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 2- 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

It is time now to handle aliases and eventually find a way to normalize units so that we can actually use them to compute quantities.

## The kilogram exception and the byte

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!

Introducing an alias for the “mkg” (as a “g”/”Gram”) would imply to cope with this exception in too much code location. It is definitely easier to “cheat” and define the Gram as being the fundamental unit instead of the Kilogram. The Kilogram is now defined in the MeasureUnit type initializer as a PrefixedMeasureUnit (and still exposed as a static property):

/// <summary>

/// The kilogram is the unit of mass; it is equal to the mass of the international

/// prototype of the kilogram.

/// This is the only SI base unit that incluedes a prefix. To avoid coping with this

/// exception in the code, we

/// define it as a PrefixedMeasureUnit based on the gram (MeasureStandardPrefix.Kilo

/// of Gram).

/// Associated abbreviation is "kg".

/// </summary>

public static readonly PrefixedMeasureUnit Kilogram;

/// <summary>

/// The gram is our fundamental unit of mass (see Kilogram).

/// Associated abbreviation is "g".

/// </summary>

public static readonly FundamentalMeasureUnit Gram;

Kilogram = RegisterPrefixed( ExpFactor.Neutral, MeasureStandardPrefix.Kilo, Gram );

Byte is also exposed as a static field on the MeasureUnit class. It uses the new AliasMeasureUnit class that now enables us to create new named units from other ones:

/// <summary>

/// A byte is now standardized as eight bits, as documented in ISO/IEC 2382-1:1993.

/// The international standard IEC 80000-13 codified this common meaning.

/// Associated abbreviation is "B" and it is an alias with a ExpFactor 2^3 on Bit.

/// </summary>

public static readonly AliasMeasureUnit Byte;

Byte = new AliasMeasureUnit( "B", "Byte", new FullFactor( new ExpFactor(3,0) ), Bit );

## Defining new units

The new alias and the original fundamental units are the only two ways to explicitly define new unit of measure:

/// <summary>

/// Defines an alias.

/// The same alias can be registered multiple times but it has to exactly match the

/// previously registered one.

/// </summary>

/// <param name="abbreviation">

/// The unit of measure abbreviation.

/// This is the key that is used. It must not be null or empty.

/// </param>

/// <param name="name">The full name. Must not be null or empty.</param>

/// <param name="definitionFactor">

/// The factor that applies to the AliasMeasureUnit.Definition. Must not be

/// FullFactor.Zero.

/// </param>

/// <param name="definition">The definition. Can be any CombinedMeasureUnit.</param>

/// <returns>The alias unit of measure.</returns>

public static AliasMeasureUnit DefineAlias(

string abbreviation,

string name,

FullFactor definitionFactor,

CombinedMeasureUnit definition ) { … }

/// <summary>

/// Define a new fundamental unit of measure (or returns the already defined one).

/// Just like DefineAlias, the same fundamental unit can be redefined multiple times

/// as long as it is actually the same: for fundamental units, the (long) name

/// must be exaclty the same.

/// </summary>

/// <param name="abbreviation">

/// The unit of measure abbreviation.

/// This is the key that is used. It must not be null or empty.

/// </param>

/// <param name="name">The full name. Must not be null or empty.</param>

/// <returns>The fundamental unit of measure.</returns>

public static FundamentalMeasureUnit DefineFundamental(

string abbreviation,

string name ) { … }

The FullFactor that appears in the DefineAlias is a simple extension to the ExpFactor previously described that adds a simple Factor field to it. We use a double for the moment but what we should use here is a rational such as the one provided by <https://www.nuget.org/packages/Rationals/> (that seems a nice project).

