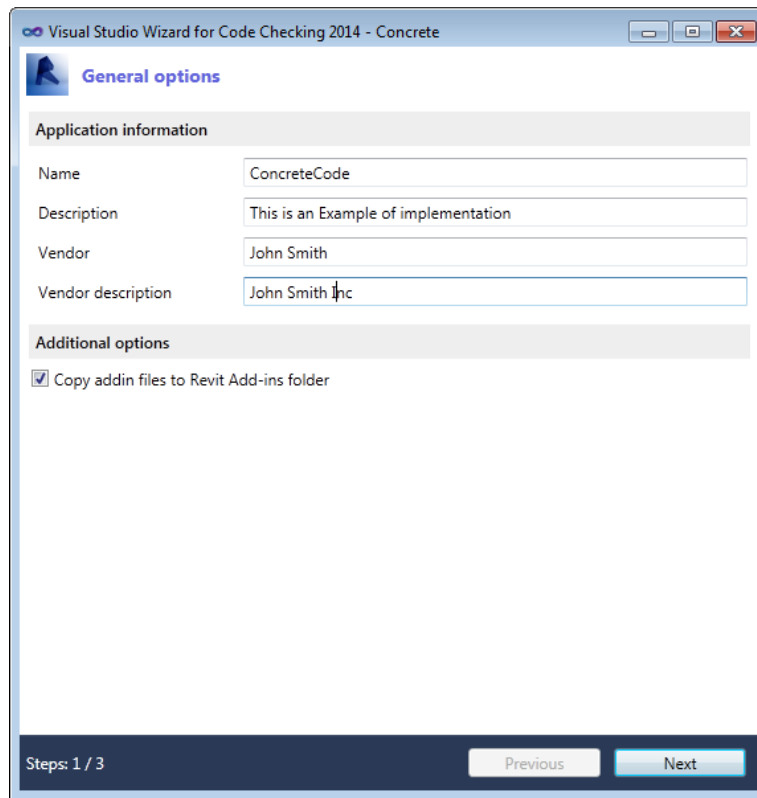


## 7 A NEW CODE CHECKING CONCRETE PROJECT

This example is based on the Code Checking Concrete template. For additional information, please refer to the “User Manual for Code Checking Framework SDK” document.

For this example, some correct foundations are needed, so we will create a new C# project using following options:

- First page



The screenshot shows the 'Visual Studio Wizard for Code Checking 2014 - Concrete' window. The title bar includes the Visual Studio icon and window controls. The main area is titled 'General options' and contains two sections: 'Application information' and 'Additional options'. Under 'Application information', there are four text input fields: 'Name' (containing 'ConcreteCode'), 'Description' (containing 'This is an Example of implementation'), 'Vendor' (containing 'John Smith'), and 'Vendor description' (containing 'John Smith Inc'). Under 'Additional options', there is a checked checkbox labeled 'Copy addin files to Revit Add-ins folder'. At the bottom, a dark blue bar shows 'Steps: 1 / 3' and two buttons: 'Previous' and 'Next'.

Name, Description, Vendor and Vendor description could be of course modified. After filling these fields, we can go to the next page.

- Second page

On this page all options should be checked and the Metric system of units should be chosen according following screenshot:



The screenshot shows a window titled "Visual Studio Wizard for Code Checking 2015 - Concrete". The main heading is "Code checking options".

**Supported material**

- ☒ Concrete

**Supported categories**

- ☒ Column
- ☒ Beam
- ☒ Floor
- ☒ FoundationSlab
- ☒ Wall

**Supported features**

- ☒ Multi server
- ☒ Updater

**Name**

**Description**

**System of units**

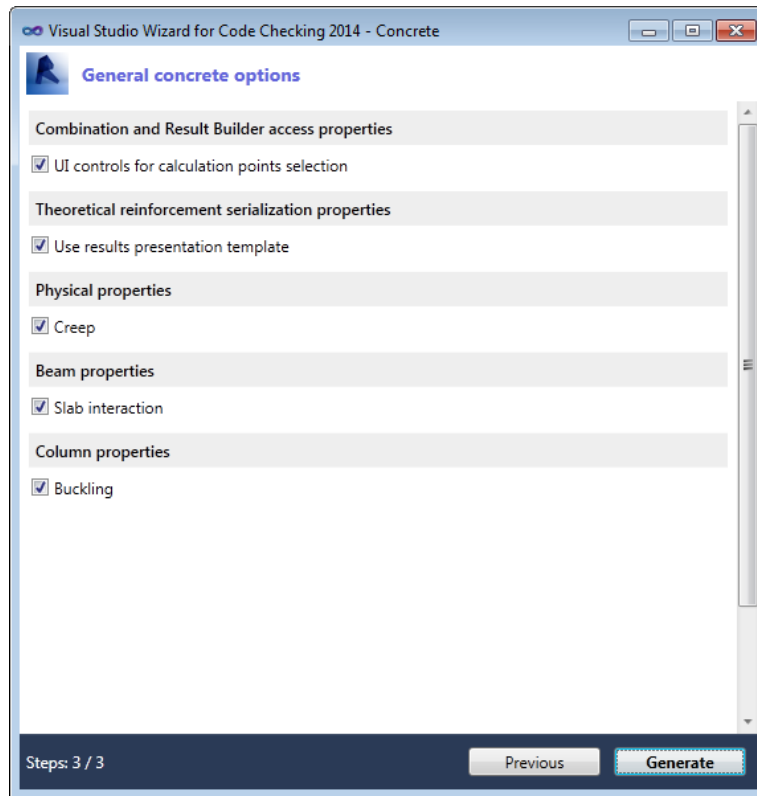
- ☒ Metric
- ☐ Imperial

Steps: 2 / 3

Previous Next

- Third page:

On the next page all options should be checked as well:



At this stage, we can use the option “Generate” and start working on the project created.

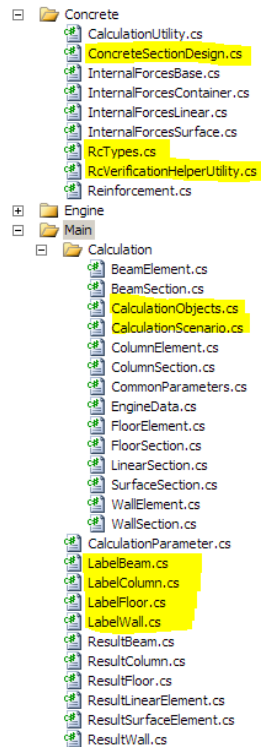
## 8 NOTATION AND REMARKS

Inside this document following convention are introduced for code regions:

- **part of code** (text with gray background): Existing code, shown only in order to identify place for adding new code
- **part of code** (text with green background): Existing code that should be removed for this example
- **part of code** (text without background): new code that should be placed in the example

To create this example, modifications that we will have to do are in 9 files:

- RCTypes.cs located here: “..\Concrete\RCTypes.cs”
- LabelBeam.cs located here: “..\Main\LabelBeam.cs”
- LabelColumn.cs located here: “..\Main\LabelColumn.cs”
- LabelFloor.cs located here: “..\Main\LabelFloor.cs”
- LabelWall.cs located here: “..\Main\LabelWall.cs”
- CalculationObjects.cs located here: “..\Main\Calculation\CalculationObjects.cs”
- CalculationScenario.cs located here: “..\Main\Calculation\CalculationScenario.cs”
- ConcreteSectionDesign.cs located here: “..\Concrete\ConcreteSectionDesign.cs”
- RcVerificationHelperUtility.cs located here: “..\Concrete\RcVerificationHelperUtility.cs”



## 9 ELEMENT DESIGN

As starting point, some modifications related to the calculation of elements should be done. For this example we will introduce some limitations that should be reflected into our code. Next, we will add some additional functionality to achieve the goals of this example.

### 9.1 RCTypes class

Class: RCTypes

From reinforcement concrete design perspective, in some cases, calculation can be simplified and made in an easier way than for other materials.

In the namespace `ConcreteTypes` in `RCTypes.cs` file, we will add new enumerator to identify these cases.

Code region

..\Concrete\RCTypes.cs file.

```

public enum CalculationType
{
    Unspecified = 0,
    LongAll = EnabledInternalForces.FX | EnabledInternalForces.MY | EnabledInternalForces.MZ,
    BendingY = EnabledInternalForces.MY,
    EccentricBendingY = EnabledInternalForces.FX | EnabledInternalForces.MY,
    AxialForce = EnabledInternalForces.FX,
    TransAll = EnabledInternalForces.FY | EnabledInternalForces.FZ | EnabledInternalForces.MX,
    ShearingZ = EnabledInternalForces.FZ,
    Torsion = EnabledInternalForces.MX,
    TorsionWithShearingZ = EnabledInternalForces.FZ | EnabledInternalForces.MX,
}

```

In the same file (RCTypes.cs) we should add 2 methods inside the class CalculationTypeHelper to facilitate the usage of this enumerator.

Code region

..\Concrete\RCTypes.cs

```

public static CalculationType GetLongitudinalCalculationType(
    this IEnumerable<EnabledInternalForces> enabledInternalForces)
{
    int val = 0;
    foreach (CalculationType calculationType in enabledInternalForces){
        if ((calculationType & CalculationType.LongAll) != 0){
            val |= (int)calculationType;
        }
    }
    return (CalculationType)val;
}

public static CalculationType GetTransversalCalculationType(
    this IEnumerable<EnabledInternalForces> enabledInternalForces)
{
    int val = 0;
    foreach (CalculationType calculationType in enabledInternalForces){
        if ((calculationType & CalculationType.TransAll) != 0){
            val |= (int)calculationType;
        }
    }
    return (CalculationType)val;
}

```

## 9.2 LabelColumn class

Class: LabelColumn

Appropriate properties for the new enumerator defined in RCTypes should be added to the column label as follow:

Code region

..\Main\LabelColumn.cs file.

```

public ConcreteTypes.CalculationType TransversalCalculationType { get {
    return EnabledInternalForces.GetTransversalCalculationType(); } }
public ConcreteTypes.CalculationType LongitudinalCalculationType { get {
    return EnabledInternalForces.GetLongitudinalCalculationType(); } }

```

## 9.3 LabelBeam class

Class: LabelBeam

As this example won't support biaxial bending and biaxial shearing for beams, we should modify the default beam label.

For this reason we create the new enumerator to describe forces which are not taken into account in the beam design (based on general enum *EnabledInternalForces*)

This enumerator will be added inside “..\Main\LabelBeam.cs” file.

Code region

..\Main\LabelBeam.cs file.

```

public enum EnabledInternalForcesForBeam
{
    FX = ConcreteTypes.EnabledInternalForces.FX,
    FZ = ConcreteTypes.EnabledInternalForces.FZ,
    MX = ConcreteTypes.EnabledInternalForces.MX,
    MY = ConcreteTypes.EnabledInternalForces.MY,
}

```

Also, we need to replace the existing label control *EnabledInternalForces* by the new created enumerator:

Code region

..\Main\LabelBeam.cs file.

```

[Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.SchemaProperty(
    FieldName = "BeamCalculationType")]
[EnumControl(
    Description = "EnabledInternalForces",
    Category = "CalculationOptions",
    EnumType = typeof(ConcreteTypes.EnabledInternalForces),
    Presentation = PresentationMode.OptionList,
    Item = PresentationItem.ImageWithText,
    ImageSize = ImageSize.Medium,
    Context = "BeamLabel")]
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.Attributes.ListValue(
    Name = "EnabledInternalForces",
    Localizable = true,
    LocalizableValue = true)]
public List<ConcreteTypes.EnabledInternalForces> EnabledInternalForces { get; set; }

```

And finally we will obtain:

Code region

..\Main\LabelBeam.cs file.

```
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.SchemaProperty(
    FieldName = "BeamCalculationType")]
[EnumControl(Description = "EnabledInternalForces",
    Category = "CalculationOptions",
    EnumType = typeof(EnabledInternalForcesForBeam),
    Presentation = PresentationMode.OptionList,
    Item = PresentationItem.ImageWithText,
    ImageSize = ImageSize.Medium,
    Context = "BeamLabel")]
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.Attributes.ListValue(
    Name = "EnabledInternalForces",
    Localizable = true,
    LocalizableValue = true)]
public List<EnabledInternalForcesForBeam> EnabledInternalForcesBeam { get; set; }
```

The standard constructor should be as well modified:

Code region

..\Main\LabelBeam.cs file.

```
public LabelBeam()
{
    LongitudinalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    CreepCoefficient = 2.0;
    TransversalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    EnabledInternalForces = new List<ConcreteTypes.EnabledInternalForces>();
    EnabledInternalForces.Add(ConcreteTypes.EnabledInternalForces.MY);
    EnabledInternalForces.Add(ConcreteTypes.EnabledInternalForces.FZ);
    SlabBeamInteraction = ConcreteTypes.BeamSectionType.WithSlabBeamInteraction;
}
```

To finally obtain:

Code region

..\Main\LabelBeam.cs file.

```
public LabelBeam()
{
    LongitudinalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    CreepCoefficient = 2.0;
    TransversalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    EnabledInternalForcesBeam = new List<EnabledInternalForcesForBeam>();
    EnabledInternalForcesBeam.Add(EnabledInternalForcesForBeam.MY);
    EnabledInternalForcesBeam.Add(EnabledInternalForcesForBeam.FZ);
    SlabBeamInteraction = ConcreteTypes.BeamSectionType.WithSlabBeamInteraction;
}
```

As for columns additional properties according to new enumerator defined in the RCTypes should be added in this label:

Code region

..\Main\LabelBeam.cs file.

```
public ConcreteTypes.CalculationType TransversalCalculationType { get {
    return EnabledInternalForces.GetTransversalCalculationType(); } }
public ConcreteTypes.CalculationType LongitudinalCalculationType { get {
    return EnabledInternalForces.GetLongitudinalCalculationType(); } }
```

Code region

..\Main\LabelBeam.cs file.

```
public List<ConcreteTypes.EnabledInternalForces> EnabledInternalForces{
    get
    {
        List<ConcreteTypes.EnabledInternalForces> EInternalForces = new
List<ConcreteTypes.EnabledInternalForces>();
        foreach (EnabledInternalForcesForBeam enabledInternalForcesBeam in EnabledInternalForcesBeam)
        {
            EInternalForces.Add((EnabledInternalForces)enabledInternalForcesBeam);
        }
        return EInternalForces;
    }
}
```

## 9.4 LabelFloor class

Class: LabelFloor

For this example Floors and Slabs foundation are treated as the same type of objects. For this reason the LabelFloor class is common for these objects.

As this example won't support bending in the plain of floor (including biaxial bending) and any type of shearing and torsion for floor and slab foundation, we should modify the default beam label.

For this reason we create the new enumerator to describe forces which are not taken into account in the beam design (based on general enum *EnabledInternalForces*)

This enumerator will be added inside “..\Main\LabelFloor.cs” file.

