Decibel units of measurement with C# framework

Sergey L. Gladkiy

Introduction

Units of measurement are widely used in many engineering applications: physical process modeling, signal processing, 3D geometry modeling, geophysics and others. This requires a mechanisms of processing data, measured with different units, including data conversion from one unit of measurement to another. Physical values can be converted from one unit of measurement to another, if the units have the same physical dimension. So, the conversion mechanism must be universal and allow converting any compatible units. In the simplest case the conversion is linear and can be made by multiplying the value with some conversion coefficient. But in general case the conversion can be even nonlinear.

This article introduces an approach of realizing universal framework for converting units of measurement. The approach is based on creating specially designed class hierarchy for providing unified conversion algorithm. The conversion algorithm allows using together linear and nonlinear units of measurement, including logarithmic and decibels.

Background

First, let us state some definitions.

Physical quantity is a property of physical substance that can be measured (http://en.wikipedia.org/wiki/Physical_quantity). Examples of physical quantities are: mass, time, electric current and so on. There are **fundamental** (base) physical quantities called **dimensions**: **length, time, mass, temperature, electric current, amount of substance, luminous intensity**. They are fundamental because the sense of all other quantities can be expressed via them. For an example, physical quantity **velocity** can be expressed via **length** and **time**: **velocity = length/time**. So, any physical quantity has the same characteristic called **physical dimension**. Velocity quantity has dimension **length/time**, acceleration has dimension **length/time^2**, **area** quantity has dimension l**ength^2** and so on.

Unit (**of measurement**) is a definite magnitude of a physical quantity (http://en.wikipedia.org/wiki/Unit_of_measurement). For an example, **meter** is a predefined magnitude of **length** physical quantity. So, as a physical quantity, every unit has such attribute as physical dimension. Physical quantity can be measured by some unit **if and only if they have equal dimensions**. There is 'many to many' relation between physical quantities and units. That is, one quantity can be measured by many units, and one unit can measure many quantities. For an example, mass can be measured with grams, pounds, kilograms and so on, as hydrostatic pressure and mechanical stress can be measured with one unit – **Pascal**.

The algorithm of unit conversion depends on the definitions of the units (https://en.wikipedia.org/wiki/Units_of_measurement#Expressing_a_physical_value_in_terms_of_another_unit).

For an example, to convert temperature **t** from degrees of **Celsius** to degrees of **Fahrenheit** the following equation is used **t °C = 5/9 (t °F - 32)**, and area conversion from meters squared to centimeters squared defined by another equation **1 m^2 = 1 m · 1 m = 100 cm · 100 cm = 10000 cm^2**.

Even if the conversion algorithm can be easily defined by a human in every separate case, it is not so easy to write an universal algorithm for all cases. There are some common approaches used for realizing the algorithm.

The first and simplest approach is creating a set of all units (classes) and define for any of them the methods for converting to any other compatible units. For an example, class Meter with methods ToInch, ToFoot and so on; class Foot with methods ToMeter, ToInch and so on; and other classes. The algorithm is universal – it can convert any compatible units. But its drawbacks are clear: for introducing new units we must make a lot of work, because the number of classes can be really high and the number of methods is proportional to the **square of the compatible units' number**. Moreover, for introducing new units we must change the existing ones. Nevertheless, there are unit conversion frameworks those use this approach.

Another approach is ignoring some special units, for an example nonlinear, and reducing the algorithm to the multiplication by some factor. This approach is much easier than the first one, but it is not universal.

Logarithmic units, and, as a particular case, the decibels (https://en.wikipedia.org/wiki/Decibel), play important role in many measurement systems. It can be proved by the fact that almost all human senses are 'logarithmic' by the nature (https://en.wikipedia.org/wiki/Weber%E2%80%93Fechner_law#Fechner's_law). So, ignoring the nonlinear units is a big drawback of the conversion algorithm.

Thus, we can state general requirements for the unit conversion framework:

- Unit conversion algorithm must be universal and support linear and nonlinear units of measurement.
- Number of new (overridden) methods must linearly depends on the number of new introduced units.
- Introducing new units must NOT change the existing class hierarchy and core (conversion) algorithms.
- Minimal code writing for introducing new units.

