

Editor component for physical applications data input

Sergey L. Gladkiy

Introduction

Modern engineering applications allow users to operate with physical values measured with different units of measurements or even select the desirable measurement system. For this purpose there is a mechanism to input the units of measurement. This article describes one way of realization for such mechanism – simple **WPF** editor component for input dimensional physical values and presenting the input data in form close to the physical notation.

Background

In common general case, the units of measurement notation includes unit symbols, like 's' (second) or 'N' (Newton), powers, like 'm²' (meter squared), indexes, like 'mmH₂O' (millimeters of water) and some other notations (https://en.wikipedia.org/wiki/International_System_of_Units). So, the mechanism for handling units of measurement must allow to input all of the notations. One way is to represent the units as string data, like 'mm^2' (millimeter squared), where '^' symbol stays for the power. This approach is simple but the data presentation sometimes is not readable. Another way is creating complicated editor with pop-up menus to select different symbols for input powers, indexes and so on. This approach make the data presentation close to the physical notation, but it is requires complicated programming and is not always user-friendly.

On the other hand, modern programming languages use Unicode character set for string data. The Unicode standard includes subscript and superscript symbols (https://en.wikipedia.org/wiki/Unicode_subscripts_and_superscripts). It is enough to represent units of measurement notation in 'natural physical' form. One problem exists – the subscript and superscript symbols cannot be input with the standard keyboard.

Thus, the main idea of the editor component for the units input is using one of the standard component classes and providing the easy way to input subscript and superscript symbols. This approach combines the following advantages:

- Presenting units of measurement in form, close to the physical notation.
- Simple interface for the user with fast input.
- Easy for programming realization.

Component realization

The component realization is based on the standard **WPF** text editing component – **EditBox**. The main functional algorithm of the new component is: listen the input of the text editor and correct the input to get physical representation of the data. For an example, if the input is 'mm^2', the result text in the editor must be 'mm²'. For realizing this functionality our editor must turn into special state, when we press one of the special symbols (the power '^' symbol in the example), and in this state convert all the symbols into according superscripts or subscripts. We will use the power symbol '^' for the superscripts and the underscore symbol '_' for subscripts.

First, we created a helper class, which has functions to define if the symbol of a string is one of special symbols and for converting a character to the corresponding superscript or subscript. It is enough for the units representation scripting only digits and '+', '-' symbols. The helper class interface is the following:

```

public static class Formatter
{
    public static bool CanBeScripted(char x) [...]
    public static char ToSuperscript(char x) [...]
    public static char ToSubscript(char x) [...]
    public static char FromSuperscript(char x) [...]
    public static char FromSubscript(char x) [...]
    public static int IsScriptSymbol(char x) [...]
    public static char ToScriptSymbol(int sign) [...]
    public static int IsScript(char x) [...]
    public static void ScriptFormatError() [...]
    public static char ToScript(char x, int sign) [...]
    public static char FromScript(char x, int sign) [...]
    public static string ToScripted(string value) [...]
    public static string FromScripted(string scripted) [...]
}

```

The **'CanBeScripted'** method defines is a symbol can be converted to the super- or sub- script:

```

public static bool CanBeScripted(char x)
{
    bool result = char.IsDigit(x) || x == '+' || x == '-' || x == 'e';
    return result;
}

```

The next method, **'ToSuperscript'**, converts a symbol to the corresponding superscript:

```

public static char ToSuperscript(char x)
{
    if (char.IsDigit(x))
    {
        return Symbols.SuperscriptDigits[int.Parse(x.ToString())];
    }
    else
    {
        switch (x)
        {
            case '+': return Symbols.SuperscriptPlus;
            case '-': return Symbols.SuperscriptMinus;
        }
        return x;
    }
}

```

where the **'Symbols'** class defined as:

```
public static class Symbols
{
    public static readonly char UnitExponentSign = '^';
    public static readonly char UnitSubscriptSign = '_';
    public static readonly char UnitFractionSign = '/';
    public static readonly char UnitSeparationSign = ' ';
    public static readonly List<char> SubscriptDigits = new List<char>( new char[] { '0', '1', '2',
'3', '4', '5', '6', '7', '8', '9' });
    public static readonly List<char> SuperscriptDigits = new List<char>( new char[] { '0', '1',
'2', '3', '4', '5', '6', '7', '8', '9' });
    public static readonly char SuperscriptPlus = '+';
    public static readonly char SuperscriptMinus = '-';
    public static readonly char SubscriptPlus = '+';
    public static readonly char SubscriptMinus = '-';
    public static readonly char SubscriptEuler = 'e';
}
```

Analogously, the **'ToSubscript'** method converts a symbol to the corresponding subscript. The next two methods, **'FromSuperscript'** and **'FromSubscript'**, implement the reverse operation – convert the scripted symbol to the original character.