A FullFactor describes a simple linear adjustment of the definition units. Thanks to it we can now define the newton, the dyne and the kilopound (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

using static CK.Core.MeasureUnit;

…

var kg = Kilogram;

var m = Metre;

var s = Second;

var newton = DefineAlias( "N", "Newton", FullFactor.Neutral, kg \* m \* (s ^ -2) );

var dyne = DefineAlias( "dyn", "Dyne", new ExpFactor( 0, -5 ), newton );

var kilopound = DefineAlias( "kp", "Kilopound", 9.80665, newton );

var newtonPerDyne = newton / dyne;

newtonPerDyne.Abbreviation.Should().Be( "N.dyn-1" );

Note:

* To shorten the code above, we have used the using static CK.Core.MeasureUnit;
* FullFactor defines implicit conversion operators from double and ExpFactor. We could have defined the newton in an even more simple way:

var newton = DefineAlias( "N", "Newton", 1.0, kg \* m \* (s ^ -2) );

The new class diagram is:

MeasureUnit

CombinedMeasureUnit

AtomicMeasureUnit

FundamentalMeasureUnit

PrefixedMeasureUnit

ExponentMeasureUnit

AliasMeasureUnit

Figure 3- Introducing Aliases.

So far so good… but… what is the actual unit of “N.dyn-1”?

You may be surprised but it could be “radian” (or “steradian“) since this has actually no unit: it is Measure.None (just like radian and steradian that are not actual units).

We are now ready to use these units to *calculate* quantities!

# Entering the unit normalization, starting calculation

A Quantity is simply a numerical value (int, float, double, rational, big integer, etc.) associated to its unit od measure.

One of the goal of this library is to help computing such quantities like:

The value of r is 2.105. That is 2.105 MeasureUnit.None for us: the N.dyn-1 must resolve to the dimensionless unit **and** to the factor that link them.

Before manipulating quantities, they must be first promoted/aligned to the same measure of units. You can always multiply/divide quantities (whatever their dimensions are) by simply creating the resulting dimension: 10 m x 40 min = 400 m.min

However, to add or subtract 2 quantities hey must have exactly the same unit: 2 m2 + 3 m2 = 5m2 or 2 m2 + 3 cm2 that is perfectly valid and equals to 20003 cm2 (or 2.0003 m2 as you prefer).

Before tackling this aspect, we need a little bit of refactoring. Current model has the MeasureUnit as a base abstract class of all units. However, the only actual specialization is the CombinedMeasureUnit and it appears that this is all what is needed (this was not obvious on day 1, this indirection seemed useful at this time). We can now simplify the model by acting that a MeasureUnit is always a CombinedMeasureUnit. The latter is gone, merged into the MeasureUnit (that is getting bigger) and to keep code clean it is split in partial files.

The final class diagram is:

MeasureUnit

AtomicMeasureUnit

FundamentalMeasureUnit

PrefixedMeasureUnit

ExponentMeasureUnit

AliasMeasureUnit

Figure 4- Final class diagram.

We could have merged Exponent and Atomic, but we did not and don’t want to. For 2 reasons:

* This (see Day 1) is currently not possible and we want to keep this:

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 );

* This would make the code less “type safe” and hence a little bit more “complex” to write and to understand.

Refactor done. Let’s go back to the task of the day: the normalization.

There is here an obvious choice: the FundamentalMeasureUnit is by design the canonical form of each and every mono-dimensional unit of measure. When multiple dimensions are involved, there is one composite MeasureUnit that contains only atomic FundamentalMeasureUnit (i.e. no more aliases or prefixed units).

From any MeasureUnit to its Canonical form, we can calculate (once for all) the FullFactor that maps the unit to it.

Introducing a new class is useless for this. It is easier to introduce this into our model by adding 2 properties to the MeasureUnit base class:

/// <summary>

/// Gets whether this MeasureUnits only contains normalized units.

/// </summary>

public bool IsNormalized { get; }

/// <summary>

/// Gets the factor that must be applied from this measure to its Normalization.

/// </summary>

public FullFactor NormalizationFactor { get; }

/// <summary>

/// Gets the canonical form of this measure.

/// Its IsNormalized property is necessarily true.

/// </summary>

public MeasureUnit Normalization { get; }

And it works!

