Domain modeling with monoids

Cyrille Martraire, Arolla, 2018

It is unlucky the word “Monoid” is so awkward, as it represents a concept that is both ubiquitous and actually simple.

Monoids are all about *composability*. It's about having abstractions in the small that compose infinitely in the large.

You are already familiar with many cases of typical monoidal composeablity, such as everything *group-by*, or everything *reduce*. What I want to emphasize is that Monoids happen to be frequent occurrences in business domains, and that you should spot them and exploit them.

Since I’ve first talked about my enthusiasm for monoids at conferences around the world, I’ve received multiple positive feedbacks of concrete situations where monoids helped teams redesign complicated parts of their domain models into something much simpler. And they would then say that the new design is “more elegant”, with a smile.

Just one important warning: please don’t confuse monoids with monads. Monoids are much easier to understand than monads. For the rest of the text, we will keep monads out of scope, to focus solely on monoids and their friends.

What are Monoids?

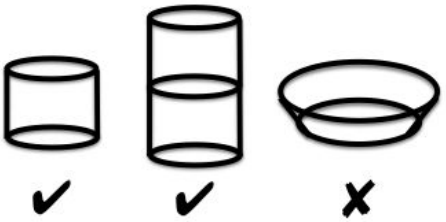
Monoids come from a part of mathematics called abstract algebra. It's totally intimidating, but it really doesn't have to, at least not for monoids, which are creatures so simple that kids pretty much master them by age of 2 (without knowing of course)

So what are monoids? A monoid is a mathematical structure. First we start with a simple set. A set is just a criterion to decide if elements belong or not to the set. Yes it’s a bit circular a definition but you get the idea.

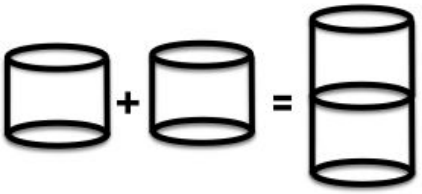
I usually explain monoids through glasses of beer, but here I will use plumbery pipes. For example, let's define a set of pipes this way:

”The set of pipes with a hole this size”

Given this definition of our set, now we can test if various elements belong to it or not.



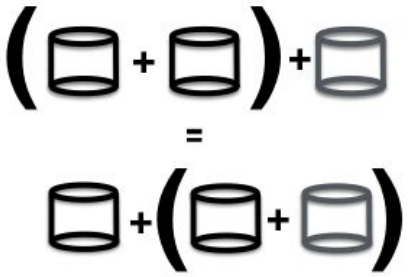
On top of this set, we define an operation, that we can call "combine", “append”, "merge", or "add", that takes two elements and combines[1] them.



One funny thing we notice now is that given two elements from the set, the result is always... in the set too! That sounds like nothing, but that's what Closure of Operations is all about: the set is closed under this operation. It matters. We have a little system that's all about itself, always. That's cool.

But there's more to it. If you first combine the first two pipes together, then combine the result with the third pipe, or if you first combine the last two pipes then you combine the first pipe with it, then you end up with the exact same result. We're not talking about changing the ordering of the pipes, just the ordering of combining them. It's like putting the parenthesis anywhere doesn't change the result. This is called Associativity, and is important.

[1] In the world of pipes it could be named ‘weld”.

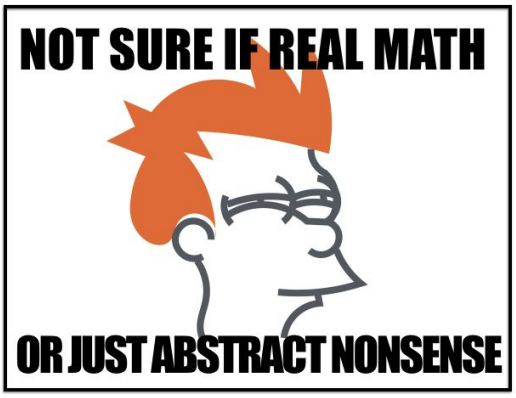


By the way changing the ordering of the pipes would be called Commutativity, and we have it if they all are identical, but this property is not as frequent.

So we have a set of elements, an operation with results that are all in the set, and that is associative. To really be a monoid, you need one more thing: you have to define one more kind of pipe, that is invisible, so I can't show it to you. But believe me, it exists, because as a developer I can create the system the way I prefer it to be. This special element belongs to the set: it has a hole the right size. So whenever I combine this special element with any other, the result is just the other element itself. It doesn't change anything. That's why we call it the neutral, or identity element.

And voilà! A set, an operation, closure of this operation, associativity, and neutral element: that's the formal definition of a monoid!

Ok at this point, you are perhaps like:



But stay with me, we'll see how this really closely relates to your daily job.

So what?

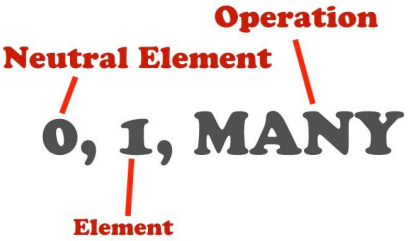
So when I started explaining monoids to people I also explained it to my wife. She instantly got it, but then asked: what for?

So I was the one wondering for a few seconds. Why does it matter to me? I believe it's all about dealing with encapsulating some diversity inside the structure.

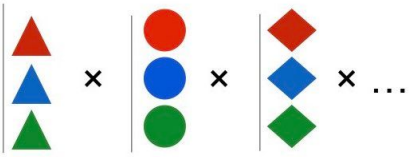
You probably know the old joke in programming:

> There are only three numbers in programming: 0, 1, and MANY.