Realization

The suggested approach of unit conversion system realization is based on a specially designed class hierarchy. First, we introduce the base abstract class for unit of measurement:

```
/// <summary>
/// The base abstract class for all units
/// </summary>
public abstract class Unit
{
    /// <summary>
    /// The name of the unit.
    /// </summary>
    public string Name
    {
        get { return GetName(); }
    }


    /// <summary>
    /// The symbol of the unit.
    /// </summary>
    public string Symbol
    {
        get { return GetSymbol(); }
    }

    /// <summary>
    /// The Dimension of the unit.
    /// </summary>
    public Dimension Dimension
    {
```

```
        get { return GetDimension(); }
    }
}
```

The class introduces common properties for using in the program code: ***Name*** – name of the unit (Pascal, Newton and so on); ***Symbol*** – symbol of the unit, used as its notation (**Pa** for **Pascal**, **N** for **Newton** and so on); ***Dimension*** – physical dimension of the unit, used for defining unit compatibility.
The class must also provide the functionality for unit conversion and it must be implemented following the requirements stated above. To satisfy the requirement of linear number of methods overriding for new units, the following approach suggested: we select some base unit system, namely **SI**, (https://en.wikipedia.org/wiki/International_System_of_Units) and define any conversion with two steps – first we convert the unit to the base one and then convert it to the required unit. This implemented with adding the following code to the class:

```
/// <summary>
/// Direct conversion to some base unit.
/// </summary>
public abstract double Direct(double value);

/// <summary>
/// Inverse conversion from some base unit.
/// </summary>
public abstract double Inverse(double __base);


/// <summary>
/// Converts 'value' from '_from' unit to '_to' one.
/// </summary>
/// <returns>Converted value</returns>
public static double Convert(Unit _from, Unit _to, double value)
{
    double result = _to.Inverse(_from.Direct(value));
    return result;
}
```

Thus, the final conversion method ***Convert*** is universal and can be applied for all units (linear and nonlinear) and for defining new units we must override (maximum) two methods for one new class – linear dependency on the number of classes. As will be shown below, the number of overridden methods for new class of final, realized unit can be even reduced to one (except the ***Name*** and ***Symbol*** methods) due to further intermediate class hierarchy.
The base abstract class also includes some other base functionality, like operator overloading (for using math operations with units in program) and other, those are skipped here. We also skip the inheritance branch for linear and other derived units (the information can be found in the guide for the **PHYSICS** framework http://sergey-l-gladkiy.narod.ru/phy_docs/PHYSICS_C_manual.en.pdf). And we are going to realize the decibel units.
It should be noted here that decibel (https://en.wikipedia.org/wiki/Decibel) is a particular case of logarithmic units (https://en.wikipedia.org/wiki/Logarithmic_units) with the base 10. Logarithmic unit conversion, in general case, described with the following formula:

$$Y = F \cdot \log_B \left(\frac{X}{R} \right)$$

where **Y** – value, measured with logarithmic unit; **X** – value, measured with some base unit; **B** – base of the logarithm; **R** – reference value (logarithmic unit expresses the measured value relative to this reference value); **F** – multiplication factor value.

Thus, the base abstract class for logarithmic units can be the following:

```
/// <summary>
/// Logarithmic Unit
/// General formula: Y = F*log{B}(X/R)
/// Where: Y - value in logarithmic units,
///        X - value in units for conversion,
///        R - reference value,
///        B - logarithm base,
///        F - multiplication factor.
/// </summary>
public abstract class LogarithmicUnit: NonlinearUnit
{
    /// <summary>
    /// Base of the Logarithm
    /// </summary>
    public double Base
    {
        get { return GetBase(); }
    }

    /// <summary>
    /// Reference value
    /// </summary>
    public double Reference
    {
        get { return GetReference(); }
    }

    /// <summary>
    /// Factor value
    /// </summary>
    public double Factor
    {
        get { return GetFactor(); }
    }

    /// <summary>
    /// Direct conversion to some base unit.
    /// X = R*B^(Y/F)
    /// </summary>
    public override double Direct(double value)
    {
        double result = Reference*Math.Pow(Base, value/Factor);
        return result;
    }

    /// <summary>
    /// Inverse conversion from some base unit.
    /// Y = F*log{B}(X/R)
    /// </summary>
    public override double Inverse(double __base)
    {
        double result = Factor*Math.Log(__base/Reference, Base);
        return result;
    }
```

```
}
```

The class introduces base properties for describing logarithmic conversion: ***Base***, ***Reference***, and ***Factor***. The conversion methods ***Inverse*** and ***Direct*** implement the algorithm (the former realizes the formula above and the latter is the inverse of the formula).