[Test]

public void equivalent\_combined\_units()

{

var metre = MeasureUnit.Metre;

var second = MeasureUnit.Second;

var acceleration = metre / (second ^ 2);

acceleration.IsNormalized.Should().BeTrue();

var micrometre = MeasureStandardPrefix.Micro[metre];

var millisecond = MeasureStandardPrefix.Milli[second];

var acceleration2 = micrometre / (millisecond ^ 2);

acceleration2.IsNormalized.Should().BeFalse();

acceleration2.Normalization.Should().BeSameAs( acceleration );

acceleration2.NormalizationFactor.Should().Be( FullFactor.Neutral );

}

[Test]

public void combined\_units\_with\_factor()

{

var metre = MeasureUnit.Metre;

var second = MeasureUnit.Second;

var speed = metre / second;

speed.IsNormalized.Should().BeTrue();

var kilometre = MeasureStandardPrefix.Kilo[metre];

var hour = MeasureUnit.DefineAlias( "h", "Hour", 60\*60, second );

var speed2 = kilometre / hour;

speed2.IsNormalized.Should().BeFalse();

speed2.Normalization.Should().BeSameAs( speed );

speed2.NormalizationFactor.ToDouble()

.Should().BeApproximately( 0.2777777778, 1e-10, "1 m/s = 0.277778 km/h" );

}

We have no more time today (day 4) to detail the (minimalist) code that does this “magic” (what is actually magic is how minimal it is 😊). We’ll talk about this tomorrow.

# Day 5 - the quantities.

Yesterday while implementing the normalization I realized that it was quite easy to separate the two notions of Canonical/Normalized unit of measure and FundamentalMeasureUnit that I initially consider as being the same thing. Thanks to this, the “kilogram exception” is no more an exception, and the cherry on the cake is that this feature is now available when you define a fundamental unit:

/// <summary>

/// Define a new fundamental unit of measure.

/// Just like DefineAlias, the same fundamental unit can be redefined multiple times

/// as long as it is actually the same: for fundamental units, the Name **(and the**

/// **normalizedPrefix if any)** must be exaclty the same.

/// </summary>

/// <param name="abbreviation">

/// The unit of measure abbreviation.

/// This is the key that is used. It must not be null or empty.

/// </param>

/// <param name="name">The full name. Must not be null or empty.</param>

**/// <param name="normalizedPrefix">**

**/// Optional prefix to be used for units where the normalized unit should not be the**

**/// FundamentalMeasureUnit but one of its PrefixedMeasureUnit.**

**/// This is the case for the "g"/"Gram" and the "kg"/"Kilogram".**

**/// Defaults to MeasureStandardPrefix.None: by default a fundamental unit is the**

**/// normalized one.**

**/// </param>**

/// <returns>The fundamental unit of measure.</returns>

public static FundamentalMeasureUnit DefineFundamental(

string abbreviation,

string name,

MeasureStandardPrefix normalizedPrefix = null )

This new normalization behavior is now compliant with the S.I:

var metre = MeasureUnit.Metre;

var second = MeasureUnit.Second;

var kilogram = MeasureUnit.Kilogram;

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

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

// With FundamentalMeasureUnit as the only normalized form:

//newton.Normalization.Abbreviation.Should().Be( "**g.m.s-2**" );

//newton.NormalizationFactor.Should().Be( **new FullFactor( new ExpFactor( 0, 3 )** ) );

// A PrefixedMeasureUnit can be the normalized form:

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

newton.NormalizationFactor.Should().Be( FullFactor.Neutral );

It’s time to explain the normalization code. The actual normalized form is computed on-demand if, and only if, the unit is not (by construction) defined as THE normalized form and if it has not been already computed:

public bool IsNormalized => \_normalization == this;

public FullFactor NormalizationFactor

{

get

{

if( \_normalization == null )

{

(\_normalization, \_normalizationFactor) = GetNormalization();

}

return \_normalizationFactor;

}

}

public MeasureUnit Normalization

{

get

{

if( \_normalization == null )

{

(\_normalization, \_normalizationFactor) = GetNormalization();

}

return \_normalization;

}

}

We are using value tuple here to return both the factor and the normalized unit from the actual computation. At the top level of the MeasureUnit, the potentially multiple units’ normalized forms are combined into one list of deduplicated units AND, at the same time, the final normalization factor is computed by multiplying all the normalization factors with an initial neutral one.