Code region - New Version 2015

..\Main\LabelFloor.cs file.

```
public enum EnabledInternalForcesForFloor
{
    FX = ConcreteTypes.EnabledInternalForces.FX,
    MY = ConcreteTypes.EnabledInternalForces.MY,
}
```

Also, we need to replace the existing label control *EnabledInternalForces* by the new created enumerator:

Code region - New Version 2015

..\Main\LabelFloor.cs file.

```

[Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.SchemaProperty(
    FieldName = "FloorCalculationType")]
[EnumControl(
    Description = "EnabledInternalForces",
    Category = "CalculationOptions",
    EnumType = typeof(ConcreteTypes.EnabledInternalForces),
    Presentation = PresentationMode.OptionList,
    Item = PresentationItem.ImageWithText,
    ImageSize = ImageSize.Medium,
    Context = "FloorLabel")]
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.Attributes.ListValue(
    Name = "EnabledInternalForces",
    Localizable = true,
    LocalizableValue = true)]
public List<ConcreteTypes.EnabledInternalForces> EnabledInternalForces { get; set; }

```

And finally we will obtain:

Code region - New Version 2015

..\Main\LabelFloor.cs file.

```

[Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.SchemaProperty(
    FieldName = "FloorCalculationType")]
[EnumControl(Description = "EnabledInternalForces",
    Category = "CalculationOptions",
    EnumType = typeof(EnabledInternalForcesForFloor),
    Presentation = PresentationMode.OptionList,
    Item = PresentationItem.ImageWithText,
    ImageSize = ImageSize.Medium,
    Context = "FloorLabel")]
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.Attributes.ListValue(
    Name = "EnabledInternalForces",
    Localizable = true,
    LocalizableValue = true)]
public List<EnabledInternalForcesForFloor> EnabledInternalForcesFloor { get; set; }

```

The standard constructor should be as well modified:

Code region - New Version 2015

..\Main\LabelFloor.cs file.

```

public LabelFloor()
{
    PrimaryReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    SecondaryReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    CreepCoefficient = 2.0;
    EnabledInternalForces = new List<ConcreteTypes.EnabledInternalForces>();
    EnabledInternalForces.Add(ConcreteTypes.EnabledInternalForces.MY);
}

```

To finally obtain:



Code region - New Version 2015

..\Main\LabelFloor.cs file.

```

public LabelFloor()
{
    PrimaryReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    SecondaryReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    CreepCoefficient = 2.0;
    EnabledInternalForcesFloor = new List<EnabledInternalForcesForFloor>();
    EnabledInternalForcesFloor.Add(EnabledInternalForcesForFloor.MY);
}

```

As for columns additional properties according to new enumerator defined in the RCTypes should be added in this label:

Code region - New Version 2015

..\Main\LabelFloor.cs file.

```

public ConcreteTypes.CalculationType LongitudinalCalculationType {
    get {
        return EnabledInternalForces.GetLongitudinalCalculationType();
    }
}

```

Code region - New Version 2015

..\Main\LabelFloor.cs file.

```

public List<ConcreteTypes.EnabledInternalForces> EnabledInternalForces{
    get{
        List<ConcreteTypes.EnabledInternalForces> EInternalForces =
            new List<ConcreteTypes.EnabledInternalForces>();
        foreach (EnabledInternalForcesForFloor enabledInternalForcesFloor in
            EnabledInternalForcesFloor){
            EInternalForces.Add((ConcreteTypes.EnabledInternalForces)enabledInternalForcesFloor);
        }
        return EInternalForces;
    }
}

```

## 9.5 LabelWall class

Class: LabelWall

As this example support only axial forces for walls, we could modify the default wall label. For this purpose, we create the new enumerator to describe forces which are not taken into account during the wall design (based on general enum *EnabledInternalForces*)

This enumerator will be added inside “..\Main\LabelWall.cs” file.

Code region - New Version 2015  
 ..\Main\LabelWall.cs file.

```
public enum EnabledInternalForcesForWall
{
    FX = ConcreteTypes.EnabledInternalForces.FX,
}
```

Also, we need to replace the existing label control *EnabledInternalForces* by the new created enumerator:

Code region - New Version 2015  
 ..\Main\LabelWall.cs file.

```
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.SchemaProperty(
    FieldName = "WallCalculationType")]
[EnumControl(
    Description = "EnabledInternalForces",
    Category = "CalculationOptions",
    EnumType = typeof(ConcreteTypes.EnabledInternalForces),
    Presentation = PresentationMode.OptionList,
    Item = PresentationItem.ImageWithText,
    ImageSize = ImageSize.Medium,
    Context = "WallLabel")]
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.Attributes.ListValue(
    Name = "EnabledInternalForces",
    Localizable = true,
    LocalizableValue = true)]
public List<ConcreteTypes.EnabledInternalForces> EnabledInternalForces { get; set; }
```

And finally we will obtain:

Code region - New Version 2015  
 ..\Main\LabelWall.cs file.

```
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Attributes.SchemaProperty(
    FieldName = "WallCalculationType")]
[EnumControl(Description = "EnabledInternalForces",
    Category = "CalculationOptions",
    EnumType = typeof(EnabledInternalForcesForWall),
    Presentation = PresentationMode.OptionList,
    Item = PresentationItem.ImageWithText,
    ImageSize = ImageSize.Medium,
    Context = "WallLabel")]
[Autodesk.Revit.DB.ExtensibleStorage.Framework.Documentation.Attributes.ListValue(
    Name = "EnabledInternalForces",
    Localizable = true,
    LocalizableValue = true)]
public List<EnabledInternalForcesForWall> EnabledInternalForcesWall { get; set; }
```

The standard constructor should be as well modified:

Code region - New Version 2015  
 ..\Main\LabelWall.cs file.

```
public LabelWall()
{
    VerticalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    HorizontalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    CreepCoefficient = 2.0;
    EnabledInternalForces = new List<ConcreteTypes.EnabledInternalForces>();
    EnabledInternalForces.Add(ConcreteTypes.EnabledInternalForces.FZ);
}
```

To finally obtain:

Code region - New Version 2015  
 ..\Main\LabelWall.cs file.

```
public LabelWall()
{
    VerticalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    HorizontalReinforcement =
        new UIComponents.RCSteelParameters.RCSteelParametersSchema();
    CreepCoefficient = 2.0;
    EnabledInternalForcesWall = new List<EnabledInternalForcesForWall>();
    EnabledInternalForcesWall.Add(EnabledInternalForcesForWall.FX);
}
```

As for columns additional properties according to new enumerator defined in the RCTypes should be added in this label:

Code region - New Version 2015  
 ..\Main\LabelWall.cs file.

```
public ConcreteTypes.CalculationType LongitudinalCalculationType{ get {
    return EnabledInternalForces.GetLongitudinalCalculationType (); } }
```

Code region - New Version 2015  
 ..\Main\LabelWall.cs file.

```
public List<ConcreteTypes.EnabledInternalForces> EnabledInternalForces{
    get{
        List<ConcreteTypes.EnabledInternalForces> EInternalForces =
            new List<ConcreteTypes.EnabledInternalForces>();
        foreach (EnabledInternalForcesForWall enabledInternalForcesWall in EnabledInternalForcesWall){
            EInternalForces.Add((ConcreteTypes.EnabledInternalForces)enabledInternalForcesWall);
        }
        return EInternalForces;
    }
}
```

## 9.6 Deflection calculation

Class: CalculateDeflection

To calculate deflection, we will create a new class `CalculateDeflection` which inherits from `ICalculationObject`. The deflection calculation results of multiplication the statistical deformation by a coefficient. This coefficient takes into account the stiffness of reinforced concrete cross sections including cracking and creep. In this example, the final coefficient is the average value of the average and maximum values for each section.

To achieve this, we should add a new class in the `CalculationObjects.cs` with a method `Run()` and create new properties as follow:

Code region

..\Main\Calculation\CalculationObjects.cs

```

public class CalculateDeflection : ICalculationObject{
    #region ICalculationObject Members
    public bool Run(ObjectDataBase obj){
        ElementDataBase elementData = obj as ElementDataBase;
        if (obj != null){
            BeamElement objBeam = obj as BeamElement;
            if (objBeam != null){
                Main.ResultBeam res =
                    elementData.Result as Main.ResultBeam;
                if (res != null){
                    double young = objBeam.Info.Material.Characteristics.YoungModulus.X;
                    double maxCoef = 0, stifCoef = 0.0, avrCoef = 0.0;
                    int sectionNo = 0;
                    foreach (SectionDataBase sd in elementData.ListSectionData){
                        BeamSection sec = sd as BeamSection;
                        if (sec != null){
                            if (sec.MinStiffness > Double.Epsilon){
                                CalcPointLinear calcPoint = sec.CalcPoint as CalcPointLinear;
                                double Iy =
                                    sec.Info.SectionsParams.AtThePoint(calcPoint.CoordRelative).Characteristics.Iy;
                                stifCoef = (Iy * young) / sec.MinStiffness;
                            }
                            else{
                                stifCoef = 0;
                                break;
                            }
                            ++sectionNo;
                            maxCoef = Math.Max(maxCoef, stifCoef);
                            avrCoef += stifCoef;
                        }
                    }
                    double finStiffnes = 0;
                    if (stifCoef > Double.Epsilon){
                        avrCoef /= sectionNo;
                        finStiffnes = 0.5 * (maxCoef + avrCoef);
                    }
                    else
                        objBeam.AddFormattedWarning(
                            new ResultStatusMessage("Calculation of deflections was not performed."));
                    foreach (SectionDataBase sd in elementData.ListSectionData){
                        BeamSection sec = sd as BeamSection;
                        if (sec != null)
                            sec.StiffnesCoeff = finStiffnes;
                    }
                }
            }
        }
        return true;
    }
    public CommonParametersBase Parameters { get; set; }
    public CalculationObjectType Type { get; set; }
    public ErrorResponse ErrorResponse { get; set; }
    public IList<Autodesk.Revit.DB.BuiltInCategory> Categories { get; set; }
    #endregion
}

```

## 9.7 Modification of input forces

Class: ModifyElementForces

This example shows how to modify the input forces due to buckling effect and supports condition. To do this, we should modify the method `Run()` in the `ModifyElementForces` class. We are going to create a method to modify a set of forces separately for beam and column.

As all national codes give specific requirements for buckling, the template includes only the signature of an empty method.

This method is called `ModifyForcesBecauseOfBuckling` and we will modify it.

Code region

..\Main\Calculation\CalculationObjects.cs

```
private void ModifyForcesBecauseOfBuckling(ObjectDataBase obj)
```

In this example, the buckling effect increase moments. New moments are calculated according to axial forces, element length and buckling coefficient.

## Code region

## ..\Main\Calculation\CalculationObjects.cs

```

private void ModifyForcesBecauseOfBuckling(ObjectDataBase obj){
    ColumnElement elem = obj as ColumnElement;
    Main.LabelColumn rclabelColumn = elem.Label as
                                                Main.LabelColumn;

    bool dirY = rclabelColumn.EnabledInternalForces.Contains(ConcreteTypes.EnabledInternalForces.MY);
    bool dirZ = rclabelColumn.EnabledInternalForces.Contains(ConcreteTypes.EnabledInternalForces.MZ);
    dirY &= rclabelColumn.BucklingDirectionY;
    dirZ &= rclabelColumn.BucklingDirectionZ;
    if (dirY || dirZ){
        double lambdaLim = 15.0, lambdaY = 0, lambdaZ = 0, length = elem.Info.GeomLength();
        bool modifMy = false, modifMz = false;
        if (dirY){
            lambdaY = (length * rclabelColumn.LengthCoefficientY) /
                        elem.Info.SectionsParams.AtTheBeg.Characteristics.rY;
            modifMy = lambdaY > lambdaLim;
        }
        if (dirZ) {
            lambdaZ = (length * rclabelColumn.LengthCoefficientZ) /
                        elem.Info.SectionsParams.AtTheBeg.Characteristics.rZ;
            modifMz = lambdaZ > lambdaLim;
        }
        double additionalMy = 0, additionalMz = 0, My = 0, Mz = 0, N = 0;
        double dimY = elem.Info.SectionsParams.AtTheBeg.Dimensions.vpy
                        + elem.Info.SectionsParams.AtTheBeg.Dimensions.vy;
        double dimZ = elem.Info.SectionsParams.AtTheBeg.Dimensions.vpz
                        + elem.Info.SectionsParams.AtTheBeg.Dimensions.vz;
        foreach (SectionDataBase sd in elem.ListSectionData){
            ColumnSection sec = sd as ColumnSection;
            if (sec != null){
                foreach (InternalForcesBase forces in sec.ListInternalForces){
                    InternalForcesLinear f = forces as InternalForcesLinear;
                    if (f != null){
                        if (f.Forces.LimitState == ForceLimitState.Uls){
                            CalcPointLinear calcPoint = sec.CalcPoint as CalcPointLinear;
                            N = f.Forces.ForceFx;
                            if (N > Double.Epsilon){
                                if (dirY){
                                    My = f.Forces.MomentMy;
                                    additionalMy = (-0.002 * N / dimZ);
                                    additionalMy *= rclabelColumn.LengthCoefficientY *
                                                    rclabelColumn.LengthCoefficientY;
                                    additionalMy *= (calcPoint.CoordAbsolute * calcPoint.CoordAbsolute -
                                                    length * calcPoint.CoordAbsolute);
                                    f.Forces.MomentMy += f.Forces.MomentMy > 0.0 ?
                                                            additionalMy : -additionalMy;
                                }
                                if (dirZ){
                                    Mz = f.Forces.MomentMz;
                                    additionalMz = (-0.002 * N / dimY);
                                    additionalMz *= rclabelColumn.LengthCoefficientZ *
                                                    rclabelColumn.LengthCoefficientZ;
                                    additionalMz *= (calcPoint.CoordAbsolute * calcPoint.CoordAbsolute -
                                                    length * calcPoint.CoordAbsolute);
                                    f.Forces.MomentMz += f.Forces.MomentMz > 0.0 ?
                                                            additionalMz : -additionalMz;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

Some national standard recommends modifying the bending moment also for the beam. In our case, to support this requirement, we add a simple rule to modify moments over beam supports. We will set the moment value over beam supports as not less than 10% of the maximum moment.

Inside the same class (`ModifyElementForces`), we should add a method to modify moments at beam ends as follow:

Code region - New Version 2015

..\Main\Calculation\CalculationObjects.cs