The **'IsScriptSymbol'** method checks if a symbol is one of the special characters:

```
public static int IsScriptSymbol(char x)
{
    if (x == Symbols.UnitExponentSign)
    {
        return 1;
    }
    else
    {
        if (x == Symbols.UnitSubscriptSign)
        {
            return -1;
        }
    }
    return 0;
}
```

The method returns **'1'** for the superscript mode and **'-1'** for the subscript one. There are some other helping functions, the main of them are the last two ones. Let us consider the first of them:

```
public static string ToScripted(string value)
{
    if (string.IsNullOrEmpty(value)) return string.Empty;
    else
    {
        string result = string.Empty;
        int sign = 0;
        for (int i = 0; i < value.Length; i++)
        {
            char x = value[i];
            int s = IsScriptSymbol(x);
```

```

        if (s != 0)
        {
            if (sign != 0 && s == sign)
            {
                ScriptFormatError();
            }
            sign = s;
            continue;
        }
        if (char.IsLetter(x) || (x == Symbols.UnitFractionSign) || (x == ' '))
        {
            sign = 0;
        }
        if (sign != 0)
        {
            x = ToScript(x, sign);
        }
        result += x;
    }
    return result;
}
}

```

The function converts entire string to the corresponding scripted representation. It loops over all the characters in the string and if a character is a special one ('**IsScriptSymbol**'), all the subsequent symbols, that can be scripted, converted to the corresponding scripts, until the unscriptable symbol appeared or another special symbol met. Analogously the '**FromScripted**' method implements the inverse operation. There are some examples of original strings and their scripted representations, got with the conversions:

Original	Scripted
g cm ⁻³	g cm ⁻³
N/m ²	N/m ²
kg m ² A ⁻² s ⁻³	kg m ² A ⁻² s ⁻³

Having this helping class we can now create the editor component. We inherited it from the base **WPF UserControl** class and put into it the standard **TextBox** editor (full XAML code can be found in the download). The class declares some internal data fields:

```

/// <summary>
/// Indexing
/// </summary>
protected internal int indexing;
/// <summary>
/// Not formatted string value
/// </summary>
protected internal string stringvalue = string.Empty;
/// <summary>
/// Formatted string value
/// </summary>
protected internal string formattedvalue = string.Empty;

```

The '**indexing**' field hold the current value for special editing mode (subscript or superscript), '**stringvalue**' is for holding original string value and '**formattedvalue**' is for the scripted representation of the string. There are some other data and functions. The main realization is in defined **EditBox** event handlers. The first is **PreviewTextInput** handler:

```

private void textBoxValue_PreviewTextInput(object sender, TextCompositionEventArgs e)
{
    string s = e.Text;
    int l = s.Length;
    if (l == 1)
    {
        char c = s[0];
        int i = Formatter.IsScriptSymbol(c);
        if (i != 0)
        {
            indexing = i;
            e.Handled = true;
        }
        else
        {
            if (!Formatter.CanBeScripted(c))
            {
                indexing = 0;
            }
        }
    }
}

```

The method intercepts any **TextBox** input and, depending on the character, turn on or off the indexing mode (in which the next input characters converted to the scripted symbols). Another handler is for the 'Text**Changed**' event, here is the part of its code:

```

int index = change.Offset;
char c = textBoxValue.Text[index];
if (c == '*')
{
    c = ' ';
}
InsertCharacter(index, c);

```

where

```

private void InsertCharacter(int index, char c)
{
    int sign = 0;
    if (indexing != 0)
    {
        sign = indexing;
    }
    else
    {
        if (index > 0)
        {
            sign = Formatter.IsScript(FormattedValue[index - 1]);
        }
    }

    if (sign != 0)
    {
        c = Formatter.ToScript(c, sign);
    }
}