The Combinator is a small private struct that encapsulates deduplication. It has been developed on Day 1 (with some refactoring since its creation) and is quite simple (its code can be found here: <https://github.com/Invenietis/CK-MeasureUnit/blob/master/CK.MeasureUnit/MeasureUnit.Combinator.cs>).

private protected virtual (MeasureUnit, FullFactor) GetNormalization()

{

Combinator measures = new Combinator( null );

var f = \_units.Aggregate( FullFactor.Neutral, ( acc, m ) =>

{

measures.Add( m.Normalization.MeasureUnits );

return acc.Multiply( m.NormalizationFactor );

} );

return (measures.GetResult(), f);

}

This is the general implementation, for the combined unit of measure. Exponent and Atomic units override this behavior and this is, up to me, a demonstration of the beauty of the good old standard object paradigm.

The normalized form of an ExponentMeasureUnit is the normalized form of its atomic measure elevated to its own power. And its normalization factor is the one of its atomic measure also elevated to its own power.

private protected override (MeasureUnit, FullFactor) GetNormalization()

{

return (

AtomicMeasureUnit.Normalization.Power( Exponent ),

AtomicMeasureUnit.NormalizationFactor.Power( Exponent )

);

}

For the AliasMeasureUnit, it is even simpler: its normalized form is the one of its definition and its normalization factor is multiplied by its own DefinitionFactor.

private protected override (MeasureUnit, FullFactor) GetNormalization()

{

return (

Definition.Normalization,

Definition.NormalizationFactor.Multiply( DefinitionFactor )

);

}

And finally, the PrefixedMeasureUnit computes its normalization factor by applying its prefix’ factor and its adjustment factor (the adjustment factor kindly handles silly prefixes like “DeciGiga” – see Day 2).

private protected override (MeasureUnit, FullFactor) GetNormalization()

{

return (

AtomicMeasureUnit.Normalization,

AtomicMeasureUnit.NormalizationFactor

.Multiply( Prefix.Factor )

.Multiply( AdjustmentFactor )

);

}

And this is it.

You may wonder “what about thread safety? This seems absolutely not thread safe!”.

This is totally thread-safe (the whole library is totally thread-safe). But since it could be quite a long discussion, I’ll talk about this later. I’m thrilled to start implementing quantities…

Quantity is a simple immutable value type that simply combines a double value and a unit. Operations on quantities are straightforward to implement. Below is the core of the Quantity type:

public struct Quantity

{

public readonly double Value;

public readonly MeasureUnit Unit;

public Quantity( double v, MeasureUnit u )

{

Value = v;

Unit = u;

}

public Quantity Multiply( Quantity q )

=> new Quantity( Value \* q.Value, Unit \* q.Unit );

public Quantity DivideBy( Quantity q )

=> new Quantity( Value / q.Value, Unit / q.Unit );

public Quantity Invert()

=> new Quantity( 1.0 / Value, Unit.Invert() );

public Quantity Power( int exp )

=> new Quantity( Math.Pow( Value, exp ), Unit.Power( exp ) );

public bool CanConvertTo( MeasureUnit u )

=> Unit.Normalization == u.Normalization;

public Quantity ConvertTo( MeasureUnit u )

{

if( !CanConvertTo( u ) )

{

throw new ArgumentException( $"Can not convert from '{Unit}' to '{u}'." );

}

FullFactor ratio = Unit.NormalizationFactor.DivideBy( u.NormalizationFactor );

return new Quantity( Value \* ratio.ToDouble(), u );

}

public static Quantity operator /( Quantity o1, Quantity o2 )

=> o1.DivideBy( o2 );

public static Quantity operator \*( Quantity o1, Quantity o2 )

=> o1.Multiply( o2 );

public static Quantity operator ^( Quantity o, int exp )

=> o.Power( exp );

public string ToString( IFormatProvider formatProvider )

=> Value.ToString( formatProvider ) + " " + Unit.ToString();

public override string ToString() => $"{Value} {Unit}";

}

With the help of a WithUnit extension methods on int and double (I generally avoid polluting basic types with extension methods but here I think it makes sense), it works:

[Test]

public void simple\_operations()