```

/// <structural_toolkit_2015>
private void ModifySupportForces(ObjectDataBase obj){
    BeamElement elem = obj as BeamElement;
    if (elem != null && elem.ListSectionData.Count() > 0){
        double maximumM =
            elem.ListSectionData.Max(sd => ((sd as BeamSection).ListInternalForces).Max(
                f => (f as InternalForcesLinear).Forces.MomentMy));

        bool isMaximumM = (maximumM > 0.0);
        if (isMaximumM){
            List<int> indexSupport = new List<int>();
            double maximumL = elem.ListSectionData.Max(
                sd => ((sd as BeamSection).CalcPoint as CalcPointLinear).CoordRelative);
            indexSupport.Add(elem.ListSectionData.FindIndex(
                sd => ((sd as BeamSection).CalcPoint as CalcPointLinear).CoordRelative.Equals(maximumL)));
            double minimumL = elem.ListSectionData.Min(
                sd => ((sd as BeamSection).CalcPoint as CalcPointLinear).CoordRelative);
            indexSupport.Add(elem.ListSectionData.FindIndex(
                sd => ((sd as BeamSection).CalcPoint as CalcPointLinear).CoordRelative.Equals(minimumL)));
            foreach (int i in indexSupport){
                if (i > -1){
                    BeamSection bs = elem.ListSectionData[i] as BeamSection;
                    if (bs != null){
                        if (bs.ListInternalForces != null){
                            IEnumerable<InternalForcesBase> ULSForces = bs.ListInternalForces.Where(
                                f => (f as InternalForcesLinear).Forces.LimitState == ForceLimitState.Uls);
                            if (ULSForces.Count() > 0){
                                double fVMax = ULSForces.Max(
                                    fUls => Math.Abs((fUls as InternalForcesLinear).Forces.ForceFz));
                                if (CalculationUtility.IsZeroN(fVMax))
                                    continue;
                                double Mmin = ULSForces.Min(
                                    fUls => (fUls as InternalForcesLinear).Forces.MomentMy);
                                double MAdd = -0.1 * maximumM;
                                if (Mmin > MAdd)
                                {
                                    double Mmax = Mmin;
                                    if (bs.ListInternalForces.Count != 1)
                                        Mmax = ULSForces.Max(
                                            fUls => (fUls as InternalForcesLinear).Forces.MomentMy);
                                    InternalForcesLinear forces = bs.ListInternalForces.Find(
                                        f => (f as InternalForcesLinear).Forces.MomentMy.Equals(Mmax))
                                        as InternalForcesLinear;

                                    forces.Forces.MomentMy = MAdd;
                                    bs.ListInternalForces.Add(forces);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
/// </structural_toolkit_2015>

```

When this is done, a call to this new `ModifySupportForces()` method should be added to the method `Run()` from the `ModifyElementForces` class.



Code region - Updated Version 2015

..\Main\Calculation\CalculationObjects.cs

```

public bool Run(ObjectDataBase obj)
{
    bool isOk = true;
    if (obj != null)
    {
        switch (obj.Category)
        {
            default:
                isOk = false;
                break;
            /// <structural_toolkit_2015>
            case Autodesk.Revit.DB.BuiltInCategory.OST_FloorAnalytical:
            case Autodesk.Revit.DB.BuiltInCategory.OST_FoundationSlabAnalytical:
            case Autodesk.Revit.DB.BuiltInCategory.OST_WallAnalytical:
                {
                    isOk = true;
                }
                break;
            /// </structural_toolkit_2015>
            case Autodesk.Revit.DB.BuiltInCategory.OST_ColumnAnalytical:
                {
                    ModifyForcesBecauseOfBuckling(obj);
                }
                break;
            case Autodesk.Revit.DB.BuiltInCategory.OST_BeamAnalytical:
                {
                    ModifySupportForces(obj);
                }
                break;
        }
    }
    return isOk;
}

```

## 9.8 Modification of design process scenario

Class: CalculationScenarioList

To use the new calculation process we have created just before, we must now add it into the calculations scenario.

The calculation scenario could be found in the CalculationScenario.cs file and the main method to look at is CalculationScenarioList(). To support the deflection calculation, we should create a new CalculateDeflection object and add to the list. As calculation form CalculateDeflection are based on section calculation, this new calculation object should be added after CalculateSection and before FillResultData sections:

## Code region - Updated Version 2015

## ..\Main\Calculation\CalculationScenario.cs

```

public List<ICalculationObject> CalculationScenarioList(Autodesk.Revit.DB.BuiltInCategory category,
Autodesk.Revit.DB.StructuralAssetClass material){
    List<ICalculationObject> scenario = new List<ICalculationObject>();
    /// <structural_toolkit_2015>
    switch (material){
        case StructuralAssetClass.Concrete:
            switch (category){
                case BuiltInCategory.OST_BeamAnalytical:
                case BuiltInCategory.OST_ColumnAnalytical:
                    {PrepareSectionData calcObj = new PrepareSectionData();
                    calcObj.Type = CalculationObjectType.Section;
                    calcObj.ErrorResponse = ErrorResponse.SkipOnError;
                    calcObj.Categories = new List<BuiltInCategory>()
                    { BuiltInCategory.OST_BeamAnalytical, BuiltInCategory.OST_ColumnAnalytical };
                    scenario.Add(calcObj);}
                    {ModifyElementForces calcObj = new ModifyElementForces();
                    calcObj.Type = CalculationObjectType.Element;
                    calcObj.ErrorResponse = ErrorResponse.SkipOnError;
                    calcObj.Categories = new List<BuiltInCategory>()
                    { BuiltInCategory.OST_BeamAnalytical, BuiltInCategory.OST_ColumnAnalytical };
                    scenario.Add(calcObj);}
                    {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); }
                    break;
                case Autodesk.Revit.DB.BuiltInCategory.OST_FloorAnalytical:
                case Autodesk.Revit.DB.BuiltInCategory.OST_FoundationSlabAnalytical:
                case Autodesk.Revit.DB.BuiltInCategory.OST_WallAnalytical:
                    {PrepareSectionData calcObj = new PrepareSectionData();
                    calcObj.Type = CalculationObjectType.Section;
                    calcObj.ErrorResponse = ErrorResponse.SkipOnError;
                    calcObj.Categories = new List<BuiltInCategory>()
                    { BuiltInCategory.OST_FloorAnalytical, BuiltInCategory.OST_FoundationSlabAnalytical,
                    BuiltInCategory.OST_WallAnalytical };
                    scenario.Add(calcObj);}
                    {ModifyElementForces calcObj = new ModifyElementForces();
                    calcObj.Type = CalculationObjectType.Element;
                    calcObj.ErrorResponse = ErrorResponse.SkipOnError;
                    calcObj.Categories = new List<BuiltInCategory>()
                    { BuiltInCategory.OST_FloorAnalytical, BuiltInCategory.OST_FoundationSlabAnalytical,
                    BuiltInCategory.OST_WallAnalytical };
                    scenario.Add(calcObj); }
                    {CalculateSection calcObj = new CalculateSection();
                    calcObj.Type = CalculationObjectType.Section;
                    calcObj.ErrorResponse = ErrorResponse.SkipOnError;
                    calcObj.Categories = new List<BuiltInCategory>()
                    { BuiltInCategory.OST_FloorAnalytical, BuiltInCategory.OST_FoundationSlabAnalytical,
                    BuiltInCategory.OST_WallAnalytical };
                    scenario.Add(calcObj);}
                    {FillResultData calcObj = new FillResultData();
                    calcObj.Type = CalculationObjectType.Element;
                    calcObj.ErrorResponse = ErrorResponse.RunOnError;
                    calcObj.Categories = new List<BuiltInCategory>()
                    { BuiltInCategory.OST_FloorAnalytical, BuiltInCategory.OST_FoundationSlabAnalytical,
                    BuiltInCategory.OST_WallAnalytical };
                    scenario.Add(calcObj);}
                    break;
            }
        }
    }
    /// </structural_toolkit_2015>
    return scenario;
}

```

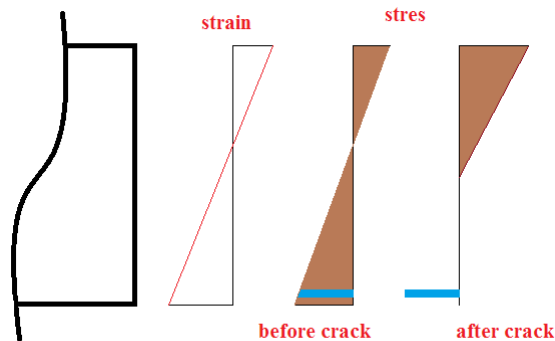
## 10 CRACKING AND STIFFNESS

Class: `RcVerificationHelperUtility`

To take into account cracked section during the design process, we could add 2 additional methods at the end of into the `RcVerificationHelperUtility` class.

This calculation is correct only for typical SLS concrete and steel behaviors:

- Steel works in his elastic limits
- Concrete has linear stress-strain relationship until crack
- After crack point, the concrete works only for compression
- After crack point, the concrete have linear stress-strain relationship for compression



### 10.1 Moment of inertia for cracking section

Method: `InertiaOfCracingSection`

In `InertiaOfCracingSection ()` method, inertia of cracked section is calculated.

To perform this calculation, we will use the solver that is provided by Autodesk.CodeChecking library (namespace `Autodesk.CodeChecking.Concrete.RCSolver`).

Taking into account the acting forces we will calculate the position of the neutral axis.

Based on this position, the cross section geometry, the position of rebars and the Young Modulus of the steel and the concrete, the moment of inertia of the cracked section will be calculated.

Code region - Updated Version 2015

..\Concrete\RcVerificationHelperUtility.cs

```

/// <structural_toolkit_2015>
public double InertiaOfCrackingSection(InternalForcesContainer inNMM)
{
    double momentOfInertiaCrackingConcreteSection = 0.0;
    SafetyFactor(inNMM);
    SetOfForces solverNMM =
        solver.GetInternalForces(Autodesk.CodeChecking.Concrete.ResultType.Section);
    double neutralAxisDist = solver.GetNeutralAxisDistance();
    double stressArea = solver.GetConcreteStressArea();
    double comprHeight = 0.5 * totalHeight + neutralAxisDist;
    Steel steel = solver.GetSteel();
    Autodesk.CodeChecking.Concrete.Concrete concrete = solver.GetConcrete();
    double n = steel.ModulusOfElasticity / concrete.ModulusOfElasticity;
    switch (crossSectionType){
        case SectionShapeType.RectangularBar:
            momentOfInertiaCrackingConcreteSection = comprHeight * comprHeight * stressArea / 3.0;
            break;
        default:
            throw new Exception(
                "InertiaOfCrackingSection. Unhandled cross section type. Only rectangular cross-section can be used on
                this path. 3th party implementation is necessary.");
    }
    foreach (Rebar bar in solver.GetRebars())
    {
        momentOfInertiaCrackingConcreteSection +=
            n * bar.Area * Math.Pow((bar.Y + neutralAxisDist),2);
    }
    return momentOfInertiaCrackingConcreteSection;
}
/// </structural_toolkit_2015>

```

## 10.2 Ratio acting forces to forces giving the first cracking

Method: ForcesToCrackingForces

In this example, the coefficient acting forces to cracking forces (forces that give first crack) is used. To perform this calculation, we will use the solver that is provided by Autodesk.CodeChecking library (namespace Autodesk.CodeChecking.Concrete.RCSolver).

Using acting forces, the solver calculates the strain on the edge of the section and as result of calculation we compare the stress evoked by acting forces and the limit due to cracking.

Code region - Updated Version 2015

..\Concrete\RcVerificationHelperUtility.cs

```

/// <structural_toolkit_2015>
public double ForcesToCrackingForces(InternalForcesContainer inNMM, double crackingStress){
    double forcesToCrackigForces = 0;
    if (!CalculationUtility.IsZeroM(inNMM.MomentMz))
        throw new Exception("Deflection calculation is not aviable for biaxial bending.");
    double actingForcesStress = 0;
    if (!CalculationUtility.IsZeroM(inNMM.MomentMy)){
        double w = solverGeometry.MomentOfInertiaX;
        w /= inNMM.MomentMy > 0.0 ? (geometryMaxY - solverGeometry.CenterOfInertia.Y) :
            (solverGeometry.CenterOfInertia.Y - geometryMinY);
        actingForcesStress += Math.Abs(inNMM.MomentMy) / w;
    }
    if (!CalculationUtility.IsZeroN(inNMM.ForceFx))
        actingForcesStress += -inNMM.ForceFx / solverGeometry.Area;
    if (actingForcesStress >= 0)
        forcesToCrackigForces = actingForcesStress / crackingStress ;
    return forcesToCrackigForces;
}
}
/// </structural_toolkit_2015>

```

## 11 CROSS SECTION DESIGN

The core of theoretical reinforcement calculation is at the cross-section level. We add at this level most of the code dependent calculation.

To achieve this, we will define new calculation methods in the existing `ConcreteSectionDesign` class.

### 11.1 Class members

Class: `ConcreteSectionDesign`

In this example, the core of calculation is implemented in the `ConcreteSectionDesign` class. All below described methods are implemented in this class if not otherwise indicated.

Per default, this class included few private members:

- `listInternalForces`: list of internal forces sets
- `sectionGeometry`: detailed geometry description
- `sectionWidth`: height of cross section
- `sectionHight`: width of cross section
- `sectionType`: shape (type) of cross section
- `concreteFc`: compressive strength of concrete
- `concreteYoungMod`: modulus of elasticity of concrete (Young modulus)
- `concreteCreepCoef`: creep coefficient
- `transReinforcementArea`: area of one rebar used to transversal reinforcement
- `transReinforcementFy`: yield strength of transverse reinforcement
- `transReinforcementNoLegs`: number of legs in the one stirrups frame
- `longReinforcementArea`: area of one rebar used to longitudinal reinforcement
- `longReinforcementFy`: yield strength of transverse reinforcement

- `longReinforcementCoverTop`: cover from cross section top edge to center of top rebar
- `longReinforcementCoverBottom`: cover from cross section bottom edge to center of bottom rebar
- `longReinforcementTopClearCover`: cover from cross section top edge to the external edge of transversal reinforcement
- `longReinforcementBottomClearCover`: cover from cross section top edge to the external edge of transversal reinforcement
- `symmetricalReinforcementPreferable`: if the symmetrical reinforcement is more preferable (depends to internal forces) this flag is set on true
- `ElementType`: type of the element
- `transReinforcementCalculationType`: let developer choose what kind of calculation should be used
- `longReinforcementCalculationType`: let developer choose what kind of calculation should be used
- `concreteParameters`: detailed concrete description
- `verificationHelper`: package of tools to help during the cross section verification
- `longitudinalReinforcement`: Container for calculated areas of longitudinal reinforcement
- `transversalReinforcement`: Container for calculated areas of transversal reinforcement
- `minStiffness`: minimum cross section stiffness under action of internal forces, including cracking and creep
- `designInfo`: optional information about design process
- `designError`: information about error in design process
- `designWarning`: information about not critical problems in design process
- `dimensioningDirection`: information about direction (X or Y) for surface elements (plate, floor and wall)
- 

## 11.2 New Class members

To simplify calculation, this example is based on a reduced list of forces sets.

To support this approach, some addition members should be defined:

- `longReinforcementInternalForcesULS`: reduced list of ULS sets of internal forces used to design longitudinal reinforcement
- `longReinforcementInternalForcesSLS`: reduced list of SLS sets of internal forces used to design longitudinal reinforcement
- `transReinforcementInternalForcesULS`: reduced list of ULS sets of internal forces used to design transversal reinforcement

To minimize “switch” and “if” conditions, we could define two additional variables defining simple cases of design.

- `transReinforcementCalculationType`: type of calculation which is taken into account for transverse reinforcement design
- `longReinforcementCalculationType`: type of calculation which is taken into account for longitudinal reinforcement design

To define symmetrical reinforcement for the element:

- `symmetricalReinforcementPreferable`: will set as true for pure compression and tension and for elements where axial force dominate

We will add these new members at the beginning of the `ConcreteSectionDesign` class analogously to other existing variables:

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
private List<InternalForcesContainer> internalForces;
private List<InternalForcesContainer> longReinforcementInternalForcesULS;
private List<InternalForcesContainer> longReinforcementInternalForcesSLS;
private List<InternalForcesContainer> transReinforcementInternalForcesULS;

private ConcreteTypes.CalculationType transReinforcementCalculationType;
private ConcreteTypes.CalculationType longReinforcementCalculationType;
private bool symmetricalReinforcementPreferable;
```