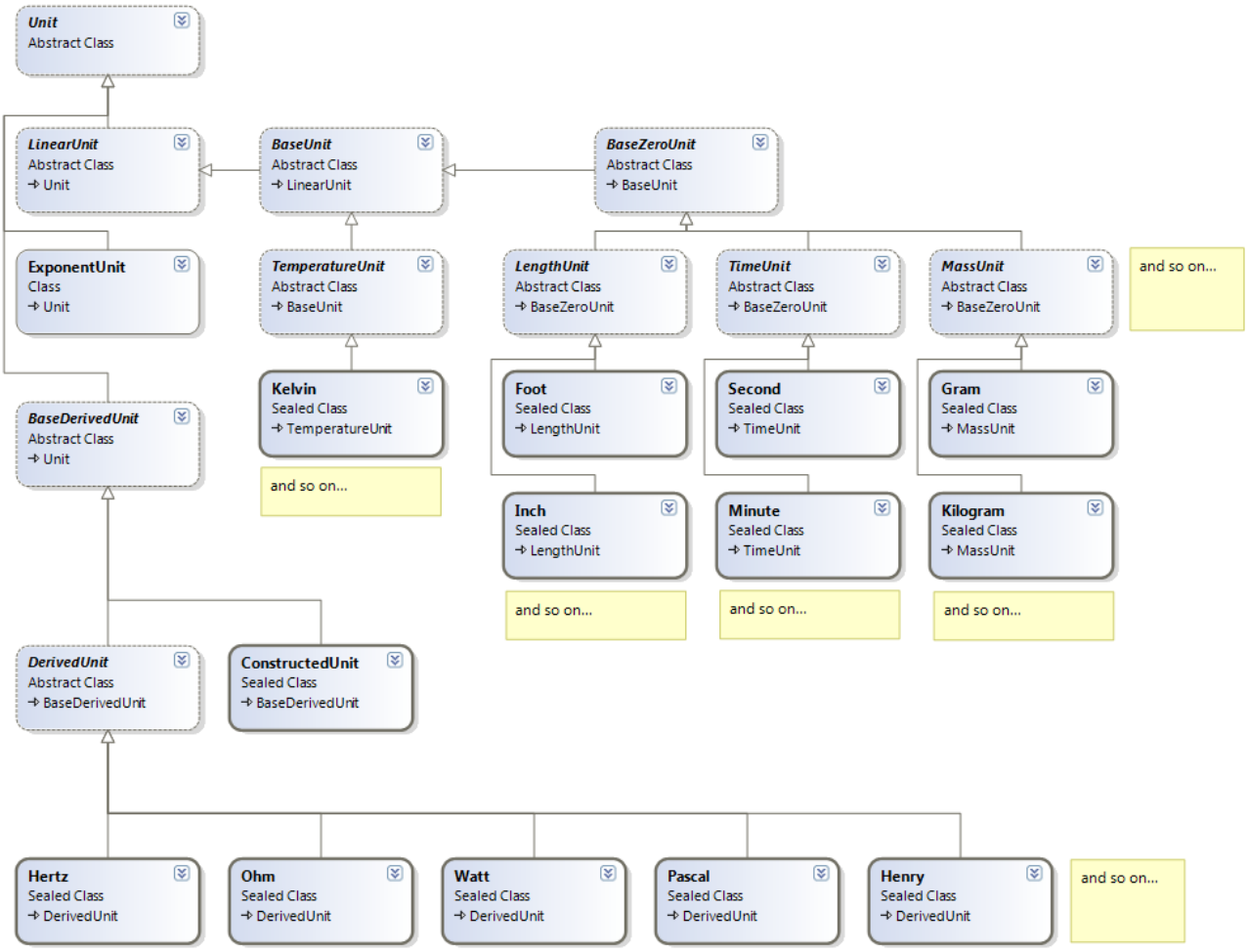
Having got the base class for logarithmic units we can extend the branch according to the existing physical concepts. As stated above, logarithmic units express the value, relative to another one, so, they are **dimensionless** and reference value is 1.0. But for measure dimensioned we need dimensioned logarithmic units and we can introduce the base abstract class for such units – ***DimensionedLogarithmic***, which contains the ***ReferenceUnit*** property. Then all logarithmic units can be divided by the logarithm base. Common logarithm base values are **2**, **e** and **10** and we inherit the following classes for them: ***BinaryUnit***, ***NaturalUnit***, ***DecimalUnit***, ***BinaryDimensioned***, ***NaturalDimensioned*** and ***DecimalDimensioned***. As decibel units have the base 10 we inherit base classes ***DecibelBase*** and ***DecibelDimensioned*** for dimensionless and dimensioned decibels. The dimensionless decibel has reference value 1.0, and the dimensioned decibel reference calculated with the ***ReferenceUnit*** value. And finally, dimensioned decibels can express relations for 'power' and 'field' physical quantities (https://en.wikipedia.org/wiki/Decibel#Power_quantities). It results in the multiplication ***Factor*** value being 10 and 20 respectively. The inherited classes for the two units are ***DecibelDimensioned10*** and ***DecibelDimensioned20***. Here is the code for described class hierarchy:

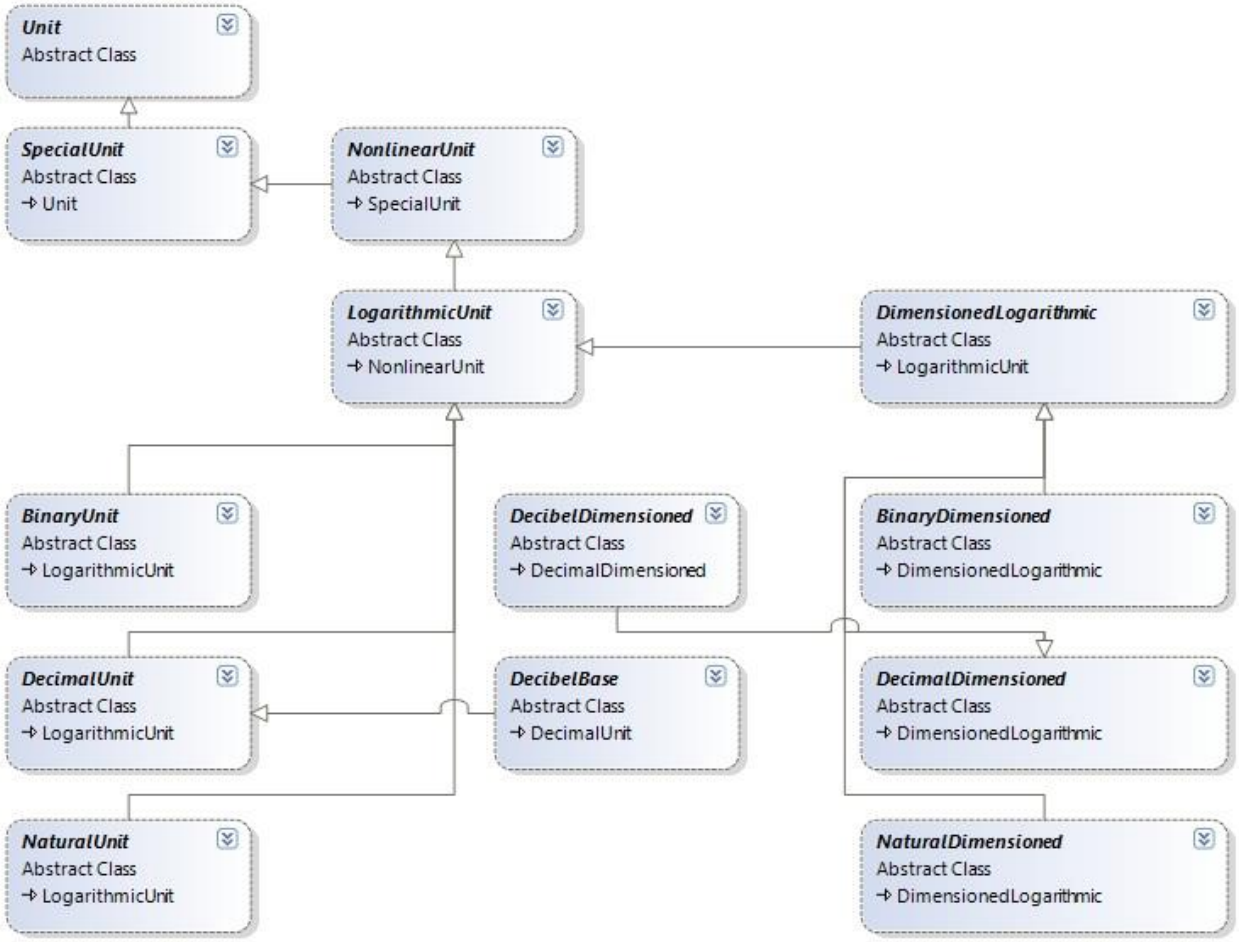
```
/// <summary>
/// Special class for DIMENSIONED logarithmic units.
/// </summary>
public abstract class DimensionedLogarithmic : LogarithmicUnit
{
    /// <summary>
    /// Reference unit - the relative physical value measured with.
    /// </summary>
    public Unit ReferenceUnit
    {
        get { return GetReferenceUnit(); }
    }
}


/// <summary>
/// Logarithmic dimensionless unit to base 2
/// </summary>
public abstract class BinaryUnit: LogarithmicUnit
{
    protected sealed override double GetBase()
    {
        return 2.0;
    }
}

/// <summary>
/// Logarithmic dimensionless unit to base e
/// </summary>
public abstract class NaturalUnit : LogarithmicUnit
{
    protected sealed override double GetBase()
    {
```

```
        return Math.E;
    }
}

/// <summary>
/// Logarithmic dimensionless unit to base 10
/// </summary>
public abstract class DecimalUnit: LogarithmicUnit
{
    protected sealed override double GetBase()
    {
        return 10.0;
    }
}


/// <summary>
/// Base dimensionless decibel unit (decimal logarithmic unit)
/// </summary>
public abstract class DecibelBase: DecimalUnit
{
    /// <summary>
    /// For all dimensionless decibels Reference = 1.0
    /// </summary>
    /// <returns></returns>