{

var metre = MeasureUnit.Metre;

var second = MeasureUnit.Second;

var kilometre = MeasureStandardPrefix.Kilo[metre];

var minute = MeasureUnit.DefineAlias( "min", "Minute", 60, second );

var hour = MeasureUnit.DefineAlias( "h", "Hour", 60, minute );

var speed = kilometre / hour;

var myDistance = 3.WithUnit( kilometre );

var mySpeed = 6.WithUnit( speed );

var myTime = myDistance / mySpeed;

myTime.ToString( CultureInfo.InvariantCulture ).Should().Be( "0.5 h" );

myTime.CanConvertTo( minute ).Should().BeTrue();

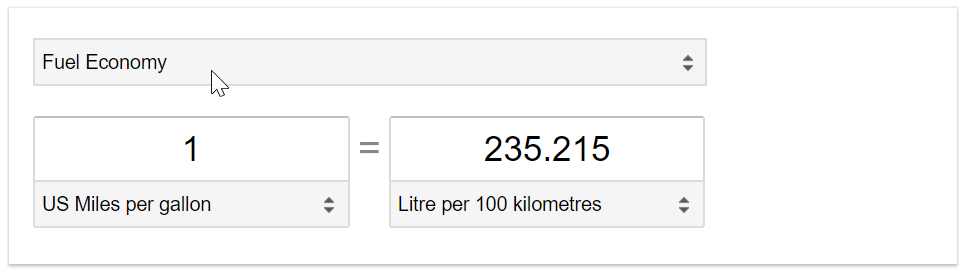
myTime.ConvertTo( minute ).ToString().Should().Be( "30 min" );

myTime.CanConvertTo( second ).Should().BeTrue();

myTime.ConvertTo( second ).ToString().Should().Be( "1800 s" );

}

And for fun (and more test), let’s check this against “poetic units”. The Google unit converter states that:



I choose the US measures because it is defined by inches (the US gallon is defined as 231 cubic inches) and hence is funnier for us than the imperial gallon that is directly related to meters.

An inch is (for sure) 2.54 centimetre. A US mile is now the International Mile that is 1.609344 km. But be careful, there are tons of different miles in US, UK and other countries (see <https://en.wikipedia.org/wiki/Mile>).

For us, French people (who inspired the modern S.I by the way), this is pure poetry!

public void poetic\_units()

{

var metre = MeasureUnit.Metre;

var decimetre = MeasureStandardPrefix.Deci[metre];

var centimetre = MeasureStandardPrefix.Centi[metre];

var kilometre = MeasureStandardPrefix.Kilo[metre];

var hundredKilometre = MeasureStandardPrefix.Hecto[kilometre];

var litre = decimetre ^ 3;

var inch = MeasureUnit.DefineAlias( "in", "Inch", 2.54, centimetre );

var gallon = MeasureUnit.DefineAlias( "gal", "US Gallon", 231, inch ^ 3 );

var mile = MeasureUnit.DefineAlias( "mile", "Mile", 1.609344, kilometre );

var milesPerGalon = mile / gallon;

var litrePerHundredKilometre = litre / hundredKilometre;

var oneMilesPerGallon = 1.WithUnit( milesPerGalon );

oneMilesPerGallon.CanConvertTo( litrePerHundredKilometre ).Should().BeTrue();

var result = oneMilesPerGallon.ConvertTo( litrePerHundredKilometre );

result.Value.Should().BeApproximately( 235.215, 1e-3 );

}

THIS FAILS!

oneMilesPerGallon.CanConvertTo( litrePerHundredKilometre) is false… simply because these two units are inverted:

* Miles/Gallon ≡ distance / distance ^3 ≡ distance^2
* Litre/Kilometre ≡ distance ^ 3 / distance ≡ distance^-2.