We should also here initialize this data into the `ConcreteSectionDesign()` constructor:

Code region - Updated Version 2015

..\Concrete\ ConcreteSectionDesign.cs

```
public ConcreteSectionDesign()
{
    internalForces = new List<InternalForcesContainer>();
    sectionGeometry = new Geometry();
    sectionWidth = 0.0;
    sectionHeight = 0.0;
    sectionType = SectionShapeType.RectangularBar;
    concreteFc = 0.0;
    concreteYoungModulus = 0.0;
    concreteCreepCoefficient = 1.0;
    transReinforcementArea = 0.0;
    transReinforcementDiameter = 0.0;
    transReinforcementFy = 0.0;
    longReinforcementArea = 0.0;
    longReinforcementDiameter = 0.0;
    longReinforcementFy = 0.0;
    longReinforcementTopCover = 0.0;
    longReinforcementBottomCover = 0.0;
    longReinforcementTopClearCover = 0.0;
    longReinforcementBottomClearCover = 0.0;
    symmetricalReinforcementPreferable = false;
    elementType = Autodesk.Revit.DB.BuiltInCategory.INVALID;
    concreteParameters = new Autodesk.CodeChecking.Concrete.Concrete();
    minStiffness = 0.0;
    designInfo = new List<string>();
    designError = new List<string>();
    designWarning = new List<string>();
    longReinforcementInternalForcesULS = new List<InternalForcesContainer>();
    longReinforcementInternalForcesSLS = new List<InternalForcesContainer>();
    transReinforcementInternalForcesULS = new List<InternalForcesContainer>();
    transReinforcementCalculationType = ConcreteTypes.CalculationType.ShearingZ;
    longReinforcementCalculationType = ConcreteTypes.CalculationType.BendingY;
    symmetricalReinforcementPreferable = false;
    dimensioningDirection = ConcreteTypes.DimensioningDirection.X;
}
```

For `CalculationType`, we should add some setters:

Code region

..\Concrete\ConcreteSectionDesign.cs

```

public ConcreteTypes.CalculationType TransversalCalculationType
{
    set { transReinforcementCalculationType = value; }
}
public ConcreteTypes.CalculationType LongitudinalCalculationType
{
    set { longReinforcementCalculationType = value; }
}

```

We should use these setters in Run method in the CalculateSection class

Code region - Updated Version 2015

..\Main\Calculation\CalculationObjects.cs

```

(...)
case BuiltInCategory.OST_ColumnAnalytical:
case BuiltInCategory.OST_BeamAnalytical:
    if (sectionData.Label is LabelColumn){
        (...)
        design.TransversalReinforcementArea = label.TransversalReinforcement.Area;
        design.TransversalReinforcementDiameter = label.TransversalReinforcement.BarDiameter;
        design.CreepCoefficient = label.CreepCoefficient;
        design.TransversalCalculationType = label.TransversalCalculationType;
        design.LongitudinalCalculationType = label.LongitudinalCalculationType;
    }
    if (sectionData.Label is LabelBeam){
        (...)
        design.TransversalReinforcementArea = label.TransversalReinforcement.Area;
        design.TransversalReinforcementDiameter = label.TransversalReinforcement.BarDiameter;
        design.CreepCoefficient = label.CreepCoefficient;
        design.LongitudinalCalculationType = label.LongitudinalCalculationType;
        design.TransversalCalculationType = label.TransversalCalculationType;
    }
}

(...)
/// <structural_toolkit_2015>
case Autodesk.Revit.DB.BuiltInCategory.OST_FloorAnalytical:
case Autodesk.Revit.DB.BuiltInCategory.OST_FoundationSlabAnalytical:
case Autodesk.Revit.DB.BuiltInCategory.OST_WallAnalytical:
    (...)
    if( labelFloor!=null)
    {
        coverT[1] += labelFloor.PrimaryReinforcement.BarDiameter;
        coverB[1] += labelFloor.PrimaryReinforcement.BarDiameter;
        design.LongitudinalCalculationType = labelFloor.LongitudinalCalculationType;
        design.CreepCoefficient = labelFloor.CreepCoefficient;
    }
    if( labelWall!=null)
    {
        coverT[1] += labelWall.VerticalReinforcement.BarDiameter;
        coverB[1] += labelWall.VerticalReinforcement.BarDiameter;
        design.LongitudinalCalculationType = labelWall.LongitudinalCalculationType;
        design.CreepCoefficient = labelWall.CreepCoefficient;
    }
}

```

## 11.3 Main design method



Method: Calculate

In the method Calculate() part of based template, following code should be removed:

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
// After your implementation remove code below!
// This is only example
string warning = "This is only a sample code. It should be edited or replaced by a 3rd party implementation.";
designWarning.Add(warning);
longitudinalReinforcement.CurrentAsTop = 0.05;
longitudinalReinforcement.CurrentToFinial();
if (designWarning.First() != warning)
    throw new Exception(warning);
// After your implementation remove code above!
```

The Calculate() method we look like this:

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
public void Calculate(){
    try{
        PreparationOfCalculationData();
        DataVeryfication();
    }
    catch (Exception e) {
        OnError(e);
    }
}
```

At this stage, you should be able to compile without issue the project.

Two existing method to prepare data and to verify them should stay and we could add now some calls to some new methods for verification, design and calculations.

Calculation process is split into three parts:

- longitudinal reinforcement
- transversal reinforcement
- stiffness of cross section

For this kind of design, some specialized methods will be used.

For longitudinal and transverse reinforcement, basic calculation flow is this one:

- Reduction of internal forces (described previously)
- Calculate reinforcement
- Set minimal reinforcement according code requirements

At the end of this process, the stiffness is calculated based on the list of forces used to calculate the longitudinal reinforcement.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
public void Calculate(){
    try{
        PreparationOfCalculationData();
        DataVerification();
        ReduceInternalForcesForLongitudinalReinforcement();
        CalculateLongitudinalReinforcement();
        SetLongitudinalMinimumReinforcement();
        ReduceInternalForcesForTransversalReinforcement();
        CalculateTransversalReinforcement();
        SetTransversalMaximumStirupSpacing();
        CalculateStiffness();
    }
    catch (Exception e) {
        OnError(e);
    }
}
```

## 11.4 Verification of input data

Method: DataVerification

Before calculations the input data are verified and the exceptions could be thrown.

In this place limitations for set for this example will be checked (see limitations section of this document).

Beside existing verifications, we will a new set of limitations.

We will assume that following are unreasonable data:

- Cross section dimension is less than 1 mm (0.04 in)
- Covers is less than 0.1 mm (0.004 in) and bigger than cross section size reduced by 0.2 mm (0.008 in),
- Concrete properties:  $f_c < 1$  kPa (0.146 psi),  $E_c < 1$  MPa (0.146 ksi),
- Reinforcement steel properties:  $f_c < 10$  kPa (1.46 psi),
- Rebar area is less than 0.1mm<sup>2</sup> (0. 0.000155 in<sup>2</sup>)
- Reinforcement properties for transverse reinforcement are checked only for beams and columns

Code region - Updated Version 2015

..\Concrete\ ConcreteSectionDesign.cs

```

public void DataVerifyfication(){
    if (Math.Min(longReinforcementTopCover, longReinforcementBottomCover) < 1e-4)
        throw new Exception(Properties.Resources.ResourceManager.GetString("ErrCover"));
    if (Autodesk.Revit.DB.BuiltInCategory.OST_ColumnAnalytical == elementType){
        if (Math.Max(longReinforcementTopCover, longReinforcementBottomCover)
            > 0.5 * Math.Min(sectionHeight, sectionWidth))
            throw new Exception(Properties.Resources.ResourceManager.GetString("ErrCover"));
    }
    else{
        if (longReinforcementTopCover + longReinforcementBottomCover > sectionHeight - 2e-4)
            throw new Exception(Properties.Resources.ResourceManager.GetString("ErrCover"));
    }
    switch(sectionType){
        default:
            throw
                new Exception(Properties.Resources.ResourceManager.GetString("ErrSectionNotSupported"));
        case SectionShapeType.T:
            break;
        case SectionShapeType.RectangularBar:
            break;
    }

    if (Math.Min(sectionWidth, sectionHeight) < 1e-3)
        throw new Exception("Types: Section dimensions are not properly defined.");
    if (concreteYoungModulus < 1e6)
        throw new Exception(Properties.Resources.ResourceManager.GetString("ErrYoungModulus"));
    if (concreteFc < 1e3)
        throw new Exception(Properties.Resources.ResourceManager.GetString("ErrConcreteCompression"));
    if (concreteCreepCoefficient < 1e-3)
        throw new Exception
            ("Element Settings: Invalid creep coefficient.Creep coefficient must be greater than zero.");
    /// <structural_toolkit_2015>
    if (longReinforcementFy < 1e4)
        throw new Exception(
            Properties.Resources.ResourceManager.GetString("ErrReinforcementYieldStress"));
    List<BIC> surfaceTypes = new List<BIC>()
        { BIC.OST_WallAnalytical, BIC.OST_FoundationSlabAnalytical, BIC.OST_FloorAnalytical };
    if (transReinforcementFy < 1e4 && !surfaceTypes.Contains( elementType ) )
        throw new Exception(
            Properties.Resources.ResourceManager.GetString("ErrReinforcementYieldStress"));
    if ( longReinforcementArea < 1e-7)
        throw new Exception("Element Settings: Rebar area is not properly defined.");
    if (transReinforcementArea < 1e-7 && !surfaceTypes.Contains(elementType) )
        throw new Exception("Element Settings: Rebar area is not properly defined.");
    /// </structural_toolkit_2015>
}

```

## 11.5 Reduction of the set of forces

Method: ReduceInternalForcesForLongitudinalReinforcement

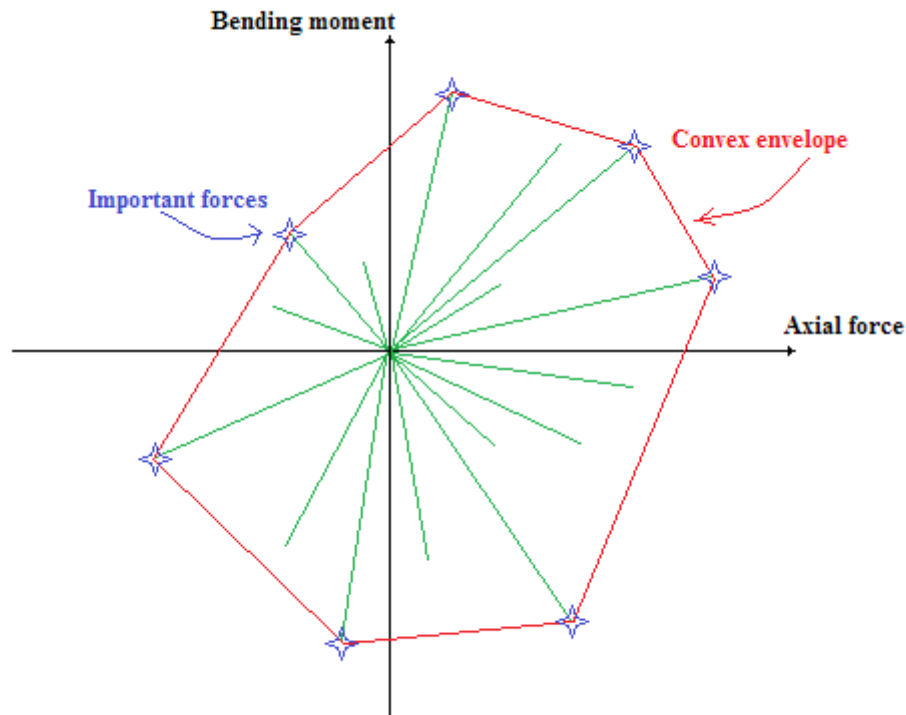
Method: ReduceInternalForcesForTransversalReinforcement

To improve the speed of calculation, the reduction of number of forces set can be used.

If only one element of the forces set should be taken into account (e.g. design for simple bending) only maximum and minimum values are important for design purposes. For a uniaxial bending (bending moment with axial force) the convex envelope of bending moments and axial forces are important.

Result of the reduction will be stored in the `List<InternalForces> listInternalForcesULSLrb` and in the `List<InternalForces> listInternalForcesSLSLrb`.

The location of the force generated by eccentricity for the uniaxial bending is verified.  
For small eccentricities, the preference for symmetrical reinforcement could be set based on an epsilon.



To create the convex envelope of the pairs: (bending moment - axial force) the convex hull (convex envelope) method can be used.

This method is implemented in the Autodesk.CodeChecking library.(namespace Autodesk.CodeChecking.Utils).

Based on a list of 2 dimensional points, the pair (bending moment - axial force must) will be converted to this type.

## Code region - Updated Version 2015

## ..\Concrete\ConcreteSectionDesign.cs