That's so true. But this also illustrates a very common kind of diversity we face all the time: the singular, the plural, and the absence. Monoids naturally represent the singular, with the elements of the set. It also deals with the plural, thanks to the *combine* operation that can take a plural and turn it back into an element as usual. And the neutral element takes care of the absence case, and it also belongs to the set. So monoids encapsulate this diversity inside their structure, so that from the outside you don't have to care about it. That's a great idea to fight the battle against complexity.

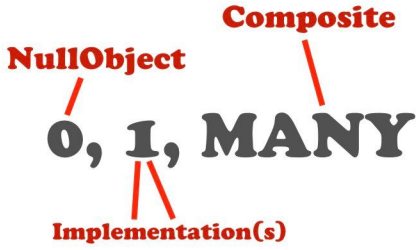


In the wild real life problems that we have, if we're not careful, we have to deal with various concepts, each potentially having their own diversity of being either singular, plural, or nothing. In the worst case, you'd end up with the cartesian product of all cases. It doesn't scale well.



Once you encapsulate this diversity inside a monoid for each concept, then you only have one case for each, and together it remains one single case.

If you apply that often, then you can deal with high levels of complexity with ease. Monoids help scale in complexity. In fact, if you're comfortable in object-oriented programming, you may recognize something familiar you're doing already: for a given interface, I often find myself using the Composite pattern to deal with the plural, and the NullObject pattern to deal with the absence. These patterns help deal with singular, plural and absence consistently, so that the caller code doesn't even have to know. That's similar in purpose.



Examples Please!

You already know a lot of monoids in your programming language:

- **Integer** with **addition**: Integers are closed under addition: int + int = int. They're associative: (3+5)+2=3+(5+2), and their neutral element is 0, since any integer plus zero gives the same integer.
- **Lists** with list **append** operation: List + List = List is closed under this appending, which is associative: (a)+(b,c)=(a, b)+(c). And the empty list is the neutral element here.
- A special case of lists, **Strings** with **concatenation**: "hello" + "world" is a string too. It's associative: "cy"+"ri"+"lle", and the neutral element is the empty string.

Note that integers can also form a monoid with the multiplication operation, in which case the neutral element would be 1. But natural integers with subtraction do not form a monoid, because 3 - 5 is not in the set of natural integers but it would in the set of integers.

All this is not difficult. Still, such a simple thing is a key to very complex behaviors. It's also the key to infinite scalability of space (think Hadoop), and the key to infinite incremental scalability (think Storm). There's one joke in Big Data circles:

> If you're doing Big Data and you don't know what an abelian group is, then you do it wrong!

It's all about *composability*, which is highly desirable pretty much everywhere. .

Implementing monoids in your usual programming language

So how do we implement monoids in plan Java code?

> Monoids are typical Functional Programming; In Functional Programming everything is a value; Therefore: Monoids are values!

That's a solid proof that monoid are value objects. Seriously though they do have to be value objects, i.e. immutable and equality by value. But monoid objects don't have to be anemic, with just data. They are supposed to have behavior, and in particular behavior that compose, like lengths, where we want to be able to write: "18 m + 16 m = 34 m". The corresponding code for this method would be:

```
public Length add(Length other){
   return new Length(
      value + other.value);
}
```

This add() method returns a new instance, as value objects should do. It must not perform any side-effect, as advocated by the DDD "Side-Effect-Free Functions" pattern. Immutability and Side-Effect-Free Functions together are good taste! That should be your default style of programming, unless you really have to do otherwise.

In addition, being immutable and side-effect-free means that testing is a no-brainer: just pass data in,

and assert the result out. Nothing else can happen to make it more complicated.

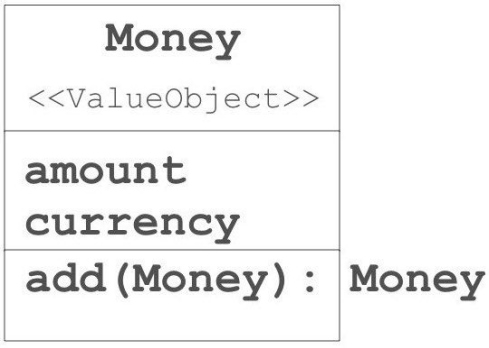
Monoids in domain modeling

Domain-specific lists, like a mailing list defined as a list of emails addresses, can form a monoid at least twice, once with the union operation, and a second time with the intersection operation. The neutral element would be nobody() in the former case, and everybody() in the latter case. Note the naming of the neutral elements that is domain-specific, instead of more generic names like *empty* or *all*. But we could go further and rename the intersection() operation into a word like *overlapping* is this was the way the domain experts talked about this problem.

Money and Quantity

Let's start with the good old Money analysis pattern, from Martin Fowler: (EUR, 25) + (EUR, 30) = (EUR, 55). And in case the currencies don't match, we would throw an exception in the add method. Note that throwing exception indeed break perfect composability, but in practice we could deal with some, as long as they only reveal coding mistakes.

Our money class would be defined in UML like this:



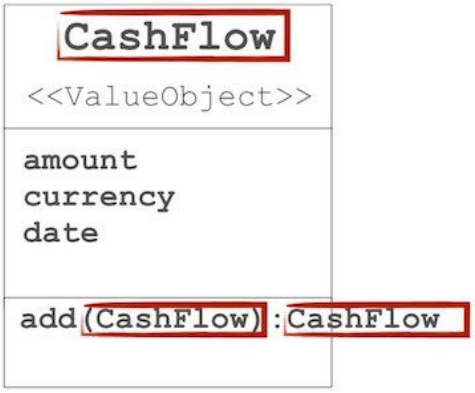
The Money pattern is indeed a special case of the more general Quantity analysis pattern: "Representing dimensioned values with both their amount and their unit" (Fowler).