We can handle this (and make our library a little bit smarter):

public bool CanConvertTo( MeasureUnit u )

=> Unit.Normalization == u.Normalization

|| Unit.Normalization == u.Normalization.Invert();

public Quantity ConvertTo( MeasureUnit u )

{

if( !CanConvertTo( u ) )

{

throw new ArgumentException( $"Can not convert from '{Unit}' to '{u}'." );

}

if( Unit.Normalization == u.Normalization )

{

FullFactor ratio = Unit.NormalizationFactor.DivideBy( u.NormalizationFactor );

return new Quantity( Value \* ratio.ToDouble(), u );

}

else

{

FullFactor ratio = Unit.NormalizationFactor.Multiply( u.NormalizationFactor );

return new Quantity( 1/(Value \* ratio.ToDouble()), u );

}

}

And this works!

While implementing the first test for quantity I used (to complicate a little bit the test) the following definition of the hour:

var minute = MeasureUnit.DefineAlias( "min", "Minute", 60, second );

var hour = MeasureUnit.DefineAlias( "h", "Hour", 60, minute );

A test from yesterday uses:

var hour = MeasureUnit.DefineAlias( "h", "Hour", 60 \* 60, second );

And tests are now broken: the first to execute registers its definition of the hour, the second redefinition is (rightly) detected as not being the same. The “culprit” is here (internal code):

static AliasMeasureUnit RegisterAlias(string a, string n, FullFactor f, MeasureUnit d)

{

return Register( abbreviation: a,

name: n,

creator: () => new AliasMeasureUnit( a, n, f, d ),

checker: m => m.DefinitionFactor == f && m.Definition == d

);

}

The last parameter is the “checker”: a function that the Register core function calls to check that the actually registered measure is the “same” (so that you cannot define different units of measure with the same abbreviation – abbreviation is the key).

Are the two “hours” defined above the same or not?

Unfortunately, it depends on your business needs.

However, it is easy to relax the check here by challenging instead of the exact definition, its normalization:

checker: m => m.Definition.Normalization == d.Normalization

&& m.NormalizationFactor == d.NormalizationFactor.Multiply( f ) );

Should we implement the strict or the relaxed check? Again, it depends on your business needs.

When developing a library that aims to be used in different contexts, my recommendation is to be very cautious about taking such abrupt decisions, I always try to NOT anchor such behaviors/choices deep inside the code (and a difficult part of library development is to identify those choices).

Before tackling this, I’d like to refactor the code. It works well but, up to me, there is a huge issue with current architecture: there is one and only one, global, context for the unit of measures. If you took the time to read this <https://en.wikipedia.org/wiki/Gallon> or this <https://en.wikipedia.org/wiki/Ounce> or <https://en.wikipedia.org/wiki/Ton-force>, can you imagine that an awful singleton will be able to work in a web application that can be used by different customers or manipulates units of measure from different fields?

“Yes… but a singleton is so easy to use!”

Right. The refactoring will preserve the current API: it will be the Default measure context. But you’ll be able to create as many independent MeasureContext as you need (some of them not having the Metre/Gram, etc. default units of measure).

Refactoring done. The static “easy to use” standard units exposed by the MeasureUnit are now simple relays to the StandardMeasureContext.Default singleton:

/// <summary>

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

/// </summary>

public static FundamentalMeasureUnit Unit => StandardMeasureContext.Default.Unit;

Whenever measures are mixed from different contexts, an exception is raised:

[Test]

public void measures\_from\_different\_contexts\_must\_not\_interfere()

{

var kilogram = MeasureUnit.Kilogram;

kilogram.Should().BeSameAs( StandardMeasureContext.Default.Kilogram );

var another = new StandardMeasureContext();

var anotherKilogram = another.Kilogram;

anotherKilogram.Should().NotBeSameAs( kilogram );

another.Invoking( c => c.DefineAlias( "derived", "Derived", 2, kilogram))

.Should().Throw<Exception>()

.Where( ex => ex.Message.Contains( "Context mismatch" ) );

StandardMeasureContext.Default

.Invoking( c => c.DefineAlias( "derived", "Derived", 2, anotherKilogram )

.Should().Throw<Exception>()

.Where( ex => ex.Message.Contains( "Context mismatch" ) );

Action a = () => Console.WriteLine( kilogram / anotherKilogram );

a.Should().Throw<Exception>()

.Where( ex => ex.Message.Contains( "Context mismatch" ) );

}

The previous API has not changed but now you can create independent MeasureContext with no unit inside except its “None”, or use the StandardMeasureContext that exposes and already define the Unit, Bit, Byte, and the 7 S.I units.

