Using physical quantities and units of measurement in C# programs

Sergey L. Gladkiy

Introduction

Using physical quantities and units of measurement is very important for developing scientific and engineering programs. By nature, these programs deal with physical quantities. Physical quantities in other hand measured in units. Moreover, different quantities have different dimensions and must be measured in units compatible with these physical dimensions. For an example, speed physical quantity has dimension L/T (length/time), so it can be measured by m/s (meter per second), but may not be measured in m/g (meter per gram), because the last has dimension L/M (length/mass). Also, one physical quantity can be measured in different units. For an example, speed can also be measured in ft/h (foot per hour).

However, when developer creates a program he uses just numbers to represent values of physical quantities, so there is no unit of measurement associated with number. This can lead to various errors in program code:

- Sum or subtract data, measured in different units.
- Sum or subtract data with different dimensions.

So, there are two issues we must keep in mind when dealing with physical quantities in programs: what dimension of the data is, for checking if the data compatible for some operation, and how to convert values to the same units of measurement to make this operation correct.

This article suggests one approach of working with physical data in C# programs. The solutions is to create special classes which encapsulate both - numerical values and units of measurement in one instance and implement all operations (via operator overloading and methods) taking into account dimensions and units of data. Using these classes allows easily make operations with physical data values in C# with dimension safety and without complicated code. In addition, there is the possibility to allow user input data with units of measurement in text format.

Background

First, let us state some definitions that will be used in the article, because there can be sometimes different understanding of these concepts.

Physical quantity is a property of physical substance that can be measured. Examples of physical quantities: 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***. The velocity quantity has dimension length/time, acceleration has dimension length/time^2, area has dimension length^2 and so on.

Unit (of measurement) is a definite magnitude of a physical quantity. 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 by one unit – Pascal.

Physical value is a value of a physical quantity which is represented by numerical value and unit the value is measured with. For an example, 5 gram is a physical value of mass physical quantity. Algebraic operations can be implemented with physical values. For an example, we can add two physical values (if their units are compatible with each other) 2 kg + 5 g = 2005 g, or to multiply two values 2 m · 3 s^-2 = 6 m/s^2.

So, there is the difference between these concepts of physical quantity and physical value. Physical quantity is an abstract concept, while physical value is the value of concrete phenomenon property.
The main purpose of this project is to implement an approach do deal with physical values in C# and do it easily and in the right way.

Realization

Let us consider the realization of physical value class. The class must encapsulate numerical value and unit of measurement. Here the general case of physical value supposed, that is numerical value can be scalar, vector, matrix and so on. Therefore, base abstract class for physical value contains only unit of measurement and also abstract methods for handling numerical values dedicated to descendant classes. The first part of realization:

```
/// <summary>
/// Physical Value = Value + Unit (of measurement)
/// Type of Value defined in descendant classes
/// </summary>
public abstract class PhysicalValue
{
    /// <summary>
    /// Unit field
    /// </summary>
    protected Unit unit;

    /// <summary>
    /// Unit of the value
    /// </summary>
    public virtual Unit Unit
    {
       get { return unit; }
       set { SetUnit(value); }
    }

    /// <summary>
    /// Sets new unit and converts value from old to new unit
    /// </summary>
    /// <param name="value"></param>
    protected virtual void SetUnit(Unit value)
    {
        if (unit != null)
        {
            if (Unit.Convertible(unit, value))
            {
                Unit from = unit;
                unit = value;
                ConvertValue(from, value);
            }
        }
        else
        {
            unit = value;
        }
    }

    /// <summary>
    /// Converts value from old to new unit
    /// </summary>
    /// <param name="_from"></param>
    /// <param name="_to"></param>
    protected abstract void ConvertValue(Unit _from, Unit _to);
```

This part contains realization of only one property ***Unit*** (unit of measurement of the physical value). The main feature of the property is that when new value is assigned, it converts the numerical value to the new unit of measurement. This feature realized using abstract method ***ConvertValue*** which will be realized in descendant classes (because it depends on the type of value). The property is of type ***Unit*** – base class for units of measurement. This article does not concern the unit conversion and uses ready to use library PHYSICS containing a lot of predefined unit classes.

The rest part of the class contains realization of conversion to string to make possibility of result output and printing. It is not interesting for the purpose of the article and can be seen in provided source code.