```

public void ReduceInternalForcesForLongitudinalReinforcement()
{
    longReinforcementInternalForcesULS.Clear(); longReinforcementInternalForcesSLS.Clear();
    switch (LongReinforcementCalculationType)
    {
        case ConcreteTypes.CalculationType.BendingV:
        {
            InternalForcesContainer forcesMaxMULs = new InternalForcesContainer(), forcesMinMULs = new InternalForcesContainer();
            InternalForcesContainer forcesMaxMSLs = new InternalForcesContainer(), forcesMinMSLs = new InternalForcesContainer();
            foreach (InternalForcesContainer forces in internalForces)
            {
                if (ForceLimitState.Sls == forces.LimitState)
                {
                    if (forces.MomentMy > forcesMaxMULs.MomentMy) forcesMaxMULs = forces;
                    else if (forces.MomentMy < forcesMinMULs.MomentMy) forcesMinMULs = forces;
                }
                else
                {
                    if (forces.MomentMy > forcesMaxMSLs.MomentMy) forcesMaxMSLs = forces;
                    else if (forces.MomentMy < forcesMinMSLs.MomentMy) forcesMinMSLs = forces;
                }
            }
            if (forcesMaxMULs.MomentMy > Double.Epsilon) longReinforcementInternalForcesULS.Add(forcesMaxMULs);
            if (forcesMinMULs.MomentMy < -Double.Epsilon) longReinforcementInternalForcesULS.Add(forcesMinMULs);
            if (forcesMaxMSLs.MomentMy > Double.Epsilon) longReinforcementInternalForcesSLS.Add(forcesMaxMSLs);
            if (forcesMinMSLs.MomentMy < -Double.Epsilon) longReinforcementInternalForcesSLS.Add(forcesMinMSLs);
        }
        break;
        /// <structural_toolkit_2015>
        case ConcreteTypes.CalculationType.EccentricBendingV:
        {
            List<Point2D> vNMULs = new List<Point2D>(), vNMSLs = new List<Point2D>();
            List<int> vNMULsIndex = new List<int>(), vNMSLsIndex = new List<int>();
            for (int i = 0; i < internalForces.Count(); i++)
            {
                if (ForceLimitState.Sls == internalForces[i].LimitState)
                {
                    vNMSLs.Add(new Point2D(internalForces[i].ForceFx, internalForces[i].MomentMy));
                    vNMSLsIndex.Add(i);
                }
                else
                {
                    vNMULs.Add(new Point2D(internalForces[i].ForceFx, internalForces[i].MomentMy));
                    vNMULsIndex.Add(i);
                }
            }
            List<int> v1NMULs = Autodesk.CodeChecking.Utils.ConvexHull(vNMULs), v1NMSLs = Autodesk.CodeChecking.Utils.ConvexHull(vNMSLs);
            double absForceFx = 0;
            double eccentricity = 0;
            foreach (int index in v1NMULs)
            {
                if (!IsZeroForces(internalForces[vNMULsIndex[index]], true))
                {
                    longReinforcementInternalForcesULS.Add(internalForces[vNMULsIndex[index]]);
                    if (!symmetricalReinforcementPreferable)
                    {
                        absForceFx = Math.Abs(internalForces[vNMULsIndex[index]].ForceFx);
                        if (absForceFx > Double.Epsilon)
                        {
                            eccentricity = Math.Abs(internalForces[vNMULsIndex[index]].MomentMy) / absForceFx;
                            symmetricalReinforcementPreferable = (eccentricity < 0.25 * sectionHeight);
                        }
                    }
                }
            }
            foreach (int index in v1NMSLs)
            {
                if (!IsZeroForces(internalForces[vNMSLsIndex[index]], true))
                {
                    longReinforcementInternalForcesSLS.Add(internalForces[vNMSLsIndex[index]]);
                    if (!symmetricalReinforcementPreferable)
                    {
                        absForceFx = Math.Abs(internalForces[vNMSLsIndex[index]].ForceFx);
                        if (absForceFx > Double.Epsilon)
                        {
                            eccentricity = Math.Abs(internalForces[vNMSLsIndex[index]].MomentMy) / absForceFx;
                            symmetricalReinforcementPreferable = (eccentricity < 0.25 * sectionHeight);
                        }
                    }
                }
            }
        }
        break;
        /// <structural_toolkit_2015>
        case ConcreteTypes.CalculationType.AxialForce:
        {
            symmetricalReinforcementPreferable = true;
            InternalForcesContainer forcesMaxNULs = new InternalForcesContainer(), forcesMinNULs = new InternalForcesContainer();
            InternalForcesContainer forcesMaxNSLs = new InternalForcesContainer(), forcesMinNSLs = new InternalForcesContainer();
            foreach (InternalForcesContainer forces in internalForces)
            {
                if (ForceLimitState.Sls == forces.LimitState)
                {
                    if (forces.ForceFx > forcesMaxNSLs.ForceFx) forcesMaxNSLs = forces;
                    else if (forces.ForceFx < forcesMinNSLs.ForceFx) forcesMinNSLs = forces;
                }
                else
                {
                    if (forces.ForceFx > forcesMaxNULs.ForceFx) forcesMaxNULs = forces;
                    else if (forces.ForceFx < forcesMinNULs.ForceFx) forcesMinNULs = forces;
                }
            }
            if (forcesMaxNULs.ForceFx > Double.Epsilon) longReinforcementInternalForcesULS.Add(forcesMaxNULs);
            if (forcesMinNULs.ForceFx < -Double.Epsilon) longReinforcementInternalForcesULS.Add(forcesMinNULs);
            if (forcesMaxNSLs.ForceFx > Double.Epsilon) longReinforcementInternalForcesSLS.Add(forcesMaxNSLs);
            if (forcesMinNSLs.ForceFx < -Double.Epsilon) longReinforcementInternalForcesSLS.Add(forcesMinNSLs);
        }
        break;
        default:
        {
            break;
        }
    }
}

```

The same type of reduction could be used for designing the transverse reinforcement. For pure shearing and pure tension only extreme values are taken into account during the design process:

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

public void ReduceInternalForcesForTransversalReinforcement()
{
    transReinforcementInternalForcesULS.Clear();
    if (ConcreteTypes.CalculationType.ShearingZ == transReinforcementCalculationType){
        InternalForcesContainer forcesMaxVULs = new InternalForcesContainer();
        InternalForcesContainer forcesMinVULs = new InternalForcesContainer();
        foreach (InternalForcesContainer forces in internalForces){
            if (ForceLimitState.Sls != forces.LimitState){
                if (forces.ForceFz > forcesMaxVULs.ForceFz)
                    forcesMaxVULs = forces;
                else if (forces.ForceFz < forcesMinVULs.ForceFz)
                    forcesMinVULs = forces;
            }
        }
        if (forcesMaxVULs.ForceFz > Double.Epsilon)
            transReinforcementInternalForcesULS.Add(forcesMaxVULs);
        if (forcesMinVULs.ForceFz < -Double.Epsilon)
            transReinforcementInternalForcesULS.Add(forcesMinVULs);
    }
}

```

Note:

The convex hull method for uniaxial shearing with torsion can be used if the code guarantees convex shape for torsion and shear capacity interaction.

## 11.6 Longitudinal and transversal reinforcement calculation

Method: CalculateLongitudinalReinforcement

Method: CalculateTransversalReinforcement

CalculateLongitudinalReinforcement() method is dedicated to the longitudinal reinforcement calculation.

Here, we have four design loops, Two for ULS and two for SLS.

Loops are doubled due to the reduction of forces.

The reduced lists of forces are checked first. If this list is empty, the full list of forces is used. otherwise calculation used the reduced list.

Before each limit state loops, the material properties for this limit state are set.

For non-zero single forces set, a specialized method is call. After each calculation current result are stored.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateLongitudinalReinforcement(){
    SetMaterialParameters(ForceLimitState.Uls);
    if (longReinforcementInternalForcesULS.Count() > 0){
        foreach (InternalForcesContainer forces in longReinforcementInternalForcesULS){
            if (IsZeroForces(forces, true))
                continue;
            CalculateLongitudinalReinforcementULS(forces);
            longitudinalReinforcement.CurrentToFinal();
        }
    }
    else{
        foreach (InternalForcesContainer forces in internalForces)
        {
            if (ForceLimitState.Sls == forces.LimitState)
                continue;
            if (IsZeroForces(forces, true))
                continue;
            CalculateLongitudinalReinforcementULS(forces);
            longitudinalReinforcement.CurrentToFinal();
        }
    }
    SetMaterialParameters(ForceLimitState.Sls);
    if (longReinforcementInternalForcesSLS.Count() > 0){
        foreach (InternalForcesContainer forces in longReinforcementInternalForcesSLS)
        {
            if (IsZeroForces(forces, true))
                continue;
            CalculateLongitudinalReinforcementSLS(forces);
            longitudinalReinforcement.CurrentToFinal();
        }
    }
    else{
        foreach (InternalForcesContainer forces in internalForces){
            if (ForceLimitState.Sls != forces.LimitState)
                continue;
            if (IsZeroForces(forces, true))
                continue;
            CalculateLongitudinalReinforcementSLS(forces);
            longitudinalReinforcement.CurrentToFinal();
        }
    }
}
}

```

Method for the transverse reinforcement design works in the same way as longitudinal.

In this example transverse reinforcement design is based on ULS only.

Base on this assumption, only the ULS loops is implemented.

At the end of calculation stirrups density are calculated. Density is based on one stirrup area and number of arms. The number of arms in the example is fixed to two.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
private void CalculateTransversalReinforcement(){
    SetMaterialParameters(ForceLimitState.Uls);
    if (transReinforcementInternalForcesULS.Count() > 0){
        foreach (InternalForcesContainer forces in transReinforcementInternalForcesULS){
            if (IsZeroForces(forces, true))
                continue;
            CalculateTransversalReinforcementULS(forces);
            transversalReinforcement.CurrentToFinal();
        }
    }
    else{
        foreach (InternalForcesContainer forces in internalForces){
            if (ForceLimitState.Sls == forces.LimitState)
                continue;
            if (IsZeroForces(forces, true))
                continue;
            CalculateTransversalReinforcementULS(forces);
            transversalReinforcement.CurrentToFinal();
        }
    }
    transversalReinforcement.SetTransversalDensity(transReinforcementNumberOfLegs,
                                                    transReinforcementArea);
}
```

## 11.7 Checking of zero value of forces

Method: IsZeroForces

What should be treated as negligible force is dependent to the type of calculation. However this kind of check is very popular, so we should create an appropriate method for this check:



Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private bool IsZeroForces(InternalForcesContainer forces, bool longitudinalReinforcemenet){
    bool zeroForces = true;
    if(longitudinalReinforcemenet){
        switch (longReinforcementCalculationType){
            case ConcreteTypes.CalculationType.BendingY:
                zeroForces = CalculationUtility.IsZeroM(forces.MomentMy);
                break;
            case ConcreteTypes.CalculationType.AxialForce:
                zeroForces = CalculationUtility.IsZeroN(forces.ForceFx);
                break;
            case ConcreteTypes.CalculationType.EccentricBendingY:
                zeroForces = CalculationUtility.IsZeroM(forces.MomentMy) &&
                    CalculationUtility.IsZeroN(forces.ForceFx);
                break;
            default:
                zeroForces = CalculationUtility.IsZeroM(forces.MomentMy) &&
                    CalculationUtility.IsZeroN(forces.ForceFx) &&
                    CalculationUtility.IsZeroM(forces.MomentMz);
                break;
        }
    }
    else{
        switch (transReinforcementCalculationType){
            case ConcreteTypes.CalculationType.ShearingZ:
                zeroForces = CalculationUtility.IsZeroN(forces.ForceFz);
                break;
            case ConcreteTypes.CalculationType.Torsion:
                zeroForces = CalculationUtility.IsZeroM(forces.MomentMx);
                break;
            case ConcreteTypes.CalculationType.TorsionWithShearingZ:
                zeroForces = CalculationUtility.IsZeroM(forces.MomentMx) &&
                    CalculationUtility.IsZeroN(forces.ForceFz);
                break;
            default:
                zeroForces = CalculationUtility.IsZeroM(forces.MomentMx) &&
                    CalculationUtility.IsZeroN(forces.ForceFz) &&
                    CalculationUtility.IsZeroN(forces.ForceFy);
                break;
        }
    }
    return zeroForces;
}

```

## 11.8 Material properties according to limit state

Method: SetMaterialParameters

For majority of national standard based on limit state concept, the material properties are different for both limit state.

So a new method is needed here to set the properties of the concrete (design strength, model) and the design strength for steel used for transverse and longitudinal rebars.

In the example, we could use `RcVerificationHelperUtility` objet to set these properties.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void SetMaterialParameters(ForceLimitState state)
{
    longitudinalReinforcement.ModulusOfElasticity = 200e9;
    switch (state)
    {
        default:
        case ForceLimitState.Uls:
            concreteParameters.SetStrainStressModelRectangular(concreteFc / 1.5,
                                                                0.0035, concreteYoungModulus, 0.8);
            longitudinalReinforcement.SetStrenghtPartialFactor(1.15);
            transversalReinforcement.SetStrenghtPartialFactor(1.15);
            break;
        case ForceLimitState.Sls:
            concreteParameters.SetStrainStressModelLinear(concreteFc,
                                                           concreteFc / concreteYoungModulus, concreteYoungModulus);
            longitudinalReinforcement.SetStrenghtPartialFactor(1.0);
            transversalReinforcement.SetStrenghtPartialFactor(1.0);
            break;
    }
    if (verificationHelper != null)
    {
        verificationHelper.SetConcrete(concreteParameters);
        verificationHelper.SetSteel(longitudinalReinforcement.Strength,
                                    longitudinalReinforcement.ModulusOfElasticity, 0.01, 1.0);
    }
}

```

## 11.9 Cross section stiffness calculation

Method: CalculateStiffness

This method is dedicated to the calculation of the stiffness of the cross section.

This type of calculation is performed only for beam elements.

We have two design loops (as before – to support the reduction of forces).

In this case only SLS forces are taken into account.

The reduced lists of forces are checked first. If the list is empty, the full list will be used.

Otherwise calculation used reduced SLS list.

Before calculation the material properties for this limit state are set.

For non-zero single forces set, a specialized method is call.

After each calculation minimum value of current and previous results are stored.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateStiffness(){
    bool noSLS = true;
    if (elementType == Autodesk.Revit.DB.BuiltInCategory.OST_BeamAnalytical){
        minStiffness = sectionGeometry.MomentOfInertiaX * concreteParameters.ModulusOfElasticity;
        SetMaterialParameters(ForceLimitState.Sls);
        if (longReinforcementInternalForcesSLS.Count() > 0){
            noSLS = false;
            foreach (InternalForcesContainer forces in longReinforcementInternalForcesSLS){
                if (IsZeroForces(forces, true))
                    continue;
                else
                    minStiffness =
                        Math.Min(minStiffness, CalculateStiffnesSLS(forces, concreteCreepCoefficient));
            }
        }
        else{
            foreach (InternalForcesContainer forces in internalForces){
                if (ForceLimitState.Sls != forces.LimitState)
                    continue;
                noSLS = false;
                if (IsZeroForces(forces, true))
                    continue;
                else
                    minStiffness =
                        Math.Min(minStiffness, CalculateStiffnesSLS(forces, concreteCreepCoefficient));
            }
        }
    }
    if (noSLS)
        minStiffness = 0;
}

```

## 11.10 Longitudinal reinforcement - ULS and SLS

Method: CalculateLongitudinalReinforcementULS

Method: CalculateLongitudinalReinforcementSLS

Design of longitudinal reinforcement could be easier and faster when a simplification is possible.

In this place possibility of simplification is checked. Simplification is depending to type of design, type of acting forces and cross section shape.

If it is possible, the best function for design could be chosen.

The easiest case is a rectangular section under simple bending.

In this example such type of design was developed only for ULS.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateLongitudinalReinforcementULS(InternalForcesContainer forces){
    if (ConcreteTypes.CalculationType.BendingY == longReinforcementCalculationType)
    {
        if (SectionShapeType.RectangularBar == sectionType)
            CalculateLongitudinalReinforcementSimplify(ref forces);
        else
            CalculateLongitudinalReinforcementUniaxial(ref forces);
    }
    else if ((ConcreteTypes.CalculationType.EccentricBendingY == longReinforcementCalculationType) ||
             CalculationUtility.IsZeroM(forces.MomentMz)) {
        if(CalculationUtility.IsZeroN(forces.ForceFx)){
            if (SectionShapeType.RectangularBar == sectionType)
                CalculateLongitudinalReinforcementSimplify(ref forces);
            else
                CalculateLongitudinalReinforcementUniaxial(ref forces);
        }
        else if (CalculationUtility.IsZeroM(forces.MomentMy))
            CalculateLongitudinalReinforcementPureAxialForce(forces.ForceFx);
        else
            CalculateLongitudinalReinforcementUniaxial(ref forces);
    }
    else{
        if (CalculationUtility.IsZeroM(forces.MomentMz)) {
            if (!CalculationUtility.IsZeroM(forces.MomentMy)){
                if (SectionShapeType.RectangularBar == sectionType)
                    CalculateLongitudinalReinforcementSimplify(ref forces);
                else
                    CalculateLongitudinalReinforcementUniaxial(ref forces);
            }
            else{
                if (!CalculationUtility.IsZeroN(forces.ForceFx))
                    CalculateLongitudinalReinforcementPureAxialForce(forces.ForceFx);
            }
        }
        else
            CalculateLongitudinalReinforcementBiaxial(ref forces);
    }
}