Enough for today.

# Day 6 – Quantity in all its Glory

I started the day by enhancing the Quantity with operators so that this is now supported:

[Test]

public void Quantity\_operators\_override()

{

var d1 = 1.WithUnit( MeasureUnit.Metre );

var d2 = 2.WithUnit( MeasureUnit.Metre );

var d3 = d1 + d2;

d3.Value.Should().Be( 3.0 );

var d6 = d3 \* 2;

d6.Value.Should().Be( 6.0 );

var s36 = d6 ^ 2;

s36.Value.Should().Be( 36.0 );

s36.Unit.Should().Be( MeasureUnit.Metre \* MeasureUnit.Metre );

var sM36 = -s36;

sM36.Value.Should().Be( -36.0 );

sM36.Unit.Should().Be( MeasureUnit.Metre \* MeasureUnit.Metre );

(s36 - sM36).Value.Should().Be( 72.0 );

(s36 + sM36).Value.Should().Be( 0.0 );

var s9 = d3 \* d3;

s9.Value.Should().Be( 9.0 );

s9.Unit.Should().Be( MeasureUnit.Metre \* MeasureUnit.Metre );

var r4 = s36 / s9;

r4.Value.Should().Be( 4 );

r4.Unit.Should().Be( MeasureUnit.None );

var s144 = r4 \* s36;

s144.Value.Should().Be( 144.0 );

s144.Unit.Should().Be( MeasureUnit.Metre \* MeasureUnit.Metre );

var v27 = d3 ^ 3;

v27.Value.Should().Be( 27.0 );

v27.Unit.Should().Be( MeasureUnit.Metre ^ 3 );

(v27 / s9).Should().Be( d3 );

((v27 / s9) == d3).Should().BeTrue();

((v27 / s9) != d3).Should().BeFalse();

}

And then I had an… issue. Any C#/Java/… developer knows that whenever you define Equals (here implementing the IEquatable<Quantity> interface) on an object, it is expected that you also override the Equals( object ) and the GetHashCode() methods.

First, we need to define the equality of quantities:

* They must be convertible to each other units.
* Their values must be the same when converted into one (or the other) units.

Thanks to this, “10 dm” (decimeter) will be equal to “1 m”, “0.1 hm” (hectometer), “0.001 km”, etc.

So far so good, implementing equality implies to convert the other quantity into the unit of this quantity and check the resulting value. Of course, if the other quantity cannot be converted the two quantities are not equal:

public bool Equals( Quantity other )

=> other.CanConvertTo( Unit ) && other.ConvertTo( Unit ).Value == Value;

And at the object level:

public override bool Equals( object obj ) => obj is Quantity q && Equals( q );

Then comes the GetHashCode… We do not have any “other” here. We must compute a hash code that must match the Equal rule but for any Quantity in “isolation”. The solution is to use the Unit.Normalization and NormalizationFactor as the “target” quantity:

public override int GetHashCode()

=> ConvertTo( Unit.Normalization ).Value.GetHashCode();

This should work… but actually not. This test is awfully red:

[Test]

public void Quantity\_with\_alias\_and\_prefixed\_units()

{

var metre = MeasureUnit.Metre;

var decametre = MeasureStandardPrefix.Deca[metre];

var decimetre = MeasureStandardPrefix.Deci[metre];

var dm1 = 1.WithUnit( decimetre );

var dam1 = 1.WithUnit( decametre );

var dm101 = dm1 + dam1;

var dam1Dot01 = dam1 + dm1;

dm101.ToString( CultureInfo.InvariantCulture ).Should().Be( "101 dm" );

dam1Dot01.ToString( CultureInfo.InvariantCulture ).Should().Be( "1.01 dam" );

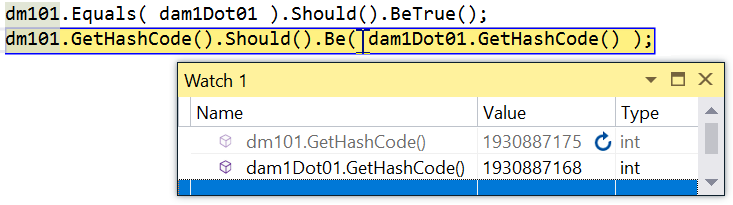
dm101.Equals( dam1Dot01 ).Should().BeTrue();

