PHYSICS C# Development Library Version 5.0

Manual

Email: lrndlrnd@mail.ru
URL: www.Sergey-L-Gladkiy.narod.ru

Contents

Introduction

PHYSICS is a C# library for developers. PHYSICS library contains special classes and allows using various physics concepts (such as physical quantities, units of measurement and so on) in .NET programs.

ADVANTAGES of PHYSICS library:

1. 100% C# code.
2. Strongly structured class hierarchy (as close as possible to modern physics concepts).
3. Universal algorithms for working with physics concepts (no need to modify primary code).
4. Many predefined physical entities (physical quantities, units of measurement).
5. Easy to introduce new physics concepts (physical quantities, units of measurement and so on).
6. Easy to localize physical entities.
7. Working with physical values – scalar, vector, tensor.
8. Logarithmic units and decibels supported.

Dependences

PHYSICS C# library depends on:

1. NRTE (NET Run-Time Environment) library.

Extensions

There are the following extensions of PHYSICS C# library:

1. Measurement (realizes many derived units and quantities).
2. Mathphysics (realizes scalar, vector and tensor physical values).

1. Physical quantities, units and values

Physical quantities

Physical quantity is a property of physical substance that can be measured (http://en.wikipedia.org/wiki/Physical quantity). Examples of physical quantities: mass, time, electric current and so on. Do not confuse concepts of physical quantity and the physical (quantity) value. Physical quantity is an abstract concept, and physical quantity value is the value of concrete phenomenon property. For an example, 5 grams is a physical value of physical quantity mass (about physical values see later).

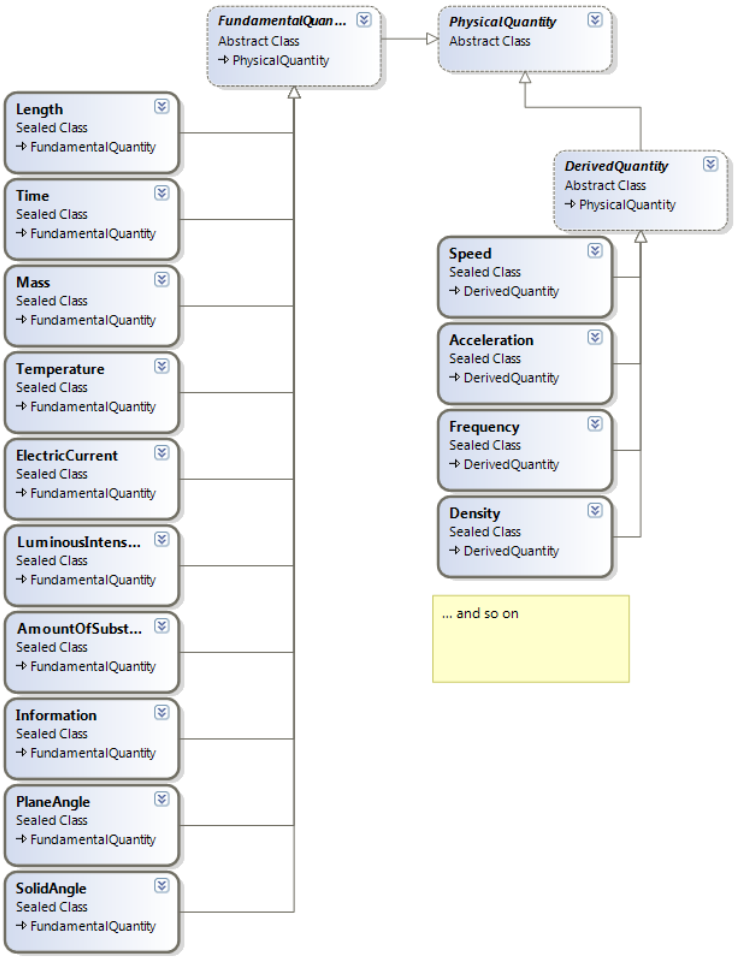
There are fundamental (base) physical quantities called dimensions: **length**, **time**, **mass**, **temperature**, **electric current**, **amount of substance**, **luminous intensity**. They are fundamental because the sense of all other quantities can be expressed via them. For an example, physical 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/time²**, area has dimension **length²** and so on.

Thus, physical quantity can be characterized by two properties: name (or symbol) and dimension. There are fundamental quantities with the same dimensions and derived quantities (expressed through the fundamental ones).