```

Method for SLS works similar to ULS. The difference is the lack of adequate methods for the simplified calculations for SLS in this example. Such simplifications are possible and can be developed.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
private void CalculateLongitudinalReinforcementSLS(InternalForcesContainer forces)
{
    switch (longReinforcementCalculationType)
    {
        case ConcreteTypes.CalculationType.BendingY:
            CalculateLongitudinalReinforcementUniaxial(ref forces);
            break;
        case ConcreteTypes.CalculationType.EccentricBendingY:
            CalculateLongitudinalReinforcementUniaxial(ref forces);
            break;
        default:
            if (CalculationUtility.IsZeroM(forces.MomentMz))
                CalculateLongitudinalReinforcementUniaxial(ref forces);
            else
                CalculateLongitudinalReinforcementBiaxial(ref forces);
            break;
    }
}
```

## 11.11 Minimum reinforcement for longitudinal reinforcement

Method: SetLongitudinalMimimumReinforcement

The calculation of minimum longitudinal reinforcement depends of the type of the element and will be implemented as follow on this example:

- For beams the value 0.1% of cross section area will be set only on the side where the reinforcement exists.
- For columns, 4 rebars in the cross section are set. Additionally the sum of reinforcement must be greater than 0.5% of cross section area. This reinforcement is placed in proportion to the existing reinforcement.
- For floor and foundation slab the value 0.05% of cross section area will be set only on the side where the reinforcement exists
- For wall 0.25% of cross section area in X direction and 0.1% of cross section area in Y direction

Code region - Updated Version 2015

..\Concrete\ConcreteSectionDesign.cs

```

private void SetLongitudinalMinimumReinforcement(){
    switch (elementType)
    {
        case BIC.OST_BeamAnalytical:
        {
            double minimumReinforcement = 0.001 * sectionGeometry.Area;
            if (longitudinalReinforcement.AsBottom > 0.0)
                longitudinalReinforcement.CurrentAsBottom = minimumReinforcement;
            if (longitudinalReinforcement.AsTop > 0.0)
                longitudinalReinforcement.CurrentAsTop = minimumReinforcement;
            longitudinalReinforcement.CurrentToFinal();
        }
        break;
        case BIC.OST_ColumnAnalytical:
        {
            double minimumReinforcementRebar = 4.0 * longReinforcementArea;
            double totalReinforcement = longitudinalReinforcement.AsTop +
                                         longitudinalReinforcement.AsBottom;
            if (totalReinforcement < minimumReinforcementRebar){
                longitudinalReinforcement.CurrentAsTop =
                    longitudinalReinforcement.CurrentAsBottom = 0.5 * minimumReinforcementRebar;
                longitudinalReinforcement.CurrentToFinal();
            }
            totalReinforcement = longitudinalReinforcement.TotalSectionReinforcement();
            double minimumReinforcement = 0.005 * sectionGeometry.Area;
            if (totalReinforcement < minimumReinforcement){
                double coef = totalReinforcement / minimumReinforcement;
                longitudinalReinforcement.CurrentAsBottom = longitudinalReinforcement.AsBottom / coef;
                longitudinalReinforcement.CurrentAsTop = longitudinalReinforcement.AsTop / coef;
                longitudinalReinforcement.CurrentAsLeft = longitudinalReinforcement.AsLeft / coef;
                longitudinalReinforcement.CurrentAsRight = longitudinalReinforcement.AsRight / coef;
                longitudinalReinforcement.CurrentToFinal();
            }
        }
        break;
        /// <structural_toolkit_2015>
        case BIC.OST_FloorAnalytical:
        case BIC.OST_FoundationSlabAnalytical:
        {
            double minimumReinforcement = 0.0005 * sectionGeometry.Area;
            if (longitudinalReinforcement.AsBottom > 0.0)
                longitudinalReinforcement.CurrentAsBottom = minimumReinforcement;
            if (longitudinalReinforcement.AsTop > 0.0)
                longitudinalReinforcement.CurrentAsTop = minimumReinforcement;
            longitudinalReinforcement.CurrentToFinal();
        }
        break;
        case BIC.OST_WallAnalytical:
        {
            double minimumReinforcement =
                (dimensioningDirection == ConcreteTypes.DimensioningDirection.X) ? 0.0025 : 0.001;
            minimumReinforcement *= sectionGeometry.Area;
            longitudinalReinforcement.CurrentAsBottom = minimumReinforcement;
            longitudinalReinforcement.CurrentAsTop = minimumReinforcement;
            longitudinalReinforcement.CurrentToFinal();
        }
        break;
        /// </structural_toolkit_2015>
        default:
            break;
    }
}

```

## 11.12 Transversal reinforcement calculation for ULS

Method: CalculateTransversalReinforcementULS

At this point, the method to calculate transverse reinforcement works in a similar way as longitudinal reinforcement. The possibility of simplification is checked and simplification depends on acting forces. If it is possible the best function for design will be chosen.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
private void CalculateTransversalReinforcementULS(InternalForcesContainer forces)
{
    switch(transReinforcementCalculationType)
    {
        case ConcreteTypes.CalculationType.ShearingZ:
            CalculateTransversalReinforcementPureShear(forces.ForceFz, sectionHeight);
            break;
        case ConcreteTypes.CalculationType.Torsion:
            CalculateTransversalReinforcementPureTorsion(forces.MomentMx);
            break;
        case ConcreteTypes.CalculationType.TorsionWithShearingZ:
            CalculateTransversalReinforcementShearTorsion(forces.ForceFy, forces.MomentMx, sectionHeight);
            break;
        default:
            {
                if (CalculationUtility.IsZeroN(forces.ForceFz) && CalculationUtility.IsZeroN(forces.ForceFy))
                    CalculateTransversalReinforcementPureTorsion(forces.MomentMx);
                else if (CalculationUtility.IsZeroM(forces.MomentMx) &&
                        CalculationUtility.IsZeroN(forces.ForceFy))
                    CalculateTransversalReinforcementPureShear(forces.ForceFz, sectionHeight);
                else if (CalculationUtility.IsZeroN(forces.ForceFy))
                    CalculateTransversalReinforcementShearTorsion(forces.ForceFy, forces.MomentMx,
                                                                sectionHeight);
            }
        else
            CalculateTransversalReinforcementGeneral(ref forces);
    }
    break;
}
```

## 11.13 Simplified longitudinal reinforcement design

Method: CalculateLongitudinalReinforcementSimplify

Method: CalculateLongitudinalReinforcementPureAxialForce

These two methods describe the easiest cases of design. However, there are very popular in the engineering world.

These types of design could be described as a collection of simple formulas. In that way we could have faster calculation for the popular problem. The formulas exposed in this example are based on the simplified (rectangular) model for concrete.

Similar formulas could be developed for other models. A simple algorithm for bending with axial force is also possible.

First method is describing a simple bending for rectangular cross section. At the beginning, based on the maximum compression force in the concrete, the maximum bending moment for one side reinforcement is

calculated. If we have reinforcement only on tension side, the area of reinforcement is calculated based on quadratic equation otherwise compression and tension reinforcement are designed.

Code region

..\Concrete\ConcreteSectionDesign.cs

```
private void CalculateLongitudinalReinforcementSimplify(ref InternalForcesContainer forces){
    bool tensionOnTop = forces.MomentMy < 0.0;
    double absM = Math.Abs(forces.MomentMy);
    double steelTensStrain = longitudinalReinforcement.Strength /
        longitudinalReinforcement.ModulusOfElasticity;
    double concreteStrain = concreteParameters.StrainUltimateLimit;
    double d = sectionHeight - (tensionOnTop ? longReinforcementTopCover : longReinforcementBottomCover);
    double x = concreteStrain * d / (concreteStrain + steelTensStrain);
    x *= concreteParameters.EffectiveHeightReductionFactor; // 0.8
    double concreteForces = x * sectionHeight * sectionWidth * concreteParameters.DesignStrength;
    double maxOneSideReforcementMoment = concreteForces * (d - 0.5 * x);
    double tensionReinforcement = 0.0;
    double compressionReinforcement = 0.0;
    if (maxOneSideReforcementMoment > absM){
        double a = -0.5 * concreteParameters.EffectiveHeightReductionFactor;
        double b = d;
        double c = -absM / (concreteParameters.DesignStrength * sectionWidth *
            concreteParameters.EffectiveHeightReductionFactor);
        double xM1 = 0;
        double xM2 = 0;
        CalculationUtility.RootsOfQuadraticEquation(a, b, c, ref xM1, ref xM2);
        if (!CalculationUtility.RootsOfQuadraticEquation(a, b, c, ref xM1, ref xM2))
            throw new Exception (String.Format(
                "Invalid CalculateLongitudinalReinforcementSimplify delta <= 0.0 for case {0}.",
                forces.CaseName));
        double coeff = xM1 * xM2;
        double xM = coeff >= Double.Epsilon ? Math.Min(xM1, xM2) : Math.Max(xM1, xM2);
        if (xM <= 0.0)
            throw new Exception (String.Format(
                "Invalid CalculateLongitudinalReinforcementSimplify xM <= 0.0 for case {0}.",
                forces.CaseName));
        tensionReinforcement = (xM * concreteParameters.EffectiveHeightReductionFactor * sectionWidth *
            concreteParameters.DesignStrength) / longitudinalReinforcement.Strength;
    }
    else{
        tensionReinforcement = concreteForces / longitudinalReinforcement.Strength;
        compressionReinforcement = (absM - maxOneSideReforcementMoment) /
            (sectionHeight - longReinforcementTopCover - longReinforcementBottomCover);
        compressionReinforcement /= longitudinalReinforcement.Strength;
        tensionReinforcement += compressionReinforcement;
    }
    longitudinalReinforcement.CurrentAsTop = tensionOnTop ?
        tensionReinforcement : compressionReinforcement;
    longitudinalReinforcement.CurrentAsBottom = tensionOnTop ?
        compressionReinforcement : tensionReinforcement;
}
```

Next method allows the design of any section under pure tension or compression. For pure tension the reinforcement must transfer whole axial force. For the case compression, at first, the concrete transfer the axial force and the reinforcement is designed for a rest of the actions.



Code region

..\Concrete\ ConcreteSectionDesign.cs

```
void CalculateLongitudinalReinforcementPureAxialForce(double forcesN){
    double totalSteelArea = 0;
    if (CalculationUtility.LtZeroN(forcesN))
        totalSteelArea = -forcesN / longitudinalReinforcement.Strength;
    else{
        double NRd = sectionGeometry.Area * concreteParameters.DesignStrength;
        forcesN -= NRd;
        if (CalculationUtility.GtZeroN(forcesN))
            totalSteelArea = forcesN / longitudinalReinforcement.Strength;
    }
    longitudinalReinforcement.CurrentAsTop = longitudinalReinforcement.CurrentAsBottom =
        longitudinalReinforcement.CurrentAsRight = longitudinalReinforcement.CurrentAsLeft =
                                                    totalSteelArea / 4.0;
}
```

## 11.14 Longitudinal reinforcement for uniaxial/biaxial bending

### 11.14.1 CalculateLongitudinalReinforcementUniaxial

Method: CalculateLongitudinalReinforcementUniaxial

At the beginning of this method, the minimum of reinforcement increase is calculated. This minimum value is set for the top or the bottom depending on the sign of acting bending moment.

If the symmetrical reinforcement is preferred, the reinforcement is set on both sides.

In the next step, a safety factor is calculated and if the safety factor doesn't fit expectations, some dedicated methods for iterative searching of reinforcement are used.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateLongitudinalReinforcementUniaxial(ref InternalForcesContainer forces){
    double minimumIncreaseReinforcement = CalculationUtility.MinimumIncreaseOfReinforcement ();
    if (verificationHelper == null){
        verificationHelper = RcVerificationHelperUtility.CreateRcVerificationHelperUtility(
            sectionType, ref sectionGeometry, longReinforcementTopCover, longReinforcementBottomCover);
        SetMaterialParameters(forces.LimitState);
    }
    if (symmetricalReinforcementPreferable)
        longitudinalReinforcement.CurrentAsTop =
            longitudinalReinforcement.CurrentAsBottom = minimumIncreaseReinforcement;
    else if (forces.MomentMy >= 0.0)
        longitudinalReinforcement.CurrentAsBottom = minimumIncreaseReinforcement;
    else
        longitudinalReinforcement.CurrentAsTop = minimumIncreaseReinforcement;
    longitudinalReinforcement.CurrentAsTop =
        Math.Max(longitudinalReinforcement.CurrentAsTop, longitudinalReinforcement.AsTop);
    longitudinalReinforcement.CurrentAsBottom =
        Math.Max(longitudinalReinforcement.CurrentAsBottom, longitudinalReinforcement.AsBottom);
    verificationHelper.SetReinforcement(longitudinalReinforcement.CurrentAsTop,
        longitudinalReinforcement.CurrentAsBottom);
    double safetyFactor = verificationHelper.SafetyFactor(forces);
    if (!CalculationUtility.IsSafety(safetyFactor))
        FindOptimalLongitudinalReinforcementUniaxial(ref safetyFactor, ref forces);
}

```

## 11.14.2 CalculateLongitudinalReinforcementBiaxial

Method: CalculateLongitudinalReinforcementBiaxial

The way of design under biaxial bending action is very similar to uniaxial bending. In this example, it is assumed that for biaxial bending top reinforcement is equal to the bottom and right is equal to the left. If results don't satisfy minimal requirements, the minimum reinforcement is set as "rebar" on the each corner.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateLongitudinalReinforcementBiaxial(ref InternalForcesContainer forces)
{
    if (verificationHelper == null){
        verificationHelper = RcVerificationHelperUtility.CreateRcVerificationHelperUtility(
            sectionType, ref sectionGeometry, longReinforcementTopCover, longReinforcementBottomCover);
        SetMaterialParameters(forces.LimitState);
    }
    bool setInitialReinforcement = CalculationUtility.IsZeroReinforcement(
        longitudinalReinforcement.TotalSectionReinforcement());
    if (setInitialReinforcement)
        longitudinalReinforcement.CurrentAsTop = longitudinalReinforcement.CurrentAsBottom =
            2 * longReinforcementArea;
    else
    {
        longitudinalReinforcement.CurrentAsTop = longitudinalReinforcement.AsTop;
        longitudinalReinforcement.CurrentAsBottom = longitudinalReinforcement.AsBottom;
        longitudinalReinforcement.CurrentAsRight = longitudinalReinforcement.AsRight;
        longitudinalReinforcement.CurrentAsLeft = longitudinalReinforcement.AsLeft;
    }
    verificationHelper.SetReinforcement(longitudinalReinforcement.CurrentAsTop,
        longitudinalReinforcement.CurrentAsBottom, longitudinalReinforcement.CurrentAsRight,
        longitudinalReinforcement.CurrentAsLeft);
    double safetyFactor = verificationHelper.SafetyFactor(forces);
    if (!CalculationUtility.IsSafety(safetyFactor))
        FindOptimalLongitudinalReinforcementBiaxial(ref safetyFactor, ref forces);
}

```

## 11.15 Cross section stiffness calculation

Method: CalculateStiffnessSLS

To calculate beam deflection, the stiffness of cross section under acting loads is necessary.