**dm101.GetHashCode().Should().Be( dam1Dot01.GetHashCode() );**

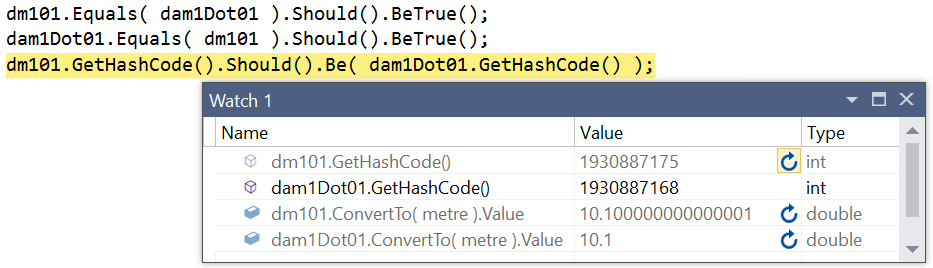
}

What? How? Any idea?

Welcome to the floating point hell!



The two double’s hash codes are slightly different. And so are their actual values:



This happens. But the fun here is that:

* The conversions between the two give exact values (both directions).
* The conversion to their common normalized unit (the metre), when expressed as strings, are also the same:

dm101.ConvertTo( metre ).ToString().Should().Be( "10.1 m" );

dam1Dot01.ConvertTo( metre ).ToString().Should().Be( "10.1 m" );

The latter is due to the double.ToString() implementation that “cleverly” optimizes its output by rounding/cleaning the value.

What are our options here?

* Forget the hash code
  + And keeping the Equals

🡺 Any dictionary that will use a Quantity as a key will fail miserably.

* + And forgetting also the Equals

🡺 This is weird. One expects a value type named “Quantity” to support equality.

* Having an hash code and the current Equals implementation
  + As it is currently implemented.

🡺 Any dictionary that will use a Quantity as a key will fail miserably.

* + By returning 0

🡺 This is perfectly valid (will not introduce any bug) but using such an object as a key in a dictionary will heavily degrade performances (O(1) to O(n)).

* + By calling Math.Round (<https://docs.microsoft.com/en-us/dotnet/api/system.math.round>) on the converted result.

🡺 What kind of rounding? What precision? I have no idea.  
GetHashCode() is a method that must run fast. Conversion alone costs, adding a subsequent rounding may not be a great idea.

* Finding another Equals and GetHashCode

🡺 Object equality has to be “coherent”, to follow the “Principle of least astonishment”, not necessarily be “exact” (what does “exact” mean with doubles anyway?).

And the idea is simply to use the string representation of the normalized quantity. Two quantities will be equal if and only if their normalized representations are the same. This NormalizedString is now exposed on the Quantity because it is a very practical and useful property.

I choose to store the string representation to avoid recomputing it each time we need it and it is lazily initialized. The Quantity struct is now a little bit heavier (one double – 8 bytes, and 2 object references – 2 x 4 or 8 bytes), this is the price to pay.

The lazy initialization does not protect concurrent access (with a costly lock or even a Compare-And-Swap - Interlocked instruction): the worst thing that may happen is that we may compute the \_normalized field twice.

public string ToNormalizedString()

{

if( \_normalized == null )

{

\_normalized = ConvertTo( Unit.Normalization )

.ToString( CultureInfo.InvariantCulture );

}

return \_normalized;

}

public override int GetHashCode() => ToNormalizedString().GetHashCode();

public bool Equals( Quantity other )

=> ToNormalizedString() == other.ToNormalizedString();

Equality issue solved. But there is two more thing that bothers me:

* the default struct Quantity - new Quantity() - has a 0.0 value and a null Unit. This null quantity Unit is currently not handled and this is a time bomb.
* comparable: should Quantity be comparable? Unfortunately, yes, as much as being equatable.