The goal of this library is to model units of measure and manipulation of quantities associated to units of measure.

# Day 1: fundamental units

To understand this one need to know bit about the metric system and dimensional equation. Please read <https://en.wikipedia.org/wiki/Metric_system> and <https://en.wikipedia.org/wiki/Dimensional_analysis>.

We use the 7 base units of the SI (<https://en.wikipedia.org/wiki/International_System_of_Units>) as a starting point: these are the FundamentalMeasureUnit. A fundamental unit is semantically equivalent (for us) to a dimension. We add 3 fundamental units (dimension) and new ones can be dynamically added to the system as required (for instance “$” and/or “€”, “£”, etc.). Note that each currency would be a dimension. Converting a quantity of “$” to “€” is not in the scope of this library.

The 3 fundamental units we add are:

/// <summary>

/// Dimensionles unit. Associated abbreviation is "" (the empty string) and its

/// name is "None".

/// </summary>

public static readonly FundamentalMeasureUnit None;

/// <summary>

/// Dimensionless unit. Used to count items. Associated abbreviation is "#".

/// </summary>

public static readonly FundamentalMeasureUnit Unit;

/// <summary>

/// A bit is defined as the information entropy of a binary random variable

/// that is 0 or 1 with equal probability.

/// Associated abbreviation is "b" (recommended by the IEEE 1541-2002 and

/// IEEE Std 260.1-2004 standards).

/// </summary>

MeasureUnit

NormalizedMeasureUnit

BasicMeasureUnit

FundamentalMeasureUnit

public static readonly FundamentalMeasureUnit Bit;

These fundamental units actually are BasicMeasureUnit with their Exponent set to 1. The BasicMeasureUnit are bound to a FundamentalMeasureUnit with an exponent: they handle the basic items of a dimensional equation m^2, s^-1, etc.

NormalizedMeasureUnit captures a list of one or more BasicMeasureUnit. The list is normalized by decreasing exponents and then by fundamental unit’s name lexicographic order.

Figure 1- Class hierarchy so far.

With this simple model and the help of C# operators overload (\*, / and ^), one can achieve this kind of funny dimensional game (from <https://en.wikipedia.org/wiki/SI_derived_unit>):

var metre = MeasureUnit.Metre;

var second = MeasureUnit.Second;

var kilogram = MeasureUnit.Kilogram;

var ampere = MeasureUnit.Ampere;

var candela = MeasureUnit.Candela;

var squaredMeter = metre^2;

var hertz = MeasureUnit.None / second;

hertz.Abbreviation.Should().Be( "s-1" );

var rad = metre \* (metre ^ -1);

rad.Should().BeSameAs( MeasureUnit.None );

var steradian = squaredMeter / squaredMeter;

steradian.Should().BeSameAs( MeasureUnit.None );

var newton = kilogram \* metre \* (second ^ -2);

newton.Abbreviation.Should().Be( "kg.m.s-2" );

var pascal = newton / squaredMeter;

pascal.Abbreviation.Should().Be( "kg.m-1.s-2" );

var joule = newton \* metre;

var watt = joule / second;

var coulomb = ampere \* second;

var volt = watt / ampere;

var volt2 = joule / coulomb;

volt.Should().BeSameAs( volt2 );

volt.Abbreviation.Should().Be( "m2.kg.A-1.s-3" );

var farad = coulomb / volt;

farad.Abbreviation.Should().Be( "s4.A2.kg-1.m-2" );

var ohm = volt / ampere;

ohm.Abbreviation.Should().Be( "m2.kg.A-2.s-3" );

var farad2 = second / ohm;

farad2.Should().BeSameAs( farad );

var siemens = MeasureUnit.None / ohm;

var siemens2 = ampere / volt;

siemens.Should().BeSameAs( siemens2 );

siemens.Abbreviation.Should().Be( "s3.A2.kg-1.m-2" );

var weber = joule / ampere;