NOTE. Three additional dimensions can be introduced for convenience: **information**, **plane angle**, **solid angle**. These quantities are dimensionless, but can be considered as additional dimensions to differentiate them from truly dimensionless data (numbers).

Physical quantity class hierarchy

In accordance with said above, the class hierarchy of physical quantities in PHYSICS library follows the concepts of modern physics. Each physical quantity is represented by a class. Base abstract class for all physical quantities is **PhysicalQuantity**. It has three main properties: Name, Symbol and Dimension. **FundamentalQuantity** and **DerivedQuantity** are directly inherited from this base class. Ten fundamental quantities (**Length**, **Time**, **Mass**, **Temperature**, **ElectricCurrent**, **AmountOfSubstance**, **LuminousIntensity**, **Information**, **PlaneAngle**, **SolidAngle**) inherited from the former class and all other inherited from the last one. Class hierarchy diagram is shown on picture 1.1.



Picture 1.1. Physical quantity class hierarchy diagram.

Introducing new physical quantity

It is very easy to introduce new physical quantity. Because every quantity is represented by a class – new class must be inherited from **DerivedQuantity**. For an example, to introduce force physical quantity the following class must be created

```
/// <summary>
/// Force
/// </summary>
public sealed class Force : DerivedQuantity
{
    protected internal override string GetName()
    {
        return "Force";
    }

    protected internal override string GetSymbol()
    {
        return "F";
    }

    protected internal override Dimension GetDimension()
    {
        return (Dimension.Mass * Dimension.Length) / (Dimension.Time ^ 2);
    }
}
```

This code introduce new physical quantity with name 'Force' (symbol 'F') and dimension **mass·length/time²** (as force has)[1].

And it is all. PHYSICS library uses .NET reflection mechanisms to find and identify physics entities. New class will be automatically found and registered. All algorithms of working with physical quantities do not depend on the type of the quantity. The program can find quantity, display it (see later about localization), find units of measurement those are compatible with the quantity (and even automatically create compatible units) and so on.

Units (of measurement)

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 by one unit – Pascal.

In accordance with said above, there are base units to measure fundamental quantities. For length quantity – meters, inches and so on, for mass – grams, pounds and so on… All units for derived quantities can be produced by multiplying and dividing base units, multiplying them by some constants and adding constant values to them (using general algebra rules). For an example, the unit of area measurement 'Are' can be got multiplying **10 m** by **10 m**, so **1 a = 100 m²**.

One of the main purposes to use units of measurement in a program is to convert values of physical quantities from one unit to other ones. Base units are converted in accordance with their definitions. 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)**. All base units are 'linear' because they have 'one dimension'. So they all converted from one to another by linear law. The law of derived unit conversion is determined by the way of unit derivation. For an example **1 m² = 1 m · 1 m = 100 cm · 100 cm = 10000 cm²**.

Even the algorithm of unit conversion is clear for humans, it is not easy for a programmer to write the code. Main difficulty is to create 'right' class hierarchy that provides one algorithm for all units and for that time easy way to introduce new units.

PHYSICS library has **unique universal algorithm** of unit conversion. It converts value **in any unit to any other unit** (if they have equal dimensions). The only thing needed is to create new unit class. Algorithm will be able to convert values in the unit to any compatible one and vice versa.