Stiffness in this example is based on the properties of the cross section (moment of inertia), material (Young modulus), creep coefficient, moment of inertia of cracking system and ratio of acting forces to cracking forces. The calculation is based on following formula:

$$\text{stiffness} = (I_{RC} \cdot E_{RC}) = \left( \left( \frac{F_{cr}}{F} \right)^2 \cdot I_{cr} + \left( 1 - \left( \frac{F_{cr}}{F} \right)^2 \right) \cdot I \right) \cdot \frac{E}{(1 - \phi)}$$

where:

- F – set of acting forces
- $F_{cr}$  - set of cracking forces where  $F_{cr}(M,N) = \alpha \cdot F = F(\alpha \cdot M, \alpha \cdot N)$
- I - moment of inertia for cross section
- $I_{cr}$  - moment of inertia for cracking section
- E – Young modulus
- $\Phi$  - creep coefficient

Code region - Updated Version 2015

..\Concrete\ ConcreteSectionDesign.cs

```

/// <structural_toolkit_2015>
private double CalculateStiffnesSLS(InternalForcesContainer forces, double creepCoefficient)
{
    double stiffnes = 0.0;
    if (SectionShapeType.RectangularBar == sectionType){
        if (verificationHelper == null){
            verificationHelper = RcVerificationHelperUtility.CreateRcVerificationHelperUtility(
                sectionType, ref sectionGeometry, longReinforcementTopCover, longReinforcementBottomCover);
            SetMaterialParameters(forces.LimitState);
        }
        double momentOfInertiaConcreteSection = sectionGeometry.MomentOfInertiaX;
        stiffnes = momentOfInertiaConcreteSection;
        if (!IsZeroForces(forces, true)){
            double concreteTensionLimit = 0.3 *
                Math.Pow(concreteParameters.DesignStrength * 1e-6, 2.0 / 3.0) * 1e6;
            double crackingCoefficient =
                verificationHelper.ForcesToCrackingForces(forces, concreteTensionLimit);
            if (crackingCoefficient.CompareTo(0.0) < 0)
                throw new Exception(String.Format(
                    "Invalid ForcesToCrackingForces in CalculateStiffnesSLS for case {0}.", forces.CaseName));
            if (crackingCoefficient > 1.0){
                crackingCoefficient = 1.0 / crackingCoefficient;
                crackingCoefficient *= crackingCoefficient;
            }
            else
                crackingCoefficient = 0;
            double momentOfInertiaCrackingConcreteSection =
                verificationHelper.InertiaOfCrackingSection(forces);
            stiffnes = (1.0 - crackingCoefficient) * momentOfInertiaConcreteSection +
                crackingCoefficient * momentOfInertiaCrackingConcreteSection;
        }
        stiffnes *= concreteParameters.ModulusOfElasticity / (1 + creepCoefficient);
    }
    else
        designWarning.Add(
            "Only rectangular cross-section can be used on this path. 3th party implementation is necessary");
    return stiffnes;
}
/// </structural_toolkit_2015>

```

## 11.16 Symmetrical and unsymmetrical reinforcement for uniaxial bending

Method: FindOptimalLongitudinalReinforcementUniaxial

In some cases the symmetrical reinforcement is preferable for uniaxial bending. This situation is typical when the axial force is dominant. In the case when the bending moment is dominant, the better solution is an unsymmetrical reinforcement.

For this reason, we could split calculations into two ways: symmetrical and the unsymmetrical.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
private void FindOptimalLongitudinalReinforcementUniaxial
    (ref double safetyFactor, ref InternalForcesContainer forces){
    if (symmetricalReinforcementPreferable)
        FindOptimalLongitudinalReinforcementUniaxialSymmetrical(ref safetyFactor, ref forces);
    else
        FindOptimalLongitudinalReinforcementUniaxialUnsymmetrical(ref safetyFactor, ref forces);
}
```

## 11.17 Search for the optimal longitudinal reinforcement

Method: FindOptimalLongitudinalReinforcementUniaxialSymmetrical

Method: FindOptimalLongitudinalReinforcementUniaxialUnsymmetrical

Method: FindOptimalLongitudinalReinforcementBiaxial

In this example, an iterative procedure to find the optimal reinforcement is used.

At the beginning of this process, the capacity and previous reinforcement area are checked.

The reinforcement is increased inside the loop until the safety factor is satisfied.

Simultaneously previous value of reinforcement is stored.

At the end we have "too big" and "too small" values of reinforcements and based on that, we will start a bisection algorithm.

For symmetrical reinforcement and uniaxial bending, both areas top and bottom are increased. This method is developed in FindOptimalLongitudinalReinforcementUniaxialSymmetrical.

For unsymmetrical reinforcement the increment of reinforcement at the top and bottom are independent and the method used is FindOptimalLongitudinalReinforcementUniaxialUnsymmetrical.

In case of biaxial bending, we could split the reinforcement into pairs. (Top – Bottom) and (Left – Right). Inside each pair, the reinforcement increase is combined and each pairs are treated independently. This method is developed in FindOptimalLongitudinalReinforcementBiaxial.

## Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void FindOptimalLongitudinalReinforcementUniaxialSymmetrical(
    ref double safetyFactor, ref InternalForcesContainer forces){
    double reinforcementIncrease = 0;
    double safetyFactorTopBottom = -1;
    bool resultNotOK = true;
    int i = 0;
    double[] asTopBottom = new double[] {longitudinalReinforcement.CurrentAsTop,
                                           longitudinalReinforcement.CurrentAsBottom, Double.MaxValue };
    double increasesafetyFactorTopBottom = 1.0;
    while (!CalculationUtility.IsIterEnd(++i) && resultNotOK){
        reinforcementIncrease = CalculationUtility.MinimumIncreaseOfReinforcement();
        verificationHelper.SetReinforcement(asTopBottom[1] + reinforcementIncrease,
                                           asTopBottom[1] + reinforcementIncrease);
        safetyFactorTopBottom = verificationHelper.SafetyFactor(forces);
        resultNotOK = !CalculationUtility.IsSafety(safetyFactorTopBottom);
        if (resultNotOK){
            asTopBottom[0] = asTopBottom[1] += reinforcementIncrease;
            increasesafetyFactorTopBottom = (safetyFactorTopBottom - safetyFactor);
            increasesafetyFactorTopBottom *= (1.0 - safetyFactor);
            if (increasesafetyFactorTopBottom > Double.Epsilon)
                reinforcementIncrease = (2.0 * CalculationUtility.MinimumIncreaseOfReinforcement()) /
                                         increasesafetyFactorTopBottom;
            else
                reinforcementIncrease = Double.MaxValue;
            AdjustReinforcementIncrease(ref reinforcementIncrease);
        }
        else{
            asTopBottom[2] = asTopBottom[1];
            safetyFactor = safetyFactorTopBottom;
        }
        asTopBottom[1] += reinforcementIncrease;
        verificationHelper.SetReinforcement(asTopBottom[1], asTopBottom[1]);
        safetyFactor = verificationHelper.SafetyFactor(forces);
        resultNotOK = !CalculationUtility.IsSafety(safetyFactor);
        if (resultNotOK)
            asTopBottom[0] = asTopBottom[1];
        else
            asTopBottom[2] = asTopBottom[1] += reinforcementIncrease;
    }
    if (resultNotOK)
        throw new Exception(String.Format(
            "Too many iteration for case {0}. FindOptimalLongitudinalReinforcementUniaxialSymmetrical.",
            forces.CaseName));
    else{
        resultNotOK = !CalculationUtility.IsSafetyOptimal(safetyFactor);
        if (resultNotOK)
            BisectionForReinforcementAdjustment(ref forces, asTopBottom, asTopBottom);
        else{
            longitudinalReinforcement.CurrentAsTop = asTopBottom[1];
            longitudinalReinforcement.CurrentAsBottom = asTopBottom[1];
        }
    }
}
}

```

## Code region

## ..\Concrete\ConcreteSectionDesign.cs

```

private void FindOptimalLongitudinalReinforcementUniaxialUnsymmetrical
    (ref double safetyFactor, ref InternalForcesContainer forces){
    double reinforcementIncreaseTop = 0, reinforcementIncreaseBottom = reinforcementIncreaseTop, safetyFactorTop = -1, safetyFactorBottom = -1;
    bool resultNotOK = true;
    int i = 0;
    double[] asTop = new double[] { longitudinalReinforcement.CurrentAsTop, longitudinalReinforcement.CurrentAsTop, Double.MaxValue };
    double[] asBottom = new double[] { longitudinalReinforcement.CurrentAsBottom, longitudinalReinforcement.CurrentAsBottom, Double.MaxValue };
    double increaseSafetyFactorTop = 1, increaseSafetyFactorBottom = 1;
    while (!CalculationUtility.IsIterEnd(++i) && resultNotOK){
        reinforcementIncreaseBottom = reinforcementIncreaseTop = Math.Max(CalculationUtility.MinimumIncreaseOfReinforcement(),
            0.005 * (asTop[1] + asBottom[0]));

        verificationHelper.SetReinforcement(asTop[1] + reinforcementIncreaseTop, asBottom[0]);
        safetyFactorTop = verificationHelper.SafetyFactor(forces);
        resultNotOK = !CalculationUtility.IsSafety(safetyFactorTop);
        if (resultNotOK){
            verificationHelper.SetReinforcement(asTop[0], asBottom[1] + reinforcementIncreaseBottom);
            safetyFactorBottom = verificationHelper.SafetyFactor(forces);
            resultNotOK = !CalculationUtility.IsSafety(safetyFactorBottom);
            if (!resultNotOK){
                asTop[2] = asTop[0];
                asBottom[2] = asBottom[1] + reinforcementIncreaseBottom;
                safetyFactor = safetyFactorBottom;
            }
        }
        else{
            asTop[2] = asTop[1] + reinforcementIncreaseTop;
            asBottom[2] = asBottom[0];
            safetyFactor = safetyFactorBottom;
        }
        if (resultNotOK){
            increaseSafetyFactorTop = safetyFactorTop - safetyFactor;
            increaseSafetyFactorBottom = safetyFactorBottom - safetyFactor;
            if (increaseSafetyFactorTop > 2.0 * increaseSafetyFactorBottom){
                reinforcementIncreaseBottom = 0;
                reinforcementIncreaseTop = reinforcementIncreaseTop / increaseSafetyFactorTop;
                reinforcementIncreaseTop *= (1.0 - safetyFactor);
            }
            else if (increaseSafetyFactorBottom > 2.0 * increaseSafetyFactorTop){
                reinforcementIncreaseTop = 0;
                reinforcementIncreaseBottom = reinforcementIncreaseBottom / increaseSafetyFactorBottom;
                reinforcementIncreaseBottom *= (1.0 - safetyFactor);
            }
            else{
                double coefficientSafetyFactor = safetyFactorTop / (safetyFactorTop + safetyFactorBottom);
                if (coefficientSafetyFactor > Double.Epsilon){
                    reinforcementIncreaseTop = coefficientSafetyFactor * reinforcementIncreaseTop / increaseSafetyFactorTop;
                    reinforcementIncreaseBottom = (1.0 - coefficientSafetyFactor) * reinforcementIncreaseBottom / increaseSafetyFactorBottom;
                    reinforcementIncreaseTop *= (1.0 - safetyFactor);
                    reinforcementIncreaseBottom *= (1.0 - safetyFactor);
                }
                else
                    reinforcementIncreaseTop = reinforcementIncreaseBottom = Double.MaxValue;
            }
        }
        AdjustReinforcementIncrease(ref reinforcementIncreaseTop);
        AdjustReinforcementIncrease(ref reinforcementIncreaseBottom);
        asTop[1] += reinforcementIncreaseTop;
        asBottom[1] += reinforcementIncreaseBottom;
        verificationHelper.SetReinforcement(asTop[1], asBottom[1]);
        safetyFactor = verificationHelper.SafetyFactor(forces);
        resultNotOK = !CalculationUtility.IsSafety(safetyFactor);
        if (resultNotOK){
            if ((asTop[1] + asBottom[1]) > 0.5 * sectionGeometry.Area)
                throw new Exception(String.Format(
                    "Too big reinforcement. Reinforcement for case {0} is more than 50% of section area.", forces.CaseName));

            asTop[0] = asTop[1];
            asBottom[0] = asBottom[1];
        }
        else{
            asTop[2] = asTop[1];
            asBottom[2] = asBottom[1];
        }
    }
    if (resultNotOK)
        throw new Exception(String.Format(
            "Too many iterations for case {0}. FindOptimalLongitudinalReinforcementUniaxialUnsymmetrical.", forces.CaseName));
    else{
        resultNotOK = !CalculationUtility.IsSafetyOptimal(safetyFactor);
        if (resultNotOK)
            BisectionForReinforcementAdjustment(ref forces, asTop, asBottom);
        else{
            longitudinalReinforcement.CurrentAsTop = asTop[1];
            longitudinalReinforcement.CurrentAsBottom = asBottom[1];
        }
    }
}