Let us consider the final realization of concrete physical value class, based on this abstract class. And it is scalar physical value – it encapsulates unit of measurement and ***double*** value as numerical property. The first part of implementation is the following:

```
/// <summary>
/// Scalar value (double value + unit)
/// </summary>
public sealed class ScalarValue : PhysicalValue
{
    /// <summary>
    /// Value field
    /// </summary>
    private double value;

    protected override void ConvertValue(Unit _from, Unit _to)
    {
        value = Unit.Convert(_from, _to, value);
    }

    /// <summary>
    /// Value
    /// </summary>
    public double Value
    {
        get{ return value; }
        set{ this.value = value; }
    }

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="aUnit">Unit</param>
    /// <param name="aValue">Value</param>
    public ScalarValue(Unit aUnit, double aValue): base(aUnit)
    {
        value = aValue;
    }
```

The realization is rather simple. First, it contains field and property ***Value*** for “numerical” part of scalar physical value. Also it overrides ***ConvertValue*** method and uses static ***Unit.Convert*** for double value conversion. And finally, there is the constructor with to parameters for creation the values and providing together the unit of measurement and the numerical value.

The next part of the class again contains realization of ***ToString*** and ***Parse*** methods and is skipped here. And the final part is the main functionality to use – operator overloading for making operations with physical values. As an example the addition operator’s code presented:

```
    /// <summary>
    /// Operator +
```

```
        /// </summary>
        public static ScalarValue operator +(ScalarValue x, ScalarValue y)
        {
            ScalarValue result = null;

            if (Unit.Convertible(x.Unit, y.Unit))
            {
                Unit u = x.Unit;
                double f1 = x.Value;
                double f2 = Unit.Convert(y.Unit, u, y.Value);

                result = new ScalarValue(u, f1 + f2);
            }

            return result;
        }
```

The implementation takes into account two rules of physical values summation: only physical values with the same dimension may be added and operation with numerical values must be done for values being measured in the same units. In the code above the first rule is guaranteed by checking that units are convertible (have the same physical dimension) with ***Unit.Convertible*** method. Then numerical value of the second operand converted to the unit of the first one with ***Unit.Convert*** method. This conversion guaranteed that when numerical values added they are measured in the same unit so the sum operation made correct.
Let us consider the implementation of one more operation with physical values – product.

```
        /// <summary>
        /// Operator *
        /// </summary>
        public static ScalarValue operator *(ScalarValue x, ScalarValue y)
        {
            Unit u = x.Unit * y.Unit;

            ScalarValue result = new ScalarValue(u, x.Value * y.Value);

            return result;
        }
```

The implementation differs from the addition operator because the rules of the operation differ too. Product of the physical values does not require the values being measured in the same units. Opposite, physical values measured in any units can be multiplied by multiplying their units and numerical values (physical sense of got values is not taken into account).
Analogously, following the rules for operations with physical values, many other operators overloaded: subtraction of physical values, negation of physical value, division of two physical values, multiplication and division of physical value and real number, power of physical value (integer power, C# operator '^'), comparison and equality operators.

Code examples

For better understanding how written classes can be used and what possibilities these features bring in, let us view some examples of code. The first one demonstrates summation of two physical values:

```
ScalarValue m1 = ScalarValue.Parse("2.5 kg");
ScalarValue m2 = ScalarValue.Parse("250 g");
ScalarValue m = m1 + m2;

Console.Out.WriteLine("m = " + m1 + " + " + m2 + " = "  + m.ToString());
```

Here, two physical values of quantity ‘mass’ created with ***ScalarValue.Parse*** method. The values measured in different units: kilograms and grams. But result sum is correctly calculated, taking into account unit conversion. The output to the console (console used here for example only and result can be used in further code) is the following:

```
m = 2.5 kg + 250 g = 2.75 kg
```

Next code example demonstrates another overloaded operator – multiplication:

```
ScalarValue a = ScalarValue.Parse("9.8 m/s^2");
ScalarValue m = ScalarValue.Parse("70.5 kg");
ScalarValue F = m * a;

Console.Out.WriteLine("F = " + m + " * " + a + " = " + F.ToString());
```

And console output is:

```
F = 70.5 kg * 9.8 m/s^2 = 690.9 kg m/s^2
```

In the last example, two physical values created: one is value for ‘acceleration’ physical quantity and another for ‘mass’. When the values multiplied, new physical value created for ‘force’ physical quantity (Newton’s second law). As can be seen from output, result values has correct physical dimension as follows from the equation.
It should be noted that in all examples input data was ‘hardcoded’ as string literals. But realization of physical values is not restricted with this case only. Input data can be provided by user via GUI or can be the result of previous calculations, can be read from TXT or XML files and so on. The only important thing to see from these examples is that such realization allows easily manipulating physical values in natural form (like real numbers) and with safety (rules of physical values operations implemented inside the classes and hidden under simple interface).