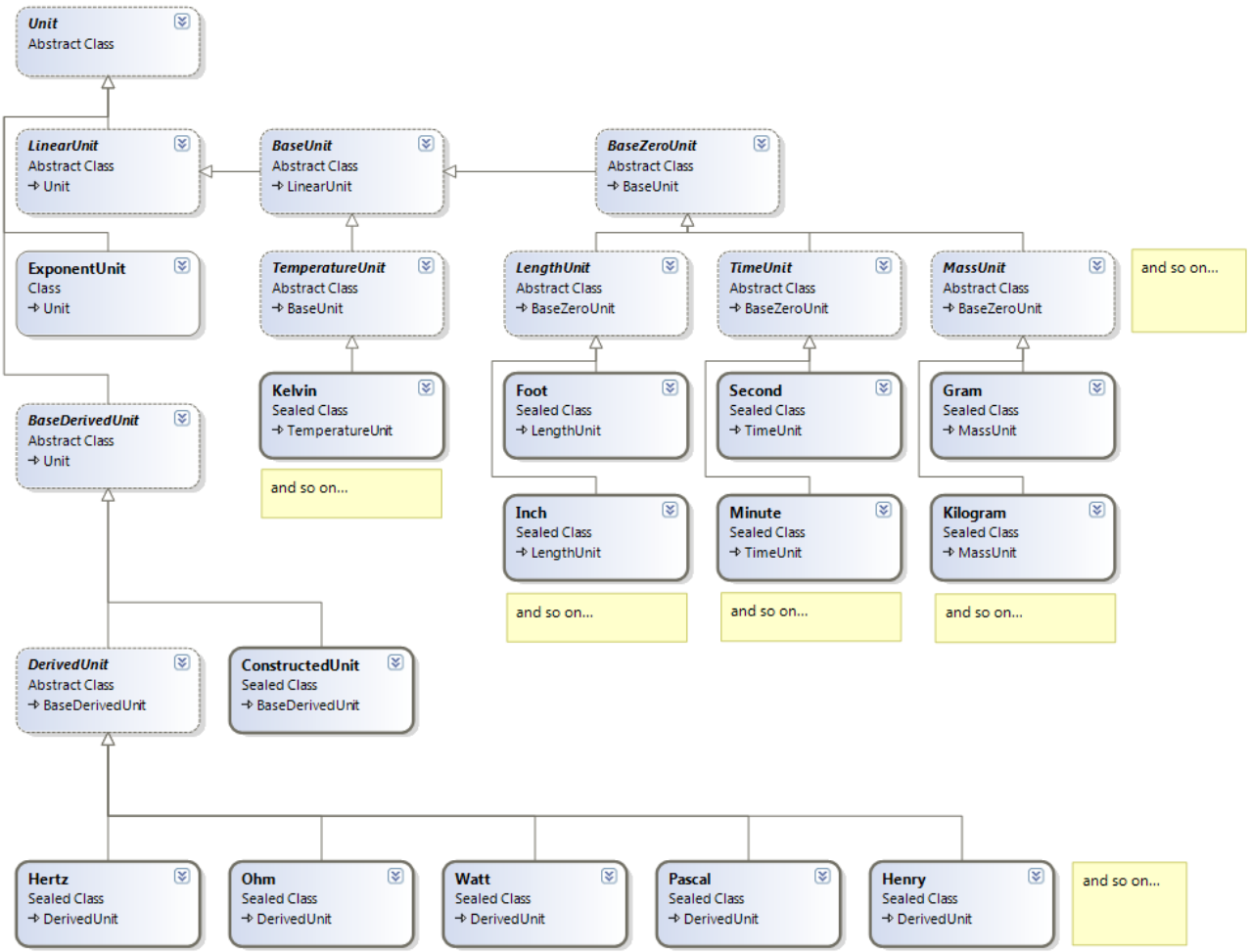
Unit class hierarchy

The unit class hierarchy (not total) is shown on picture 1.2. The hierarchy is rather complicated. The hierarchy was constructed with the purpose to represent the modern physics concept of 'units of measurement' as close as possible and to provide universal algorithm of unit conversion and the easiest way to introduce new units.

The base abstract class for all units is **Unit**. It provides all common unit properties – Name, Symbol, Dimension, and common algorithms – conversion values from one unit to another, checking units for compatibility and so on. The following operator overloading also provided: ==, !=, *, /, ^ (integer power operator), implicit unit to string conversion.