    protected override double GetReference()
    {
        return 1.0;
    }
}


/// <summary>
/// Logarithmic dimensioned unit to base 2
/// </summary>
public abstract class BinaryDimensioned : DimensionedLogarithmic
{
    protected sealed override double GetBase()
    {
        return 2.0;
    }
}

/// <summary>
/// Logarithmic dimensioned unit to base e
/// </summary>
public abstract class NaturalDimensioned : DimensionedLogarithmic
{
    protected sealed override double GetBase()
    {
        return Math.E;
    }
}

/// <summary>
/// Logarithmic dimensioned unit to base 10
/// </summary>
public abstract class DecimalDimensioned : DimensionedLogarithmic
{
    protected sealed override double GetBase()
```

```
    {
        return 10.0;
    }
}
/// <summary>
/// Base dimensioned decibel unit
/// </summary>
public abstract class DecibelDimensioned : DecimalDimensioned
{
}

/// <summary>
/// Dimensioned Decibel with Factor = 10
/// </summary>
public abstract class DecibelDimensioned10 : DecibelDimensioned
{
    protected override double GetFactor()
    {
        return 10.0;
    }
}

/// <summary>
/// Dimensioned Decibel with Factor = 20
/// </summary>
public abstract class DecibelDimensioned20 : DecibelDimensioned
{
    protected override double GetFactor()
    {
        return 20.0;
    }
}
```