```

```

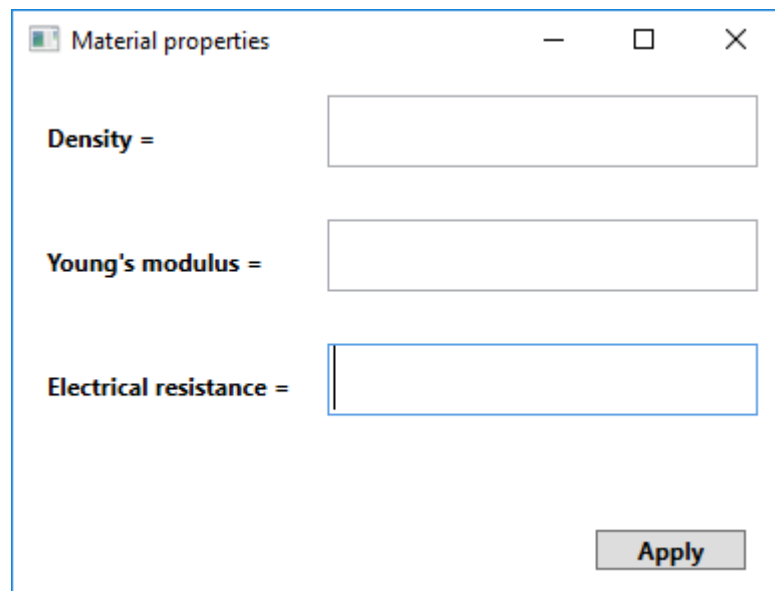
    }
    FormattedValue = FormattedValue.Insert(index, c.ToString());
}

```

Thus, the method replaces any multiplication character by the space (it is close to the natural notation where the multiplication symbol between units just omitted) or inserts input symbol to the formatted string data, taking into account possible conversion to the scripted form. The full source code for the editor component realization is in the download available.

Code example: material properties editor

Now, let us consider an example of the realized component using for creating the application for setting material properties. For simplification, the selected properties are the following three: **density**, **Young's modulus** and the **electrical resistance**. Then, the material property editor application is a simple **WPF** window with three editors of the realized class:



Our goal is to allow user input data for the three material properties and check if the input data is compatible with the required physical quantities (https://en.wikipedia.org/wiki/Physical_quantity). For checking the correctness of the input data and the compatibility of data with physical quantities the **PHYSICS** library used (<http://sergey-l-gladkiy.narod.ru/index/physics/0-14>). The library allows converting string data to the units of measurement with the dimension analysis. The code for the button handler is the following:

```

private void Button_Click(object sender, RoutedEventArgs e)
{
    string sv1 = PhysEdit1.StringValue;
    string sv2 = PhysEdit2.StringValue;
    string sv3 = PhysEdit3.StringValue;
    try
    {
        ScalarValue density = ScalarValue.Parse(sv1);
        ScalarValue young = ScalarValue.Parse(sv2);
    }
}

```

```

        ScalarValue resistance = ScalarValue.Parse(sv3);

        CheckCompatibility(new Density(), density);
        CheckCompatibility(new Stress(), young);
        CheckCompatibility(new ElectricalResistance(), resistance);

        MessageBox.Show("Ok!");
        this.Close();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

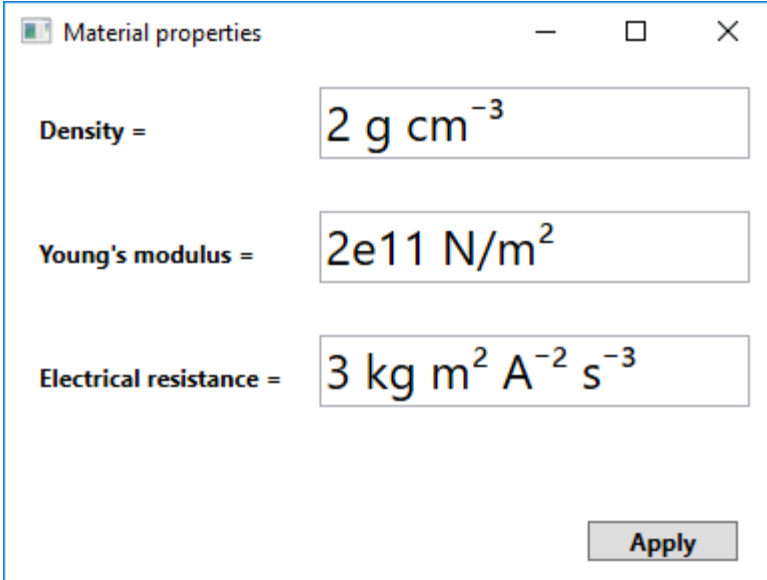
where