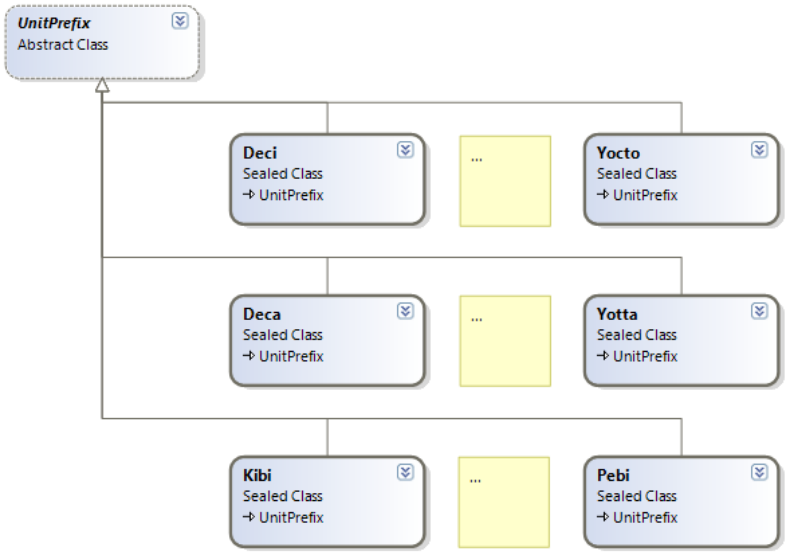
---

[1] Be careful of using overloaded power operator '^' in dimension construction code. Because C# operator '^' has lower precedence than '*' and '/' operators have, it must always be enclosed in parentheses to get 'mathematical' operator precedence.



Picture 1.2. Unit class hierarchy diagram.

Class **LinearUnit** directly derived from base abstract class **Unit** realizes abstract methods for units using linear conversion. As was said above, all units for fundamental physical quantities (dimensions) are linear, so **BaseUnit** class realizes abstract mechanisms for such units. Base units separated into eleven branches corresponding fundamental quantities (ten dimensions + dimensionless unit). All units corresponding fundamental quantities must be inherited from those eleven units. There are many predefined base units: **Foot**, **Gram**, **Second** and so on.



Picture 1.3. Unit prefix class hierarchy diagram.

All other units are constructed from base units. The simplest way to get new unit is to multiply it by some value and/or exponentiate it. For an example, from unit meter **m** new unit can be constructed **10 m²**. The **ExponentUnit** class realizes this concept. The special abstract class **UnitPrefix** (picture 1.3) is designed to realize standard multipliers for units

(http://en.wikipedia.org/wiki/SI_prefix). All standard multipliers realized in PHYSICS library and can be used for unit derivation.

And finally, general way to derive new unit from base units is to multiply several exponent units. For an example **kg·m²·A⁻¹·s⁻²**. This concept is realized by **BaseDerivedUnit** class. All derived units can be divided in two categories. The first – units have their own names. For an example – Newton, Pascal, Henry and so on. The second – units those have no names, they have only symbols. For an example **km·s⁻²**. These concepts realized by **DerivedUnit** and **ConstructedUnit** classes correspondingly.

Many units already defined in PHYSICS library. New units can be simply added to the library without need to modify algorithms of unit using.

Introducing new units

The unit class hierarchy is rather complicated, but there is no need to know all features of the structure to introduce new units in a program and use them. PHYSICS library provides simple mechanisms to introduce and use new units.

There are two different ways to use new units in a program. The first is to create new classes for units. This way must be used for all units, those have their own names (Pascal, Newton and so on). For units corresponding base physical quantities the class must be inherited from one of the base derived units. As an example there is code for unit 'yard'.

```
/// <summary>
/// Yard = 0.9144 m
/// </summary>
public sealed class Yard : LengthUnit
{
    protected internal override double GetFactor()
    {
        return 0.9144;
    }

    protected internal override string GetName()
    {
        return "Yard";
    }

    protected internal override string GetPlural()
    {
        return "Yards";
    }

    protected internal override string GetSymbol()
    {
        return "yd";
    }

}
```

The **Yard** unit is inherited from the **LengthUnit**, because it is a measure for **length** physical dimension. It overrides methods providing the name and the symbol of the unit. Also it overrides the GetFactor method, that returns the coefficient of conversion to base unit. It is the only information needed for conversion algorithm. All coefficients must correspond to conversion of the unit to base SI units (http://en.wikipedia.org/wiki/SI).

All complex units must be directly inherited from **DerivedUnit** class. As an example there is code for unit 'watt'.

```
/// <summary>
/// Watt ( kg m^2/s^3 )
/// </summary>
public sealed class Watt : DerivedUnit
{
    protected internal override string GetName()
    {
        return "Watt";
    }

    protected internal override string GetPlural()
    {
        return "Watts";
    }

    protected internal override string GetSymbol()
    {
        return "W";
    }

    protected internal override void RecreateUnits()
    {
        fUnits = new ExponentUnit[3];
        fUnits[0] = new ExponentUnit(new Kilogram(), 1);
```



```
            fUnits[1] = new ExponentUnit(new Metre(), 2);
            fUnits[2] = new ExponentUnit(new Second(), -3);
        }
    }
```