```

## Code region

## ..\Concrete\ ConcreteSectionDesign.cs

```

private void FindOptimalLongitudinalReinforcementBiaxial(ref double safetyFactor, ref InternalForcesContainer forces){
    bool onlyTopBottom = false, onlyRightLeft = false, resultNotOK = true;
    double currentReinforcement = longitudinalReinforcement.CurrentAsTop + longitudinalReinforcement.CurrentAsBottom +
        longitudinalReinforcement.CurrentAsRight + longitudinalReinforcement.CurrentAsLeft;

    double reinforcementIncreaseTopBottom = 0.5 * (currentReinforcement / safetyFactor - currentReinforcement);
    AdjustReinforcementIncrease(ref reinforcementIncreaseTopBottom);
    double reinforcementIncreaseRightLeft = reinforcementIncreaseTopBottom;
    verificationHelper.SetReinforcement(longitudinalReinforcement.CurrentAsTop + reinforcementIncreaseTopBottom,
        longitudinalReinforcement.CurrentAsBottom + reinforcementIncreaseTopBottom,
        longitudinalReinforcement.CurrentAsRight, longitudinalReinforcement.CurrentAsLeft);

    double safetyFactorTopBottom = onlyRightLeft ? 0.0 : verificationHelper.SafetyFactor(forces);
    verificationHelper.SetReinforcement(longitudinalReinforcement.CurrentAsTop, longitudinalReinforcement.CurrentAsBottom,
        longitudinalReinforcement.CurrentAsRight + reinforcementIncreaseRightLeft,
        longitudinalReinforcement.CurrentAsLeft + reinforcementIncreaseRightLeft);

    double safetyFactorRightLeft = onlyTopBottom ? 0.0 : verificationHelper.SafetyFactor(forces);

    int i = 0;
    double[] asTopBottom = new double[] { longitudinalReinforcement.CurrentAsTop, Double.MaxValue, Double.MaxValue };
    double[] asRightLeft = new double[] { longitudinalReinforcement.CurrentAsRight, Double.MaxValue, Double.MaxValue };
    double increaseSafetyFactorTopBottom = 1.0, increaseSafetyFactorRightLeft = 1.0;
    while (!CalculationUtility.IsIterEnd(++i) && resultNotOK){
        increaseSafetyFactorTopBottom = safetyFactorTopBottom - safetyFactor;
        increaseSafetyFactorRightLeft = safetyFactorRightLeft - safetyFactor;
        double dRTopBottom2dSf = increaseSafetyFactorTopBottom > Double.Epsilon ? reinforcementIncreaseTopBottom / increaseSafetyFactorTopBottom : 0;
        double dRRRightLeft2dSf = increaseSafetyFactorRightLeft > Double.Epsilon ? reinforcementIncreaseRightLeft / increaseSafetyFactorRightLeft : 0;
        double dCoef = dRTopBottom2dSf / (dRTopBottom2dSf + dRRRightLeft2dSf);
        dCoef *= (1.0 - safetyFactor);
        reinforcementIncreaseTopBottom = dCoef * dRTopBottom2dSf;
        reinforcementIncreaseRightLeft = (1.0 - dCoef) * dRRRightLeft2dSf;
        AdjustReinforcementIncrease(ref reinforcementIncreaseTopBottom);
        AdjustReinforcementIncrease(ref reinforcementIncreaseRightLeft);
        asTopBottom[1] = asTopBottom[0] + reinforcementIncreaseTopBottom;
        asRightLeft[1] = asRightLeft[0] + reinforcementIncreaseRightLeft;
        verificationHelper.SetReinforcement(asTopBottom[1], asTopBottom[1], asRightLeft[1], asRightLeft[1]);
        safetyFactor = verificationHelper.SafetyFactor(forces);
        resultNotOK = !CalculationUtility.IsSafety(safetyFactor);
        if (resultNotOK){
            reinforcementIncreaseTopBottom = CalculationUtility.MinimumIncreaseOfReinforcement();
            verificationHelper.SetReinforcement(asTopBottom[1] + reinforcementIncreaseTopBottom,
                asTopBottom[1] + reinforcementIncreaseTopBottom, asRightLeft[1], asRightLeft[1]);
            safetyFactorTopBottom = onlyRightLeft ? 0.0 : verificationHelper.SafetyFactor(forces);
            resultNotOK = !CalculationUtility.IsSafety(safetyFactorTopBottom);
            if (resultNotOK){
                reinforcementIncreaseRightLeft = CalculationUtility.MinimumIncreaseOfReinforcement();
                verificationHelper.SetReinforcement(asTopBottom[1], asTopBottom[1],
                    asRightLeft[1] + reinforcementIncreaseRightLeft, asRightLeft[1] + reinforcementIncreaseRightLeft);
                safetyFactorRightLeft = onlyTopBottom ? 0.0 : verificationHelper.SafetyFactor(forces);
                resultNotOK = !CalculationUtility.IsSafety(safetyFactorRightLeft);
                if (!resultNotOK){
                    asTopBottom[2] = asTopBottom[1];
                    asRightLeft[2] = asRightLeft[1];
                    safetyFactor = safetyFactorRightLeft;
                }
            }
        }
        else{
            asTopBottom[2] = asTopBottom[1];
            asRightLeft[2] = asRightLeft[1];
            safetyFactor = safetyFactorTopBottom;
        }
        asTopBottom[0] = asTopBottom[1];
        asRightLeft[0] = asRightLeft[1];
    }
    else{
        asTopBottom[2] = asTopBottom[1];
        asRightLeft[2] = asRightLeft[1];
    }
}

if (resultNotOK)
    throw new Exception(String.Format("Too many iterations for case {0}. CalculateLongitudinalReinforcementBiaxial.", forces.CaseName));
else{
    resultNotOK = !CalculationUtility.IsSafetyOptimal(safetyFactor);
    if (resultNotOK)
        BisectionForReinforcementAdjustment(ref forces, asTopBottom, asTopBottom, asRightLeft, asRightLeft);
    else{
        longitudinalReinforcement.CurrentAsTop = asTopBottom[1];
        longitudinalReinforcement.CurrentAsBottom = asTopBottom[1];
        longitudinalReinforcement.CurrentAsRight = asRightLeft[1];
        longitudinalReinforcement.CurrentAsLeft = asRightLeft[1];
    }
}
}
}

```



## 11.18 Final Reinforcement adjustments

Method: BisectionForReinforcementAdjustment

Final adjustment of reinforcement is realized by a bisection algorithm.

At the beginning (input data) we have 4 arrays with reinforcement, one for each side. Each of them has 3 values of reinforcement: "too high", "current" and "too low".

The idea of this algorithm is to search the best reinforcement between "too high" and "too low" reinforcement values. Current value of reinforcement is calculated as mean value of "too high" and "too low". When this average value is calculated, calculation of the reinforcement safety factor is done and we should decide if current reinforcement value should be set to "too high" or to "too low" accordingly.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void BisectionForReinforcementAdjustment(ref InternalForcesContainer forces, double[] asTop,
double[] asBottom, double[] asRight = null, double[] asLeft = null){
    bool onlyTopBottomReinforcemet = (asRight == null || asLeft == null);
    if( (asTop.Length != 3 || asTop.Length !=3) ||
        (!onlyTopBottomReinforcemet && (asRight.Length != 3 || asLeft.Length !=3)))
        throw new Exception(String.Format(
            "Invalid parameter in BisectionForReinforcementAdjustment for case {0}.",
            forces.CaseName));
    bool resultNotOK = true;
    double safetyFactor = 0;
    int i = 0;
    while (!CalculationUtility.IsIterEnd(++i) && resultNotOK){
        asTop[1] = 0.5 * (asTop[0] + asTop[2] );
        asBottom[1] = 0.5 * (asBottom[0] + asBottom[2]);
        if(onlyTopBottomReinforcemet)
            verificationHelper.SetReinforcement(asTop[1], asBottom[1]);
        else{
            asRight[1] = 0.5 * (asRight[0] + asRight[2] );
            asLeft[1] = 0.5 * (asLeft[0] + asLeft[2] );
            verificationHelper.SetReinforcement(asTop[1], asBottom[1], asRight[1], asLeft[1]);
        }
        safetyFactor = verificationHelper.SafetyFactor(forces);
        resultNotOK = CalculationUtility.IsSafetyOptimal(safetyFactor);
        if (resultNotOK){
            if (!CalculationUtility.IsSafety(safetyFactor)){
                asTop[0] = asTop[1];
                asBottom[0] = asBottom[1];
                if(!onlyTopBottomReinforcemet){
                    asRight[0] = asRight[1];
                    asLeft[0] = asLeft[1];
                }
            }
            else{
                asTop[2] = asTop[1];
                asBottom[2] = asBottom[1];
                if(!onlyTopBottomReinforcemet){
                    asRight[2] = asRight[1];
                    asLeft[2] = asLeft[1];
                }
            }
        }
        longitudinalReinforcement.CurrentAsTop = resultNotOK ? asTop[2] : asTop[1];
        longitudinalReinforcement.CurrentAsBottom = resultNotOK ? asBottom[2] : asBottom[1];
        if(!onlyTopBottomReinforcemet){
            longitudinalReinforcement.CurrentAsRight = resultNotOK ? asRight[2] : asRight[1];
            longitudinalReinforcement.CurrentAsLeft = resultNotOK ? asLeft[2] : asLeft[1];
        }
    }
}

```

## 11.19 Basic transverse reinforcement calculation

Method: CalculateTransversalReinforcementPureShear

Method: CalculateTransversalReinforcementPureTorsion

Method: TorsionParameters

The shear calculation in this example is based on these assumptions:

- For shear force less than VRdc, calculation of stirrups are not necessary
- For shear force greater than VRdmax, stirrups calculation is not possible

- Otherwise stirrups spacing is calculated according to shear force, reinforcement parameters and section geometry.

Below, the parameter “dim” represents the dimension of the cross section parallel to the shear force.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
private void CalculateTransversalReinforcementPureShear(double v, double dim){
    double vAbs = Math.Abs(v);
    double vRdc = (0.02 * Math.Pow(concreteParameters.DesignStrength * 1e-6, 0.3)
                  * sectionGeometry.Area)*1e6;

    if (vAbs > vRdc){
        double vRdmax = Vrdmax();
        if (vAbs > vRdmax)
            throw new Exception("Shear force is too large.");
        else
            transversalReinforcement.CurrentSpacing =
                (1.62 * transReinforcementArea * dim * longReinforcementFy) / vAbs;
    }
}
```

The maximum value of shear force is calculated according to the strength of concrete and the cross section geometry.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```
private double Vrdmax(){
    return 0.5 * sectionWidth * concreteParameters.DesignStrength;
}
```

Calculation for torsion in the example is available only for rectangular section.

If the moment of torsion is bigger than maximum value (TRdmax), an exception is thrown.

Stirrup spacing and additional longitudinal reinforcement are also calculated in this place.

Typical cross section parameters necessary for this calculation are calculated in other methods and are delivered in a dictionary.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateTransversalReinforcementPureTorsion(double t){
    if (SectionShapeType.RectangularBar == sectionType){
        double tAbs = Math.Abs(t);
        Dictionary<string, double> tp = TorsionParameters();
        double tRdmax = tp["tArea"] * tp["tPerimeter"] * concreteParameters.DesignStrength;
        if (tAbs > tRdmax)
            throw new Exception("Torsional moment is too large.");
        else{
            double externalArea = 2.0 * transReinforcementArea;
            transversalReinforcement.CurrentSpacing =
                (tp["tArea"] * transversalReinforcement.Strength * externalArea) / (0.5 * tAbs);
            transversalReinforcement.CurrentAsBottom = transversalReinforcement.CurrentAsTop =
                0.5 * (tp["tPerimeter"] * tAbs / (2.0 * tp["tArea"] * longitudinalReinforcement.Strength));
        }
    }
    else
        throw new Exception(
            "Invalid section for torsion. Only rectangular cross-sections can be used on this path.");
}

```

Parameters necessary for torsion calculation are calculated in this place. All these parameters based on the cross section geometry and are returned in a dictionary.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private Dictionary<string, double> TorsionParameters(){
    double shearFlowPathThin = sectionGeometry.Area / sectionGeometry.Perimeter;
    double tArea = (sectionWidth - 0.5 * shearFlowPathThin) *
        (sectionHeight - 0.5 * shearFlowPathThin);
    double tPerimeter = 2.0 * (sectionWidth - 0.5 * shearFlowPathThin) +
        2.0 * (sectionHeight - 0.5 * shearFlowPathThin);
    Dictionary<string, double> tp = new Dictionary<string, double> {
        { "shearFlowPathThin", shearFlowPathThin },
        { "tArea", tArea },
        { "tPerimeter", tPerimeter } };
    return tp;
}

```

## 11.20 Complex calculation for transverse reinforcement

Method: CalculateTransversalReinforcementShearTorsion

Method: CalculateTransversalReinforcementGeneral

In this example complex situation for transversal reinforcement calculation are treated as sum of basic calculations and by an additional check for complex forces limit.

When shear and torsion are analyzed, the condition of simultaneity is checked and results from both analyses are summarized.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateTransversalReinforcementShearTorsion(double v, double t, double dim){
    double vAbs = Math.Abs(v);
    double vRdmax = Vrdmax();
    double tAbs = Math.Abs(t);
    double tRdmax = 0;
    if (SectionShapeType.RectangularBar == sectionType){
        Dictionary<string, double> tp = TorsionParameters();
        tRdmax = tp["tArea"] * tp["tPerimeter"] * concreteParameters.DesignStrength;
    }
    else{
        throw new Exception("Invalid section for torsion. rectangular cross-sections can be used on this path.");
    }
    double coeffMax = vAbs / vRdmax + tAbs / tRdmax;
    if (coeffMax > 1.0)
        throw new Exception("Shear and torsions forces are too large.");
    else
    {
        CalculateTransversalReinforcementPureShear(vAbs, dim);
        double tempSpacing = transversalReinforcement.CurrentSpacing;
        CalculateTransversalReinforcementPureTorsion(vAbs);
        transversalReinforcement.CurrentSpacing = 1.0 /
            (1.0 / transversalReinforcement.CurrentSpacing + 1.0 / tempSpacing);
    }
}

```

In the case of biaxial shear, we calculate the resultant of the two acting shear forces. Later, the biaxial shear calculation is simplified to a uniaxial shear case. The acting force we will use for calculation has as intensity the resultant calculated previously and as direction, the direction of the force with the biggest intensity from our 2 initial forces.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void CalculateTransversalReinforcementGeneral(ref InternalForcesContainer forces){
    double vAbs = Math.Sqrt(Math.Pow(forces.ForceFy, 2.0) + Math.Pow(forces.ForceFz, 2.0));
    double vRdmax = Vrdmax();
    if (vAbs > vRdmax)
        throw new Exception("Shear forces and torsion are too large.");
    else{
        if (Math.Abs(forces.ForceFz) > Math.Abs(forces.ForceFy))
            CalculateTransversalReinforcementShearTorsion(vAbs, forces.MomentMx, sectionHeight);
        else
            CalculateTransversalReinforcementShearTorsion(vAbs, forces.MomentMx, sectionWidth);
    }
}

```

## 11.21 Maximum stirrups spacing

Method: SetTransversalMaximumStirupSpacing

This method allows us to meet the requirements of the code regarding the minimum stirrup spacing. In this example the minimum stirrups spacing depends on the type of the calculations and the cross section geometry. In the real code implementation, it could be more complicate.

Code region

..\Concrete\ ConcreteSectionDesign.cs

```

private void SetTransversalMaximumStirupSpacing(){
    switch (transReinforcementCalculationType){
        default:
            case ConcreteTypes.CalculationType.ShearingZ:
                transversalReinforcement.CurrentSpacing = Math.Min(0.5 * sectionHeight, 0.4);
                break;
            case ConcreteTypes.CalculationType.TorsionWithShearingZ:
                transversalReinforcement.CurrentSpacing = Math.Min(0.4 * sectionHeight, 0.25);
                break;
            case ConcreteTypes.CalculationType.TransAll:
                transversalReinforcement.CurrentSpacing =
                    Math.Min(0.4 * Math.Min(sectionHeight, sectionWidth), 0.25);
                break;
    }
    transversalReinforcement.CurrentToFinial();
    transversalReinforcement.SetTransversalDensity(transReinforcementNumberOfLegs, transReinforcementArea);
}

```

## 11.22 Adjustment of the reinforcement increment

Method: AdjustReinforcementIncrease

Due to the calculation time the step of increase of reinforcement in the iterative process cannot be too small or too large.

Following method adjusts the increment:

Code region - Updated Version 2015

..\Concrete\ ConcreteSectionDesign.cs

```

/// <structural_toolkit_2015>
public void AdjustReinforcementIncrease(ref double reinforcementIncrease)
{
    if (reinforcementIncrease > Double.Epsilon)
    {
        reinforcementIncrease = Math.Max(reinforcementIncrease,
            CalculationUtility.MinimumIncreaseOfReinforcement());
        reinforcementIncrease = Math.Min(reinforcementIncrease, 0.01 * sectionGeometry.Area);
    }
    else
        reinforcementIncrease = 0;
}
/// </structural_toolkit_2015>

```

## 12 SUMMARY

As a result of all these actions, we have a working example for reinforcement concrete. It is fully functional with a limitations described at the beginning of this document. A real implementation will contain more complex functionality; however this example presents how to implement your own code.