Now we have the class hierarchy for creating final, realized classes for concrete decibel units. But before going to this, let us show the total diagram for class hierarchy, including skipped ‘linear’ units branch (the diagrams for the two branches shown on separate pictures).


Unit
Abstract Class
LinearUnit
Abstract Class
→ Unit
BaseUnit
Abstract Class
→ LinearUnit
BaseZeroUnit
Abstract Class
→ BaseUnit
ExponentUnit
Class
→ Unit
TemperatureUnit
Abstract Class
→ BaseUnit
LengthUnit
Abstract Class
→ BaseZeroUnit
TimeUnit
Abstract Class
→ BaseZeroUnit
MassUnit
Abstract Class
→ BaseZeroUnit
and so on...
Kelvin
Sealed Class
→ TemperatureUnit
and so on...
Foot
Sealed Class
→ LengthUnit
Second
Sealed Class
→ TimeUnit
Gram
Sealed Class
→ MassUnit
BaseDerivedUnit
Abstract Class
→ Unit
Inch
Sealed Class
→ LengthUnit
Minute
Sealed Class
→ TimeUnit
Kilogram
Sealed Class
→ MassUnit
and so on...
and so on...
and so on...
DerivedUnit
Abstract Class
→ BaseDerivedUnit
ConstructedUnit
Sealed Class
→ BaseDerivedUnit
Hertz
Sealed Class
→ DerivedUnit
Ohm
Sealed Class
→ DerivedUnit
Watt
Sealed Class
→ DerivedUnit
Pascal
Sealed Class
→ DerivedUnit
Henry
Sealed Class
→ DerivedUnit
and so on...



The class hierarchy seems to be very complicated. But the main aim of the hierarchy is providing universal algorithm for handling any units of measurement. Once having created, the hierarchy is never going to change. The only thing to do for the developer, using the framework, is adding new units of measurement. But due to the strongly structured class hierarchy this task is simple and requires minimal code writing. Let us consider the code for one dimensioned decibel unit of measurement – **Decibel-Voltage** (https://en.wikipedia.org/wiki/Decibel#Voltage):

```
/// <summary>
/// Decibel - Voltage
/// </summary>
public sealed class DecibelVoltage : DecibelDimensioned20
{
    /// <summary>
    /// Reference unit - 1 Volt
    /// </summary>
    /// <returns></returns>
    protected override Unit CreateReferenceUnit()
    {
        Unit r = new Volt();

        return r;
    }

    protected override string GetName()
```

```
    {
        return "Decibel-Volts";
    }

    protected override string GetSymbol()
    {
        return "dBV";
    }
}
```