The class overrides methods providing the name and symbol, and also it overrides the **RecreateUnits** method which provides main information for conversion and other algorithms. In this method the unit 'watt' is constructed with three exponent units **Kilogram**^{1} **·Metre**^{2} **·Second**$^{-3}$**.**

The second way to use new units in a program is creating object instances dynamically in run-time. The simplest way is using overloaded operators. The following code demonstrates run-time unit creation.

```
Unit u1 = new Kilogram();
Unit u2 = new Metre();
Unit u3 = new Second();
Unit x = u1 * u2 * (u3 ^ -2);
```

The created unit x is **kg·m/s**$^{2.2}$

Unit conversion

The unit conversion is very simple to implement with PHYSICS library. The base class Unit has static method **Convert** to convert value from any unit to any compatible one. The following code demonstrates an example of conversion

```
Unit u1 = new Pascal();
Unit u2 = new Atmosphere();
double x1 = 100;
double x2 = Unit.Convert(u1, u2, x1);
```

In this example double value converted from Pascal to Atmosphere unit. The units must be compatible to convert values. Units are compatible if they have equal dimensions. Use Unit class static method **Convertible** to check that two units are compatible for conversion.

Fast unit conversion

The unit conversion, described above, is universal and can be applied to any (convertible) units. But it can be slow for conversion of huge array of values. The more 'complicated' units are the slower conversion is.

At the same time, all units can be divided into two categories[3]: 'zero based' and not 'zero based'. 'Zero based' units are such units that their scale zeros equal. For an example, mass units are zero based, because zero mass is always zero, no matter what unit is used to measure it. As opposite, temperature units are not zero based, because **0°C is not equal to 0°K**.

The conversion of 'zero based' units if always linear and can be made using one factor coefficient **f**. Then the formula is $\mathbf{x\ unit_1 = f \cdot x\ unit_2}$, where **x** is the value, $\mathbf{unit_1}$ and $\mathbf{unit_2}$ are the units to convert from and to convert to, **f** is the conversion factor. The **Unit** class contains method **IsZeroBased** to check, if the unit is zero based.

The conversion of not 'zero based' units is more complicated and can be not linear. But, when 'interval' conversion must be made, all unit conversions are linear and can be made by multiplying the value with 'scale factor'. For an example, the temperature interval of **10°C** can be converted to temperature interval measured in Fahrenheit's degrees by multiplying it with **9/5**.

In accordance with said above, for 'zero based' units and for 'interval' conversion the fast scale factor can be used to convert values in one unit to another. The **Unit** class contains static method **ScaleFactor** to calculate this value. The example code below shows how to use this method to convert the array of values.

Let there is an array of pressure values measured in 'millimeters of mercury'.

```
double[] pressures;
```

And all values must be converted to Pascal units. The code of conversion is the following:

```
Unit x1 = new MillimetreOfMercury();
Unit x2 = new Pascal();
double z = Unit.ScaleFactor(x1, x2);
for (int i = 0; i < pressures.Length; i++)
{
    pressures[i] = z * pressures[i];
}
```

String to Unit conversion

The PHYSICS library provides mechanisms to convert string data to units of measurement. It is useful when a program User wants to input units to measure some value. The mechanism of string-to-unit conversion is provided by the **UnitConverter** class. The following code demonstrates conversion of a string to a unit.

```
UnitConverter uconverter = new UnitConverter();
string ustr = "N mm^2/ns";
string error;
Unit u = uconverter.ConvertString(ustr, out error);
```

In this example new unit x is constructed from string and it is **Newton·millimeter**^{2}**/nanosecond**.

The rules of string-to-unit conversion:

---

[2] Be careful of using overloaded power operator '^' in unit construction code. Because C# operator '^' has lower precedence than '*' and '/' operators have, it must always be enclosed in parentheses to get 'mathematical' operator precedence.