Cashflows and sequences of cashflows

Now that we have a money amount, we can make it into a cashflow, by adding a date:

(EUR, 25, TODAY)
+ (EUR, 30, TODAY)
= (EUR, 55, TODAY)

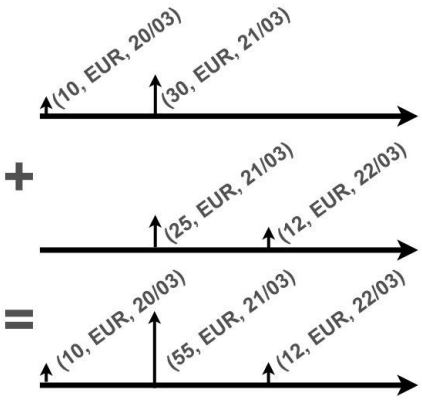
And again we could throw an exception if the dates don't match.



Looking at the UML class diagram, it's striking that the class only refers to its own type, or primitives (in the constructor, not shown here). Methods take Cashflow as parameter, and return Cashflow, and nothing else.

This is what Closure of Operation means in code. This type is egotistic, it only talks about itself. That's a desirable quality for code, as advocated in the DDD book, and it's one of the mandatory properties of a monoid.

But why stop there? We typically deal with many cashflows that go together, and we also think of them as stuff we can add:



So once again, we want to be able to write the code the exact same way:

Cashflow Sequence
+ Cashflow Sequence
= Cashflow Sequence.

At this point you get the picture: monoid are all about what we could call an arithmetics of objects.

Note that the addition operation in the Cashflow Sequences above is in basically the list concatenation (e.g. *addAll()* in Java), where cashflows on the same date and on the same currency are then added together using the addition operation of the cashflow themselves.

Ranges

A range of numbers or a range of dates can be seen as a monoid, for example with the compact-union operation, and the empty range as the neutral element:

[1, 3] Union [2, 4] = [1, 4] // compact union
[1, 3] Union][= [1, 3] // neutral element

By defining the operation of “compact” union:

```
public final class Range{
   private final int min;
   private final int max;
   public final static EMPTY
                    = new Range();

   public Range union(Range other){
       return new Range(
          min(this.min, other.min),
          max(this.max,other.max));
   }
}
```

Note that the internal implementation can absolutely delegate the work to some off-the-shelf implementation, e.g. some well-known, well-tested open-source library.

Predicates

Predicates are natural monoids, with logical AND and the ALWAYS_TRUE predicate, or with logical OR and the ALWAYS_FALSE predicate.

Grants

But even unexpected stuff like read/write/execute grants can form a monoid with some merge operation defined for example as "the most secure wins"::

r merge w = r
w merge x = w

The implementation could be an enum and perform a MIN on the internal ordering of each value.

```
public final enum Grant{
   R, W, X;
   public Grant merge(Grant other){
       return min(this.ordinal(),
                  other.ordinal());
   }
}
```

Of course it’s up to your domain expert to decide which exact behavior is expected here, and how the operation should be named.

Monoids of monoids are monoids

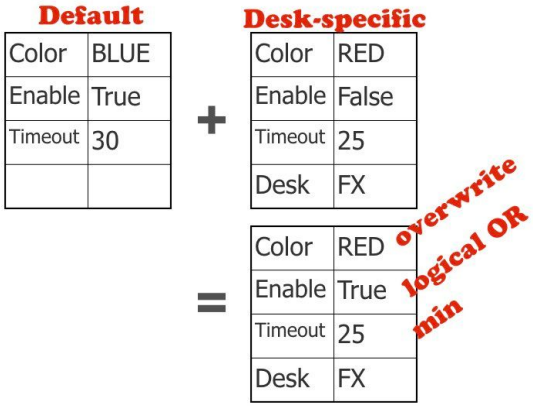
Nesting monoids can easily lead to monoids. For example in many systems you have configuration maps for the settings of an application. You often have a default hardcoded one, then by order of precedence one by department, then another by desk, and ultimately one by user. This leads naturally to a monoid form:

MonoidMap + MonoidMap = MonoidMap

One simple way to do that is just combine the maps with the LAST ONE WINS policy:

```
public MonoidMap append(
              MonoidMap other) {
  Map<String, Object> result
      = new HashMap<>(this.map);
  result.putAll(other.map);
  return new MonoidMap(result);
}
```

But we can go further if all values are also monoids, and let each value make its own monoidal magic:



In our example, colors are combined by an OVERWRITE operation (last value wins), Enable values are combined by a logical OR operation, while Timeout values are combined by an integer MIN operation. You can see here that all the value are monoids by themselves with these operations. By defining the map-level combine operation (here noted +) by delegating to the monoid operation of each value, in parallel for each key, then we also have the configuration maps as monoids. Their neutral element could be either an empty map, or a map with all the neutral elements of each type of value.

```
public NestedMonoidMap append(
             NestedMonoidMap other) {
  Map<String, Monoid<?>> result
        = new HashMap<>(map);
  for (String key:other.map.keySet()){
   Monoid value = map.get(key);
   Monoid value2 = other.map.get(key);
   result.put(key, value == null ?
     value2 :
    (Monoid)value.append(value2));
  }
  return new NestedMonoidMap(result);
}
```

Of course in this example, each value would have to be itself a monoid, with its own specific way to append or merge.

What I like in this example is also that it shows that value objects don't have to be small-ish. We can have huge objects trees as values and as monoids, and it works well. And don't obsess too much about the memory allocation here, most of the values are reused many times, really.

Non Linear

But not everything is that easy to model as monoids. For example, if you have to deal with partial averages and want to compose them into a bigger average, you cannot write: Average + Average as it would just be WRONG:

Average + Average = WRONG

Average calculation just doesn't compose at all. This makes my panda sad.