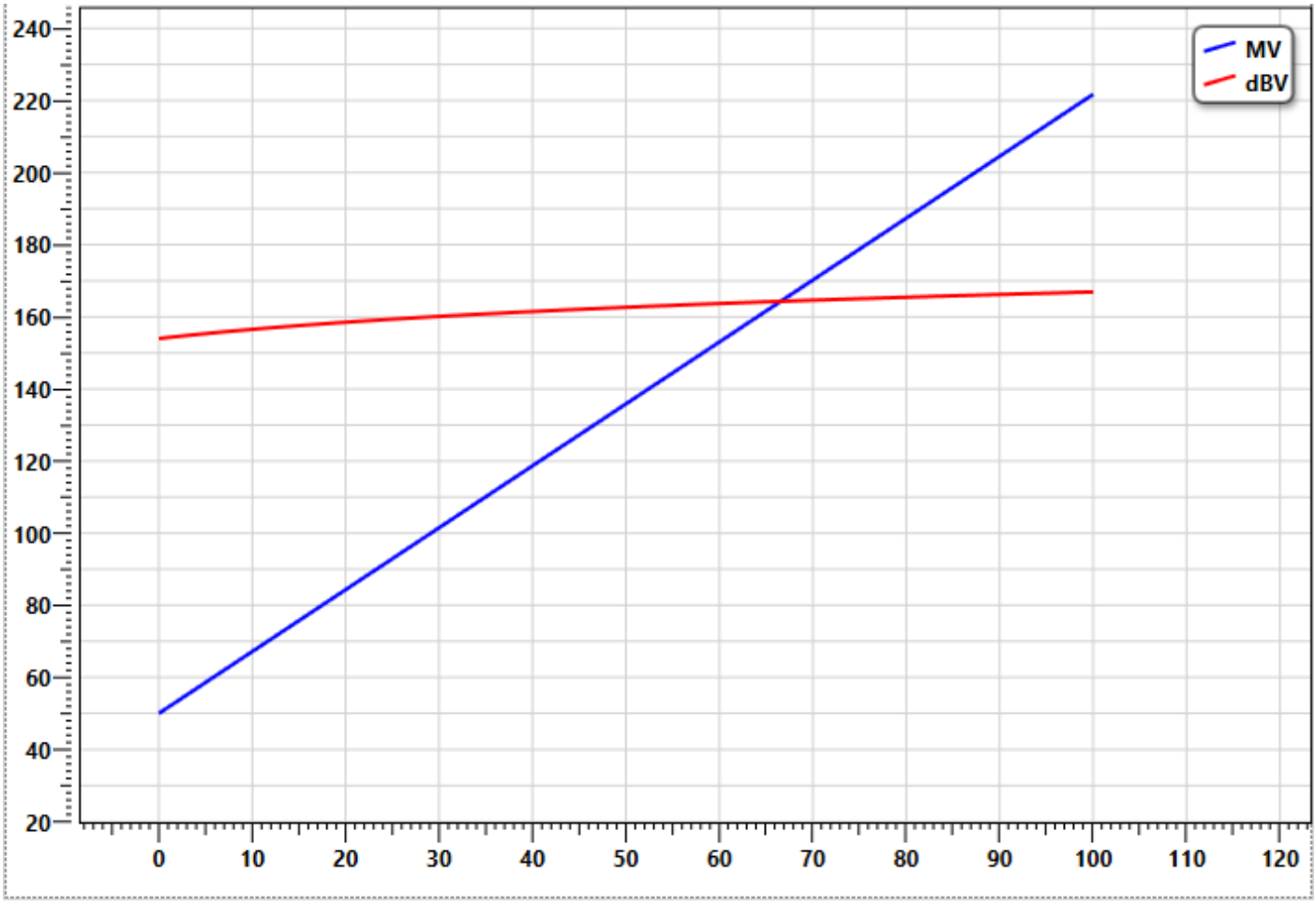
The example of the **Decibel-Voltage** unit shows that introducing new unit requires only one method overriding, except the methods for providing name and symbol. All other functionality for converting units realized in the described class hierarchy. After introducing the unit it can be used in the conversions and the framework will automatically provide right conversion algorithm for it.

Using the code

Now, we are going to consider a simple example of using decibel units. Let we have in the program an array of double values, measured in **megavolts**. The task is to convert them into the **Decibel-Voltage** unit. With the **PHYSICS** framework the code for solving the task is:

```
double[] data = ...;

UnitConverter converter = new UnitConverter();
Unit u1 = converter.ConvertString("MV");
Unit u2 = converter.ConvertString("dBV");

int n = data.Length;
double[] dbdata = new double[n];
for (int i = 0; i < n; i++)
{
    dbdata[i] = Unit.Convert(u1, u2, data[i]);
}

// Using the data...
```

First we created the ***UnitConverter*** instance (the class is intended to convert string unit representation to units of measurements and vice versa). Then we created two units of measurements by their string representation (symbols) and used in the loop the method ***Convert*** of the ***Unit*** class to get new values, measured with **Decibel-Voltage**. The result data shown on the picture below:



As can be seen from the picture, linear values measured in megavolts are not linear (logarithmic) if they measured in decibel-volts, as it is expected. The code for implementing the conversion is simple and does not depend on the units used. The introduced **Decibel-Volts** unit can be used in conversions with any other compatible units, linear and nonlinear.

The source code of the example (VS 2010 solution) can be found in the download for the article.

Conclusions

This article introduced an approach of realizing nonlinear and decibel units in C# physics framework. The approach is based on creating specially designed class hierarchy and allows using universal conversion algorithm for linear and nonlinear units. New units of measurement can be introduced in the framework by creating inherited class with minimal method overriding and without modifying core conversion algorithm and existing unit classes. The **PHYSICS** framework can be found here http://sergey-l-gladkiy.narod.ru/index/physics/0-14.