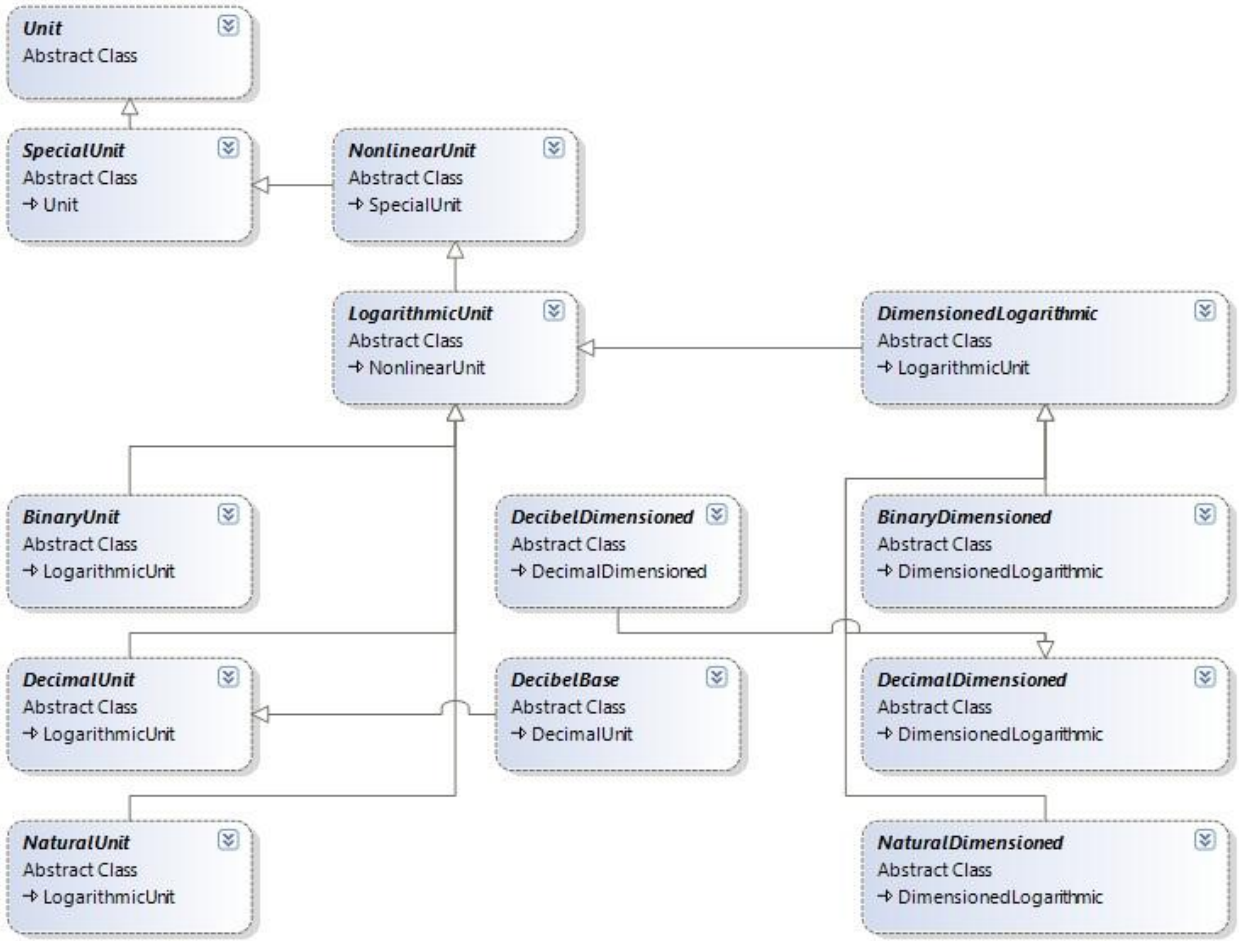
[3] For other cases see 'Special units' part.

- The string can contain unit symbols, prefixes, spaces, division operator sign and power operator sign.
- Only one division sign may be in the string.
- Prefixes are not separated by spaces from units they belong to.
- Exponent units are separated by spaces (that is spaces used as multiplication operator).

If there is an error in the string, its description is returned via error out parameter.

Special units (logarithmic and decibels)

The unit class hierarchy presented above is not full. It was reduced for the reason of simplification. Really, there is another branch of inheritance shown on picture 1.4.



Picture 1.4. Diagram of special unit class branch of inheritance.

The main branch node is **SpecialUnit**. This branch was introduced for special needs. The **SpecialUnit** class used in all core algorithms of the library, but implementation supposes special cases. Common example of special unit is Decibel – this is a logarithmic unit with special rules of conversion and operations (https://en.wikipedia.org/wiki/Decibel). Decibel is defined as logarithmic unit by base 10, so it inherited from **Decimal** logarithmic unit. Really, all logarithmic units are dimensionless, because they express the relative values. But dimensioned decibels can be introduced by supposing base dimensioned value (https://en.wikipedia.org/wiki/Decibel#Suffixes and reference values).