var tesla = volt \* second / squaredMeter;

var tesla2 = weber / squaredMeter;

var tesla3 = newton / (ampere \* metre);

tesla2.Should().BeSameAs( tesla );

tesla3.Should().BeSameAs( tesla );

tesla.Abbreviation.Should().Be( "kg.A-1.s-2" );

var henry = ohm \* second;

var henry2 = volt \* second / ampere;

var henry3 = weber / ampere;

henry2.Should().BeSameAs( henry );

henry3.Should().BeSameAs( henry );

henry.Abbreviation.Should().Be( "m2.kg.A-2.s-2" );

var lumen = candela \* steradian;

var lux = lumen / squaredMeter;

lux.Abbreviation.Should().Be( "cd.m-2" );

Important note: the code above shows that the objects are actually the same object (reference equality) when they define the same unit of measure. This is one of the goal of this library: units must be fully normalized and cached even if new unit of measure can be dynamically created at any time. This is achieved thanks to a ConcurrentDictionary, the “Abbreviation” property of the measures being the key.

Even if it works like a charm and is funny, this is not enough. We must handle:

* Prefixes: standard ones like “d”/”Deci” like in “dm”/”Decimeter” or “m”/”Milli” like in “mm”/”Millimeter”. See <https://en.wikipedia.org/wiki/Metric_prefix> and since we added bit, we should also handle binary prefixes (see <https://en.wikipedia.org/wiki/Binary_prefix>): “Ki”/”Kibbi” like “Kib”/”KibiBit”.

Introducing this is not as easy as it seems. We must be able to transparently handle equivalence like, for instance, the fact that m.s-2 (metre per squared second, ie. an acceleration) is the same as μm.ms-2 (micrometer per squared milliseconds).

* Derived units: a Derived Unit is like an alias, for instance “l”/”Liter” is the same as “dm3”, ie. 10-3 m3. Some of the actually used units introduce a factor (that does not fit into a simple exponentiation like the liter).

For instance (from <https://en.wikipedia.org/wiki/Newton_(unit)>):

* + A newton (N): its definition directly uses fundamental units: 1 N = 1 kg.m.s-2
  + A dyne (dyn) is defined with the newton: 1 dyn = 10-5 N
  + A kilopond (kp) is: 1 kp = 9.80665 N

This definitely requires a more complex model.

# Day 2: Handling prefixes.

Let’s start with standard and binary prefixes. These prefixes (“G”/”Giga”, “k”/”Kilo”, “K”/”Kibi”, “m”/”Milli” etc.) apply only to non-exponentiated units like the fundamental units and can be used wherever a fundamental unit is used.

A deci(squared metre) is NOT Valid, however a squared( decimetre ) is. To properly model this, we refine a little bit our model by introducing a PrefixedMeasureUnit at the same (lowest) level as the FundamentalMeasureUnit.

To unify these two types we introduce a new AtomicMeasureUnit class that is the base class of FundamentalMeasureUnit and PrefixedMeasureUnit.

The NormalizedMeasureUnit now handles one or more (possibly exponentiated) fundamental units as well as PrefixedMeasureUnit that are not “fundamentals”. Its name (“normalized”) is no more accurate: we change it to just be a CombinedMeasureUnit that better reflects what it is.

Since we are dealing with (bad) names, the “Basic measure unit” does not convey a clear idea of what it is. We rename the BasicMeasureUnit to be ExponentMeasureUnit.

The new class diagram is:

MeasureUnit

CombinedMeasureUnit

AtomicMeasureUnit

FundamentalMeasureUnit

PrefixedMeasureUnit

ExponentMeasureUnit

Figure - Introducing Prefixes.

The standard S.I. prefixes (all of them) are captured and exposed by the MeasureStandardPrefix class through a set of singletons. This set is not extensible (you can’t define your own prefix).