But if you really want to make it into a monoid, then you can do it! The usual trick is to go back to the intermediate calculation, in which you can find some composable intermediate sub-calculations:

avg = sum / count

And it turns out that put together as a tupple, it composes quite well, using a tuple-level addition defined as the addition of each term:

(sum, count) + (sum, count) = (sum, count)

Which internally becomes:

```
(sum_0, count_0)
+ (sum_1, count_1)
= (sum_0 + sum_1, count_0 + count_1)
```

So you can combine tuples at large scale, across many nodes for example, and then when you get the final result as a tuple, then you just finish the work by taking the average out of it by actually doing the division sum/count.

```
public class Average {
  private final int count;
  private final int sum;
  public static final Average NEUTRAL
              = new Average(0, 0);

  public static final
      Average of(int... values) {
    return new Average(
          values.length,
          stream(values).sum());
  }

  private Average(int count, int sum){
    this.count = count;
    this.sum = sum;
  }

  public double average() {
    return (double) sum / count;
  }

  public int count() {
    return count;
  }

  public Average add(Average other) {
    return new Average(
       count + other.count,
       sum + other.sum);
  }
  …// hashcode, equals, toString
}
```

And if you need the standard deviation, you can do the same trick, just by adding the sum of the values at the power of two (sum2):

```
(sum2, sum, count)
+ (sum2, sum, count)
= (sum2, sum, count),
```

Which internally becomes:

```
(sum2_0, sum_0, count_0)
+ (sum2_1, sum_1, count_1)
= (sum2_0,  + sum2_1,
   sum_0 + sum_1,
   count_0 + count_1)
```

as there's a formula to get the standard deviation out of that:

> “the standard deviation is equal to the square root of the difference between the average of the squares of the values and the square of the average value.”

STD = square root[1/N.Sum(x^2) - (1/N.Sum(x))^2]

Monoids don’t have to use addition, here’s an example of a monoid of ratios with the operation of multiplication:

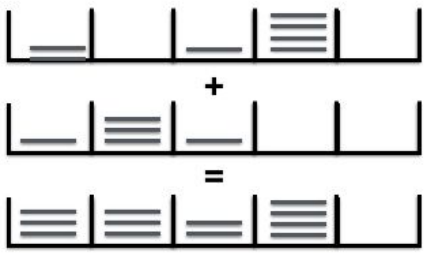
```
public class Ratio {
  private final int numerator;
  private final int denumerator;
  public static final Ratio NEUTRAL =
                    new Ratio(1, 1);

  public Ratio(
      int numerator, int denumerator){
    this.numerator = numerator;
    this.denumerator = denumerator;
  }

  public double ratio() {
    return numerator / denumerator;
  }

  public Ratio multiply(Ratio other) {
    return new Ratio(
     numerator * other.numerator,
     denumerator * other.denumerator);
  }
  ...// hashcode, equals, toString
}
```

Over the years I've grown the confidence that anything can be made into a monoid, with these kinds of tricks. Histograms with fixed buckets naturally combine, bucket by bucket:



The corresponding code for the add operation adds the number of elements in each respective bucket:

```
public Histogram add(Histogram other)
{
  if (buckets.length !=
      other.buckets.length) {
   throw new IllegalArgumentException(
    "Histograms must have same size");
  }
  int[] bins = new
int[buckets.length];
  for
    (int i = 0; i < bins.length; i++){
    bins[i]
    = buckets[i] + other.buckets[i];
  }
  return new Histogram(bins);
}
```

If histograms have heterogeneous buckets, they can be made to compose using approximations (eg curves like splines) that compose.

Moving average don't compose unless you keep all their respective memories and combine them just like the histograms. But by looking into small-memory microcontroller litterature you can find alternative ways to calculate them that compose with much less memory footprint, e.g. using just a couple of registers.

Note that one potential impediment to making an arbitrary calculation into a monoid could be concerns such as being ill-conditioned, or value overflow, but I never had this issue myself.

Monoids And Friends: Applications Notes

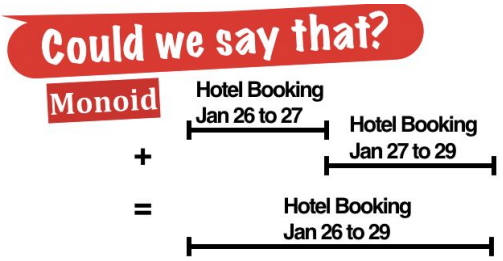
As shown with the pipes, monoids are ubiquitous in our daily lives, and are part of our universal language to describe things, even without ignoring their abstract definition. Everybody knows how to stack glasses or chairs. My kids know how to combine wooden trains together to create longer trains.

Declarative style

This ability to compose stuff is part of our mental models, and as such can be part of our Ubiquitous Language in the DDD sense. For example in the hotel booking domain, we could say that a booking from January 21 to 23 combined to another booking in the same hotel from January 23 to 24 is equivalent to one single booking from January 21 to 24:

Booking [21, 21]
+ Booking [23, 24]
= Booking [21, 24]

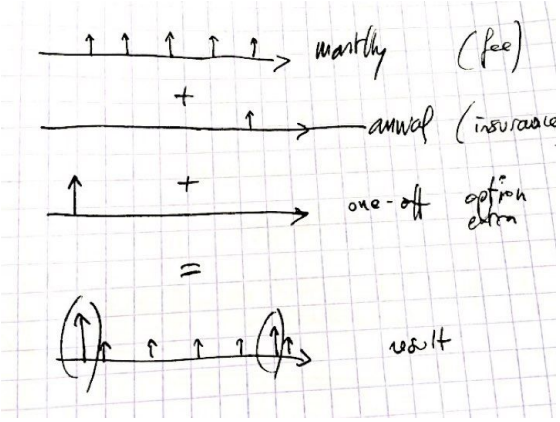
Which we could sketch like this, as ranges with some union operation:



The code for the operation would just take the min and max of both dates and check they share one date. It would probably be a commutative operation in this case. And in programming languages with operator overloading like Scala, we could really replace expressions like *a.add(b)* by *a + b*

Using monoids helps having a more declarative style in our code, another point that is advocated for by Eric Evans in the DDD book.