Common feature of logarithmic (and decibel) units is that they are NOT SCALABLE. It means that 'interval' conversion for them cannot be done by multiplying with a factor value (in general case). The conversion between logarithmic units and other ones can be done only with general conversion algorithm (no fast conversion). Use method **IsScalable** of Unit class for checking if unit can be converted to another one by scale factor. Also, method **MakeConversion** can be used for creating suitable conversion delegate for ANY units, taking into account their properties.

Summary of logarithmic and decibel unit features:

- Logarithmic units are not scalable with other ones.
- There are dimensionless and dimensioned decibels.
- Logarithmic units and decibels cannot be multiplied with other logarithmic ones.
- Logarithmic units and decibels cannot be raise to a power (except 1).
- When making operations with physical values (see below) logarithmic units make special sense: the logarithmic values cannot be multiplied; the logarithmic values cannot be raise to a power; sum of two logarithmic values can be done if they have the same dimension or if one of them is dimensionless.

Physical (quantity) values[4]

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 (any) two values **2 m · 3 s^{-2} = 6 m/s^2**.

The base abstract class for physical values is **PhysicalValue**. It provides base data (unit field) and functionality (conversion from string to value and vice versa). Inherited class **ScalarValue** overrides and adds functionality to work with scalar physical values. The following code demonstrates using of scalar values

```
string s1 = "9.8 m/s^2";
string s2 = "70.5 kg";
ScalarValue v1 = ScalarValue.Parse(s1);
ScalarValue v2 = ScalarValue.Parse(s2);
ScalarValue f = v1 * v2;
string s = f.ToString();
```

Two scalar values v1 and v2 created from string values. The former value represents acceleration and the last represents mass. When the values multiplied new value f created which represents force. As a result string s will have value "690.9 m kg/s^2".

Analogously other physical values can be defined – vector, tensor, matrix and so on. There are complete realization of **VectorValue** and **TensorValue** classes (for 3D vectors and Tensors respectively) with overloaded operators and **Parse** function implementation. Code examples of using vector and tensor physical values:

```
VectorValue s1 = VectorValue.Parse("(0 1 0) m");
VectorValue s2 = VectorValue.Parse("(203e-1 0 0) ft");
VectorValue F = VectorValue.Parse("(2 1 -1) N");
ScalarValue W = VectorValue.Dot(F, s1+s2);
…
TensorValue x1 = TensorValue.Parse("(1 2 -1 0 1 -2 1 1 0) mm g");
TensorValue x2 = TensorValue.Parse("(0 -1 1 1 1 -2 1 1 0) s");
TensorValue s = x1/x2;
```

And the result values are the following:

```
W = 13.37488 N m
s = (-1 0 1 -1 0.5 -0.5 0 0 1) mm g/s
```

Rules of physical value operations:

- Two physical values can be added, if their values can be added by mathematical rules and their units are compatible (have the same dimensions).
- When physical values summarized, the second value converted to the first value unit.
- If conversion done, it is implemented as 'interval' conversion (when we add one meter and two centimeters we add them as intervals) if it is possible.
- When adding logarithmic values they are converted by the general algorithm.
- When multiplying values with the same dimensions, the result is expressed in terms of squared first unit (except dimensionless – not squared).
- When dividing values with the same dimensions, the result is expressed in terms of dimensionless (parts of 1) (except dimensionless – first unit kept).

Localization

PHYSICS library provides simple mechanism to localize all displayed values. There are two classes **LocalizedResourceUnitConverter** and **LocalizedResourceQuantitiesManager** to localize units and quantities respectively. Global static value of **PhysicalValue.UnitConverter** must be assigned to a **LocalizedResourceUnitConverter** instance to localize physical value conversion.

The localization classes use standard .NET resource mechanism to get localization values for various cultures. The names of resources are identified by class and property names. For an example, to localize the Symbol of Pascal unit there must be resource with name "Pascal_Symbol" for localization culture. This resource must be provided for the **LocalizedResourceUnitConverter** with the resource manager in constructor.

There is default localization for Russian culture in the PHYSICS library that can be used as an example. Note, that if no resource is provided for some symbol, default value (as defined in code) can be used. So, maximum two values can be used for any symbol in the same time – default and localized.

[4] Physical value classes defined in Mathphysics assembly.