These prefixes can be applied to any AtomicMeasureUnit to obtain another AtomicMeasureUnit since the result of the prefix application may actually be a prefixed unit or a fundamental unit.

var centimetre = MeasureStandardPrefix.Centi.On( MeasureUnit.Metre );  
centimetre.Abbreviation.Should().Be( "cm" );  
centimetre.Name.Should().Be( "Centimetre" );

If we apply the Hecto prefix to this centimetre, we obtain the (fundamental) metre.

var hectocentimetre = MeasureStandardPrefix.Hecto.On( centimetre );

hectocentimetre.Should().BeSameAs( MeasureUnit.Metre );

Below is the whole test for this new game:

[Test]

public void playing\_with\_decimetre\_and\_centimeter()

{

var decimetre = MeasureStandardPrefix.Deci.On( MeasureUnit.Metre );

decimetre.Abbreviation.Should().Be( "dm" );

decimetre.Name.Should().Be( "Decimetre" );

var decimetreCube = decimetre ^ 3;

decimetreCube.Abbreviation.Should().Be( "dm3" );

decimetreCube.Name.Should().Be( "Decimetre^3" );

// This does'nt compile and this is perfect! :)

//var notPossible = MeasureStandardPrefix.Deci.On( decimetreCube );

var centimetre = MeasureStandardPrefix.Centi.On( MeasureUnit.Metre );

centimetre.Abbreviation.Should().Be( "cm" );

centimetre.Name.Should().Be( "Centimetre" );

var decidecimetre = MeasureStandardPrefix.Deci.On( decimetre );

decidecimetre.Should().BeSameAs( centimetre );

var hectocentimetre = MeasureStandardPrefix.Hecto.On( centimetre );

hectocentimetre.Should().BeSameAs( MeasureUnit.Metre );

var kilocentimeter = MeasureStandardPrefix.Kilo.On( centimetre );

kilocentimeter.Abbreviation.Should().Be( "dam" );

var decametre = MeasureStandardPrefix.Deca.On( MeasureUnit.Metre );

decametre.Should().BeSameAs( decametre );

}

To conclude this second day, one need to consider a not-so-edge-case: standard prefixes are not “complete” enough to be safely used. What happens if you want the “Deci” (10-1) of a “Giga” (109)? There is no prefix available for 108. Similar question: what is a Kilo Yotta (Yotta is the biggest prefix: 1024).

Note that you may not ask it directly, but this may happen indirectly in a complex system. We must be able to cope with these cases, even if we’ll always try to ***eventually*** avoid such “intermediate” prefixes.

The idea is to introduce an “adjustment factor” to our PrefixedMeasureUnit. This adjustment is a ExpFactor that enables us to handle exponents without relying on floating point types (and their intrinsic limitations).

/// <summary>

/// Immutable value type that captures 10^<see cref="Exp10"/>.2^<see cref="Exp2"/>.

/// </summary>

public struct ExpFactor : IComparable<ExpFactor>, IEquatable<ExpFactor>

{

public static readonly ExpFactor Neutral; // The neutral factor (0,0).

public readonly int Exp2; // The base 2 exponent.

public readonly int Exp10; // The base 10 exponent.

public ExpFactor Power( int p )

=> new ExpFactor( Exp2 \* p, Exp10 \* p );

public ExpFactor Multiply( ExpFactor x )

=> new ExpFactor( Exp2 + x.Exp2, Exp10 + x.Exp10 );

public ExpFactor DivideBy( ExpFactor x )

=> new ExpFactor( Exp2 - x.Exp2, Exp10 - x.Exp10 );

}

Note: There is a little bit more code in the struct above, we only show here the most relevant aspects.

There are two exponents: one in base 2 (since we support binary prefixes) and the other in base 10 for matric prefixes.

This allows us to generate “intermediate prefixes” or “out-of-bound” prefixes as required and keep the whole system safe and coherent as long as we find a way to express/show/recognize these beasts.