Let's consider another example of price plans of mobile phones. There's a potential fixed monthly fee, a potential annual fee for things like insurance, some potential one-off fees for extra options that you pay when you activate it etc. For a given price plan for a given customer, you have to select the cash flows sequences that match, then add them to create the invoice. We could draw the domain problem like this:



Unfortunately then developers tend to implement this kind of problem with an accumulation of special cases:

```
// without monoids
PaymentsFees(...)
PaymentsFeesWithOptions(...)
PaymentsFeesWithInsuranceAndOptions(.)
PaymentsFeesWithInsurance(...)
NoFeesButInsurance(...)
...
```

Whereas once you recognize that the cash flow sequences form a monoid, then you can just implement exactly the way you think about it:

```
// basic generators
monthlyFee(...) : Payments
options(...) : Payments
insurance(...) : Payments

// your custom code to combine
Payments invoice = monthlyFee
    .add(options)
    .add(insurance);
```

One major benefit is that the cognitive load is minimal. You just have to learn the type and its combine method, and that's it. And yet it gives you an infinite number of possibilities to combine them into exactly what you want.

Domain-Specific, within one Bounded Context

You may be tempted to reuse monoidal value objects across various parts of a larger system, but I would not advocate that. Even something as simple a Money class can be specific to some sub-domain.

For example in pretrade you would have a Money optimized for speed and expressed as an integer, as a multiple of the trading lot size, whereas for accounting you'd use a BigDecimal-based implementation that would ensure the expected accuracy even after summing many amounts and even after many foreign exchange conversions.

Another example this time with cashflows: in a tax-related domain, you can't just add an reimbursement cashflow to an interest cashflow, as they are treated very differently by the tax institution, whereas in an investment domain you would just add them all together without any constraint. For more on that point, I suggest Mathias Verraes blog post where he notes:

> *...dealing with money is too critical to be regarded as a Generic Subdomain. Different projects have different needs and expectations of how money will be handled. If money matters, you need to build a model that fits your specific problem space...*

Monoid, multiple times.

It's not uncommon for some domain concept to be a monoid more than once, for example once with addition and the neutral element ZERO, and a second time with multiplication and the neutral element ONE. As long as it makes enough sense from a domain perspective, then it's desirable to have more structures (being a monoid multiple times) or to be a stronger structure (see other mathematical structures later in this document).

Internal implementation hackery

Also note that neutral elements may call for some internal magical hacks for their implementation. You may rely on a magic value like -1 or Integer.MIN-VALUE, or on some special combination of magic values. You may think it's bad code, and it would be if it was meant to be seen or used regularly. However as long as it's well-tested (or built from the tests) and as long as it's totally invisible for the caller of the class, then it will only

cause harm when you are changing this class itself, which is probably acceptable. A monoid typically is not subject to a lot of changes, it's a finely tuned and highly consistent system that just works perfectly, thanks to its mathematical ground.

Established Formalisms, for Living Documentation

Monoids are one of the most frequent algebraic structures we can observe in the world of business domains. But other structures like groups (monoids with inverse elements), space vectors (addition with multiplication by a real number coefficient) and cyclic groups (think modulo) are also common. You can learn more about all these structures on Wikipedia and see whether they apply for your practical domain problems.

But the fact that these structures are totally described in the maths literature is important. It means that these solutions are established formalisms, which successfully passed the test of time. The DDD book actually advocates drawing on Established Formalisms for this reason.

Another reason is that you don't have to document them yourself. Just refer to the reference with a link and you're done. That's very much Living Documentation!

So if we want to document the fact that we implement a monoid, we could create a specific annotation @Monoid(String neutralElement), that could then be used to annotate a combine method on some class:

Alternatively since Java 8 you could define a class-level annotation

```
@Monoid(neutralElement="emptyList",
        operation="union")
```

Since a class can be a monoid several times, you would also need to mark the custom annotation as @Repeatable and define its container annotation Monoids, so that you can then annotate a class multiple times:

```
@Monoid(neutralElement="one",
        operation="multiply")
@Monoid(neutralElement="zero",
        operation="add")
```

Self-Explaining Values

Now suppose you want a complete audit on all the calculations, from the initial inputs to the result. Without monoids you'll have a bad time going through all the calculations to insert logs at each step, while making the code unreadable and with plenty of side-effects.

But if the calculations are done on a type like a monoid, with custom operations, then you could just enrich the operations with internal traceability audit trail:

```
public static class Ratio {

private final int numerator;
private final int denumerator;
private final String trace;

public Ratio multiply(Ratio other) {
  return new Ratio(
     numerator * other.numerator,
     denumerator * other.denumerator,
     "(" + asString() + ")*("
         + other.asString() + ")");
}

public String asString() {
 return numerator + "/" + denumerator;
}
```

With this built-in traceability, we can ask for the explanation of the calculation afterwards:

```
new Ratio(1, 3)
   .multiply(new Ratio(5, 2))
   .trace()
// trace: "(1/3)*(5/2)"
```

One question with the trace is to decide whether or not to use for the object equality. For example, is (5/6, "") really equal to (⅚, "(1/3)*(5/2)")? One way is to ignore the trace in the main object equals(), and to create another strictEquals() if necessary that uses it.