Vector values realization

As was said above, the implementation of physical value classes supposes realization of other value types – vector, matrix and so on. Such structure of class hierarchy follows directly from the area of interest, namely physics. Physics deals not only with scalar values, described by one real number, but with more complicated physical properties, described by more complicated mathematical objects – vectors, tensors and others. Let us consider realization of vector physical value class that is descendant of base class described above. Here is the first code part:

```
/// <summary>
/// Vector value (vector + unit)
/// </summary>
public sealed class VectorValue : PhysicalValue
{
    /// <summary>
    /// Value field
    /// </summary>
    private Vector3D value;

    /// <summary>
    /// Vector
    /// </summary>
    public Vector3D Value
    {
        get { return value; }
        set { this.value = value; }
    }
```

```
protected override void ConvertValue(Unit _from, Unit _to)
{
    value.X1 = Unit.Convert(_from, _to, value.X1);
    value.X2 = Unit.Convert(_from, _to, value.X2);
    value.X3 = Unit.Convert(_from, _to, value.X3);
}

/// <summary>
/// Constructor
/// </summary>
public VectorValue(Unit aUnit, Vector3D aVector): base(aUnit)
{
    value = aVector;
}

/// <summary>
/// Component of the Vector value is Scalar value.
/// NOTE: index = 1..3
/// </summary>
public ScalarValue this[int index]
{
    get
    {
        return new ScalarValue(unit, value[index]);
    }
    set
    {
        double v = Unit.Convert(value.Unit, unit, value.Value);
        this.value[index] = v;
    }
}
```

First the class implements field and property ***Value*** for encapsulating 3D vector data. Then it overrides base method ***ConvertValue*** for implementation of 3D vector conversion when the unit property changed. Also it realizes constructor with unit and vector parameters and indexing property for accessing vector's components (note that physical vector component is scalar physical value).
There is also code part for string conversion, it is omitted. And the last part is operators overloading realization. As examples, code for subtract operator and dot product method given:

```
    /// <summary>
    /// Operator -
    /// </summary>
    public static VectorValue operator -(VectorValue x, VectorValue y)
    {
        VectorValue result = null;

        if (Unit.Convertible(x.Unit, y.Unit))
        {
            Unit u = x.Unit;
            VectorValue v = new VectorValue(y.Unit, y.Value);
            v.Unit = u; // value conversion here
            Vector3D f1 = x.Value;
            Vector3D f2 = v.Value;

            result = new VectorValue(u, f1 - f2);
        }

        return result;
    }

    /// <summary>
```

```
        /// Dot product
        /// </summary>
        public static ScalarValue Dot(VectorValue v1, VectorValue v2)
        {
              Unit u = v1.Unit*v2.Unit;
              double r = Vector3D.Dot(v1.Value, v2.Value);
              return new ScalarValue(u, r);
        }
```

For the subtraction operation there is the check for units of the first and the second values being convertible (having the same physical dimension). Then the vector data of the second value converted to the unit of the first one because operation required values being measured in the same unit. Conversion made by implicit method – a copy of vector value made and then it's unit property assigned to new value. It forced the copied vector value making conversion. Finally, the result values calculated using overloaded subtraction operator for 3D values.

The method for dot product operation need not check of units being the same because of the nature of the operation (for an example, dot product of force and displacement is mechanical work https://en.wikipedia.org/wiki/Dot_product). The return type of the operation is ***ScalarValue*** class because dot product of vectors is scalar.

Vector values code example

As an example of using physical vector values in C# programs here is the solution of the following problem: calculate the work W produced by force F acting on a body while it moved consequentially along two strait displacement vectors s1 and s2. Code for solution:

```
VectorValue s1 = VectorValue.Parse("(0 1 0) m");
VectorValue s2 = VectorValue.Parse("(20.3 0 0) ft");
VectorValue F = VectorValue.Parse("(2 1 -1) N");
ScalarValue W = VectorValue.Dot(F, s1+s2);

Console.Out.WriteLine("W = " + W.ToString());
```

Console output:

```
W = 13.37488 N m
```

As can be seen from the code above, displacement vectors set up in different units of measurement – meters and feet. It is not difficult for realized system – it automatically converts second value to meters. Force vector set up in Newtons, so the result value measured in Newton-meters as expected for work physical quantity.

Conclusions

The article describes one approach and realization of physical values in C#. Working with dimensional data in programming code must be implemented carefully, keeping in mind two important things: when making operations with data their dimensions must satisfy the rules applied for the operations; conversion algorithm must be applied to numerical data before operation for correct calculations. Suggested approach provides easy way to forget about these complicated things and do all operations like with common real numbers. Implemented classes encapsulate all operation logic inside simple programming interface and provide safety of the operations. Suggested class hierarchy allows realizing physical values of different types – scalar, vector, matrix and so on. Implementation of new physical value classes implies overriding common abstract methods overriding and realization of new methods specific for

this physical value class. There are realized classes for scalar, 3D vector and tensor physical values. The approach can be easily extended to other physical values.