```

private static void CheckCompatibility(PhysicalQuantity q, PhysicalValue v)
{
    if (!PhysicalQuantity.Compatible(q, v.Unit))
    {
        throw new ArgumentException("Value " + v.ToString() + " is not compatible with " + q.Name +
        ".");
    }
}

```

The handler just gets the string values from the editors, and, using the PHYSICS library classes, converts them into the physical values and checks their compatibility with the corresponding physical quantities. Note, that the **Young's modulus** compared with the '**Stress**' quantity, because it has the same dimension as the stress physical quantity. Here is an example of correct data input for the material properties:

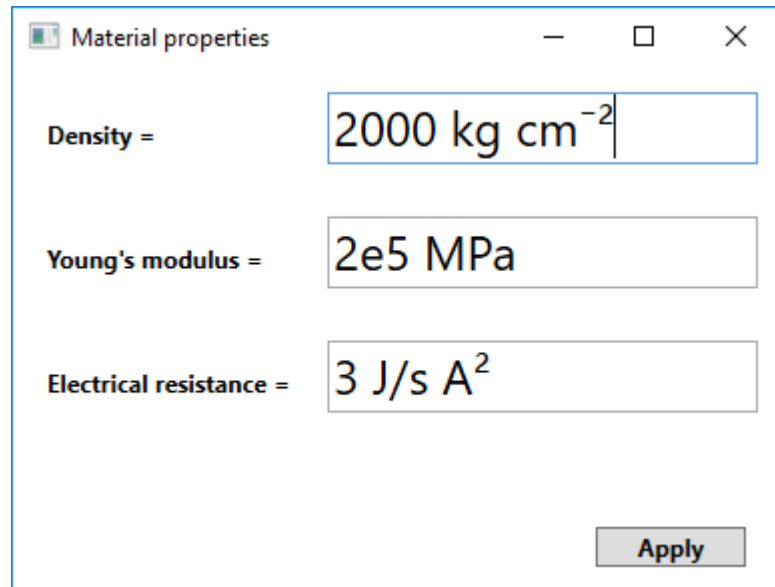


The screenshot shows a window titled "Material properties" with standard Windows window controls (minimize, maximize, close). Inside the window, there are three rows of labels followed by text input boxes:

- Label: "Density =" followed by an input box containing the text "2 g cm⁻³".
- Label: "Young's modulus =" followed by an input box containing the text "2e11 N/m²".
- Label: "Electrical resistance =" followed by an input box containing the text "3 kg m² A⁻² s⁻³".

At the bottom right of the window, there is a button labeled "Apply".

Pressing the button with this input we get '**Ok**' message. But if data is wrong:

A screenshot of a software window titled "Material properties". It contains three input fields for physical properties. The first field is labeled "Density =" and contains the text "2000 kg cm⁻²". The second field is labeled "Young's modulus =" and contains the text "2e5 MPa". The third field is labeled "Electrical resistance =" and contains the text "3 J/s A²". At the bottom right of the window is a button labeled "Apply".

Density =	2000 kg cm ⁻²
Young's modulus =	2e5 MPa
Electrical resistance =	3 J/s A ²

Apply

(the data for density property has wrong dimension), we get the error message:

"Value 2000 kg/cm² is not compatible with Density."

Conclusions

In the article an approach for creating editor component for physical data input considered. The approach is based on the using of Unicode superscript and subscript characters for presenting units of measurement is form, close to the physical notations. The approach allows realize the task of physical values input with minimal code writing. The designed component is easy for user and allows input all the required data with standard keyboard, without using complicated pop-up menus or other **UI** controls.