Encapsulated Error Handling

Many calculations can fail. We cannot divide a number by the number zero. We cannot subtract 7 from 5 in the set of natural integers. In software, error handling is an important source of accidental complexity, like in defensive programming, with checks statements bloating every other line of code.

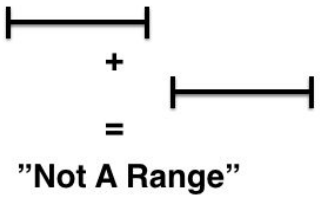
The traditional way to go to simplify is to throw exceptions, however it defeats the purpose or composability since it breaks the control flow. In practice I observe that throwing is acceptable for errors that are nothing but coding errors; once fixed they should never happen anymore, so for practical matters we have apparent composability.

An alternative to exceptions that can really happen at runtime is to make the monoidal operation a *total function*. A total function is a function that accepts any possible value for all its parameters, and therefore always returns a result for them. In practice the trick is to introduce a special value that represents the error case. For example in the case if division by zero, Java has introduced the special value NaN, for Not-a-Number.

Because a monoid has to follow the Closure of operation, it follows that the special extra value has to be part of the set of legal values for the monoid, not just as output but also as input. Usually the implementation of the operation would just bypass the actual operation and immediately return NaN when you get a NaN as a parameter: you propagate the failure, but in a composable fashion.

This idea was proposed by Ward Cunningham as the Whole Object pattern from his CHECKS patterns. Java Optional, and monads in functional programming languages, like the Maybe monad and its two values Some or None, are similar mechanisms to achieve this invisible propagation of failure. This is a property of an **Absorbing Element** like NaN: a + NaN = a

In the case of ranges with the union operation, you may want to introduce a special element NotARange to represent the error case of the union of disjoint ranges:



In the case of natural integers and subtraction, the way to make the function total is to extend the set with negative integers (which at the same time will promote the monoid into a group with inverses) The way to make the square root function (nothing to do with a monoid) a total function would be to extend the set from real numbers to the superset of complex numbers.

How to turn anything into a monoid

This idea of extending the initial set with additional special values is a common trick for monoids, and not just for error handling. It's also useful to artificially turn any set into one that is closed under a given operation.

Given a function with some input I and output O that are not of the same type, we can always turn it into a monoid by introducing the artificial type that is the tuple of both types: (I, O).

For example, given a function that gets a String and returns an integer, we can introduce the type Something (String, int), so that we now have a function that gets a Something and also returns a Something.

Testing Monoids

As mentioned already, monoids are easy to test as they're immutable and have no side-effect. Given the same inputs, the *combine* operation will always return the same result. And with just one single function, the testing surface of a Monoid is minimal.

Still monoids should be tested on all their important properties, like being associative, and on some random values, with an emphasis on the neutral elements, any artificial value like NaN or Special Cases, and when approaching the limits of the set (MAX_VALUE...).

Since monoids are all about properties ("expressions that hold true") like the following:

> Associativity
> FOR ANY 3 values X, Y and Z,
> THEN (X + Y) + Z == X + (Y + Z)
>
> Neutral element
> FOR ANY value X
> THEN X + NEUTRAL = X
> AND NEUTRAL + X = X
>
> Absorbing element
> FOR ANY value X
> THEN X + NaN = NaN

Property-based Testing is therefore a perfect fit for testing monoids, since PBT tools can directly express and test these properties. For example in Java we can use JunitQuickCheck to turn test cases into properties. Let us express the above properties for some custom Balance class with its neutral element and some absorbing element called Error:

```
public class Balance {

private final int balance;
private final boolean error;

public final static Balance ZERO
                    = new Balance(0);
public final static Balance ERROR
               = new Balance(0, true);

public Balance add(Balance other) {
    return error ? ERROR :
    other.error ? ERROR :
 new Balance(balance + other.balance);
}
```

The properties could be written:

```
@RunWith(JUnitQuickcheck.class)
public class MonoidTest {

@Property
public void neutralElement(
     @From(Ctor.class) Balance a) {
  assertEquals(a.add(ZERO), a);
  assertEquals(ZERO.add(a), a);
}

@Property
public void associativity(
       @From(Ctor.class) Balance a,
       @From(Ctor.class) Balance b,
       @From(Ctor.class) Balance c) {
  assertEquals(a.add(b).add(c),
               a.add(b.add(c)));
}

@Property
public void errorIsAbsorbingElement(
        @From(Ctor.class) Balance a) {
  assertEquals(a.add(ERROR), ERROR);
  assertEquals(ERROR.add(a), ERROR);
}
```

The PBT tool will run these test cases for a number (the default being 100) of random values.

Beyond monoids

When we model real-life domains into software, we most frequently recognize Monoids as the underlying mathematical structures that best matches the domain as we think about it.

But there are many other mathematical structures that are valuable to know. To be fair, you don't have to know their esoteric names to use them, you just have to focus on their respective distinguishing feature, or I should say "distinctive property".

Here are some common algebraic structures, each with their name (useful if you want to impress people) and most importantly the specific property that makes them special.

If you just have the closure of the operation, then it's called a "**magma**", don't ask me why. It's a much weaker structure than monoids. If you also have associativity, then it's called a "**semigroup**". If you add the neutral element, then it becomes a **Monoid**

indeed. And from that we can keep on adding specific properties. Note that all these structures are all about composition, plus something that helps composition even more.

If for any value there exists an *inverse value*, then the monoid becomes a "**group**":

value + inverse-value = neutral element.

It's a strong property to have inverses for all values. For example, natural integers don't have inverse with respect to addition, but signed integers do: 3 + (-3) == 0. In business domains, having inverses is less frequent, so groups are less frequently used than monoids. Groups are all about compensation, with inverse values that can always compensate the effect of any value.