If an adjustment factor is not the neutral one, it appears in the abbreviation and name of the unit of measure: “(10^-1)Gm”/”(10^-1)Gigameter” will be the “Decigigametre” that do not exist. Of course, you will never meet the “(10^-1)cm” since this is a “mm”/”Millimetre”!

Note the syntax: we have deliberately chosen a syntax that differs from the standard exponent (with the base and the caret ^) so that these beasts can easily been spotted.

Here are the tests that conclude this Day 2:

[Test]

public void playing\_with\_adjustment\_factors()

{

var gigametre = MeasureStandardPrefix.Giga[MeasureUnit.Metre];

gigametre.Abbreviation.Should().Be( "Gm" );

gigametre.Name.Should().Be( "Gigametre" );

var decigigametre = MeasureStandardPrefix.Deci[gigametre];

decigigametre.Abbreviation.Should().Be( "(10^-1)Gm" );

decigigametre.Name.Should().Be( "(10^-1)Gigametre" );

// Instead of "(10^-2)Gigametre", we always try to minimize the absolute value   
// of the adjustement factor: here we generate the "(10^1)Megametre".

var decidecigigametre = MeasureStandardPrefix.Deci[decigigametre];

decidecigigametre.Abbreviation.Should().Be( "(10^1)Mm" );

decidecigigametre.Name.Should().Be( "(10^1)Megametre" );

var decidecidecigigametre = MeasureStandardPrefix.Deci[decidecigigametre];

decidecidecigigametre.Abbreviation.Should().Be( "Mm" );

decidecidecigigametre.Name.Should().Be( "Megametre" );

}

[Test]

public void out\_of\_bounds\_adjustment\_factors()

{

var yottametre = MeasureStandardPrefix.Yotta[MeasureUnit.Metre];

var lotOfMetre = MeasureStandardPrefix.Hecto[yottametre];

lotOfMetre.Abbreviation.Should().Be( "(10^2)Ym" );

var evenMore = MeasureStandardPrefix.Deca[lotOfMetre];

evenMore.Abbreviation.Should().Be( "(10^3)Ym" );

var backToReality = MeasureStandardPrefix.Yocto[evenMore];

backToReality.Abbreviation.Should().Be( "km" );

var belowTheAtom = MeasureStandardPrefix.Yocto[backToReality];

belowTheAtom.Abbreviation.Should().Be( "zm" );

belowTheAtom.Name.Should().Be( "Zeptometre", "The Zeptometre is 10^-21 metre." );

var decizettametre = MeasureStandardPrefix.Deci[belowTheAtom];

decizettametre.Abbreviation.Should().Be( "(10^-1)zm" );

var decidecizettametre = MeasureStandardPrefix.Deci[decizettametre];

decidecizettametre.Abbreviation.Should().Be( "(10^1)ym" );

var yoctometre = MeasureStandardPrefix.Deci[decidecizettametre];

yoctometre.Abbreviation.Should().Be( "ym" );

var below1 = MeasureStandardPrefix.Deci[yoctometre];

below1.Abbreviation.Should().Be( "(10^-1)ym" );

var below2 = MeasureStandardPrefix.Deci[below1];

below2.Abbreviation.Should().Be( "(10^-2)ym" );

}

# Day 3: Handling aliases.

What we have so far is interesting however trivial issues are not yet covered. The fact that previous code shown use metre and not kilogram is by no mean an accident… Currently, there is no way to manipulate grams since for the moment a gram is a Milli Kilogram: Kilogram is the official standard unit of weight. You don’t want so see the “mkg” unit!

Of course, we can “cheat” and define the Gram as being the fundamental unit instead of the Kilogram (and everything works fine). But there should be a better way… like supporting aliases and using it to stipulate that “mkg” (Millikilogram) is actually “g”. This substitution must be done everywhere a “mkg” could “popup”, and specifically inside the MeasureStandardPrefix class.