Going further, if we can compose not just whole elements but also compose values *partially*, then we have a **vector space**. For example we would write that (1, 3) + 0.5(6, 8) = (4, 7). Notice the coefficient (the real number 0.5 here) that modulates the impact of the second term. Money with addition can be seen not just as a monoid, but as a group, and even as a space vector:

EUR25
+ **1.5**.EUR30
= EUR70.

Space vector is all about addition with multiplication by a scalar coefficient.

Any structure that happens to yield the same result regardless of the ordering of the values is called "**commutative**": a + b = b + a. This is a very strong property, not so frequent, so don't expect it too much in domain modeling, but if you see it or manage to make it so, go for it. Commutativity helps a lot especially for situations of "out of order" events, for example when distributed systems communicate through a network, it's frequent for events a, b, c that used to be in this ordering to arrive out of order, e.g. b, a, c. Being commutative is the most elegant way to deal with that. Exotic structures like CRDT rely on commutativity for that, but it's far beyond the scope of this text.

There's another structure that I like a lot. It's a very simple yet common one, and is called the "**Cyclic Group**", and its key idea is the modulo, hence the name cyclic. The days of the week form such a cyclic group of order 7 (its size), and the months of the year are another of order 12. Cyclic groups have a finite number of values, and when you reach the last value your cycle back to the first one: if the order of the cyclic group is 3, then the values are {0, 1, 2} and 2+1 = 0.

Numeration and time love cyclic groups, and as a result, domain-specific numeration and time also love them. For example, in finance, financial derivatives like options and futures are identified by their expiry date, simplified as a month code and a year code, e.g. H9 means March 2019, but also March 2029 or March 2009. The letter codifies the month, and the number codifies the year. They're both cyclic groups (one of order 12, and the other of order 10). And it turns out that the product of both is also a cyclic group of order 12*10 = 120, that's what we can learn by looking Wikipedia on Cyclic Groups. One benefit from using established formalism is that it comes with a lot of proven properties and theorems we can rely on safely. One of interest is that every cyclic group of order N is isomorphic (think equivalent) to the one on integers of order N, called Z/nZ. This means in practice that you can always implement it with integers as the internal state and modulo arithmetics.

Then there are many other more complicated algebraic structures available that deal with more than one operation and how the operations interact: a **ring** for example generalizes the arithmetic operations of addition and multiplication. It extends a commutative group (addition) with a second operation (multiplication), and requires that this second operation distributes with the first one:

a.(b+c) = a.b + a.c

Over the past 15 years I've created my own domain-specific values from all the above-mentioned structures, and many times without knowing the corresponding name. Still, it helps to pay attention to the properties that we can build upon or not, for a recap, with a given operation noted "+":

- **closure of operation** T + T is a T
- **associativity**: a + (b + c) = (a + b) + c
- **neutral element** e such as a + e = a
- **inverse** (-a) for any a such as a + (-a) = 0 (your own zero)
- **using a coefficient**: a + alpha.b

- **commutativity** : a + b = b + a
- **cycle** of order N: a + N = a

Inheriting the algebraic properties from the implementation structures

We usually implement domain-specific concepts from the standard built-in types of the programming language: boolean, integers and other numbers, finite sets of enums, Strings, and all kinds of lists. It happens that all these types exhibit many properties: numbers are rings, groups, space vectors, enums can be seen as Cyclic Groups, boolean are groups, lists and maps can easily be seen as monoids. And it happens that putting several structures next to each other as fields (product types) usually preserves the relation if the operation on the whole is defined as the field-wise operation of each of the components. This explains why so many domain concepts *inherit* part of their internal implementation structure, unless you mess with their operation. Think about it when you implement.

Make your own arithmetic to encapsulate additional concerns

Creating your own arithmetic helps keep your code simple even when you need to perform calculation of a value “with something else”, like keeping track of the accuracy of the calculation, or of its uncertainty, or anything else. The idea is to expand the value into a tuple with the other thing you also are about:

(Value, Accuracy)
(Value, Uncertainty)

And to expand the operation into the tuple-level operation, trying to preserve some desirable properties along the way.

For example for the type TrustedNumber(Value, Uncertainty) you could define the addition operation this way, in a pessimist fashion such as the resulting uncertainty is the worst of both operands:

```
public add(TrustedNumber o){
  return new TrustedNumber(
      value + o.value,
      max(uncertainty, uncertainty));
}
```

This approach is standard in mathematics, for example for a complex numbers, or dual numbers.

Creating your own arithmetic is more natural and more good-looking with operator overloading, which does not exist in Java.

For more examples on how drawing on established formalisms and algebraic structures, don’t hesitate to dig into JScience; I did a decade ago and I learnt a lot from it. It’s built on a *linear algebra* layer of supertypes, from which everything else is built upon.

Case Study: Environmental Impact Across a Supply Chain

Just like other code snippet across this article, the code for this case study is online.

Putting together all what we’ve seen so far, we will study the case of a social network to track the environmental impact of companies and their suppliers.

Let’s consider a pizza restaurant willing to track its environmental impact across its complete supply chain. Its supply chain can be huge, with many direct suppliers, each of them having in turn many suppliers, and so forth. The idea is that each company in the supply chain will get invited to provide its own metrics, at its own level, along with the names of its direct suppliers. This happens massively in parallel, all around the world, across potentially hundreds of companies. And it also happens incrementally, with each supplier deciding to share their impact when they become able or willing to. Still, at any time, we would like to compute the most up-to-date aggregated impact for the pizza restaurant at the top.

The impacts we are interested in include the **number of suppliers** involved for one pizza, the **total energy consumption** and **carbon emission** by pizza produced, along with the respective **margins of error** for these numbers, and also the **proportion of certified numbers** (weighted by their respective mass in the final product) over the whole chain.

We could collect all the basic facts, and then regularly run queries to calculate the aggregated metrics over the whole dataset each time, a brutal approach that would require lots of CPU and I/O. Or we could try to start from what we already had and then extending it with the latest contributions in order to update the result. This later approach can save a lot of processing (by reusing past calculations, in addition to enabling a map-reduce-ish approach), but requires each impact to compose smoothly with any other.

We decide to go the later route. We want to compose, or ”chain" the impacts together all across the chain, to compute the full impact for one pizza in our restaurant at the root of the supply chain.

From what we've seen, we need to define a concept of Environmental Impact that:

- can represent the metrics available for one supplier in isolation
- can represent the metrics that matter for the pizza restaurant in terms of impact at the top of the supply chain
- can compose all supplier's metrics, and their supplier's, into the aggregated metrics for the pizza restaurant.

Out of the impacts we want, the number of suppliers is easy to compose: for each supplier (level N), its supplier count is exactly 1 plus the supplier counts of all its direct suppliers (level N-1). It's naturally additive, in the simplest possible way. The energy consumption and carbon emissions are naturally additive too. This suggests the following concept in code:

```
public static class
EnvironmentalImpact {
  private final int supplierCount;
  private final
            Amount energyConsumption;
  private final Amount carbonEmission;

  // … equals, hashcode, toString
}
```

Now in order to compose partial impacts in a way that is weighted by their respective contribution to the pizza, we make this value a space vector, with the “addition” and "multiplication by a scalar” operations:

```
public EnvironmentalImpact add
       (EnvironmentalImpact other) {
  return new EnvironmentalImpact(
    supplierCount
        +other.supplierCount,
    energyConsumption
        .add(other.energyConsumption),
    carbonEmission
        .add(other.carbonEmission));
}

public EnvironmentalImpact times
              (double coefficient) {
   return new EnvironmentalImpact(
       supplierCount,
    energyConsumption
         .times(coefficient),
    carbonEmission
        .times(coefficient));
}
```

Because all these amounts are not that easy to measure, they come with significant margins of error, which we'd like to track when it comes to the end result. This is specially important when suppliers don't provide their impact, so we have to guess it, with some larger margin of error. This could make the calculations quite complicated, but we know how to do that in a simple way, using another tuple that gathers the amount, its unit and its margin of error:

```
public static class Amount {
  private final double value;
  private final String unit;
  private final double errorMargin;
  // … equals, hashcode, toString
}
```

And because we want to add these amounts weighted by coefficients, we want to make it a space vector as well, with the addition and multiplication by a scalar:

```
public Amount add(Amount other) {
  if (!unit.equals(other.unit))
      throw new
IllegalArgumentException(
      "Amounts must have same units: "
        + unit + " <> " + other.unit);
   return new Amount(
     value + other.value,
     unit,
     errorMargin + other.errorMargin);
}

 public Amount times(
             double coefficient) {
    return new Amount(
      coefficient * value,
      unit,
      coefficient * errorMargin);
}
```

We're lucky the error margins are additive too. But it's also possible to calculate them for any other operation than just addition if we wanted to.

Now we're almost done, but remember we wanted to track the proportion of certified numbers in the whole chain. A proportion is typically expressed in percentage, and it's a ratio. If we compose one impact that is 100% certified with two other that are not at all, then we should end up with a proportion of certification of 1/3, i.e. 33%. But we want this proportion to be weighted by the respective mass of each supplier in the final product. We notice that this kind of weighted ratio is not additive at all, so we need to use the trick of making it into a tuple: (total certification percents, total of the weights in kg), which we can compose with addition and multiplication by a scalar.

So we now decorate the Amount class with a CertifiedAmount class that expands it with this tuple:

```
/** An amount that keeps track of its
percentage of certification */
public static class CertifiedAmount {
  private final Amount amount;
  // the total certification score
  private final double score;
  // the total weight of the certified
thing
  private final double weight;
```

And we update our EnvironmentalImpact class to use the CertifiedAmount instead of the Amount, which is easy since it has the exact same methods names and signatures.

Now let's use that for 1 pizza, that is made of 1 dough, 0.3 (kg) of tomato sauce and some cooking in the restaurant.

```
EnvironmentalImpact cooking =
singleSupplier(
  certified(1, "kWh", 0.3), // energy
  certified(1, "T", 0.25)); // carbon
EnvironmentalImpact dough =
singleSupplier(
  uncertified(5, "kWh", 5.),
  uncertified(0.5, "T", 1.));
EnvironmentalImpact tomatoSauce =
singleSupplier(
  uncertified(3, "kWh", 1.),
  certified(0.2, "T", 0.1));
```

Which displayed into the console:

```
EnvironmentalImpact(1 supplier,
  energy: 1.0+/-0.3 kWh (100% certified),
  carbon: 1.0+/-0.25 T (100% certified))
EnvironmentalImpact(1 supplier,
  energy: 5.0+/-5.0 kWh (0% certified),
  carbon: 0.5+/-1.0 T (0% certified))
EnvironmentalImpact(1 supplier,
  energy: 3.0+/-1.0 kWh (0% certified),
  carbon: 0.2+/-0.1 T (100% certified))
```

From that we can calculate the full impact of the restaurant by chaining each impact:

```
EnvironmentalImpact pizza = cooking
  .add(dough)
  .add(tomatoSauce.times(0.3));
```

If we print the resulting impact into the console, we get:

```
EnvironmentalImpact(3 suppliers,
  energy: 6.9+/-5.6 kWh (43% certified),
  carbon: 1.56+/-1.28 T (56% certified))
```

Which is what we wanted. We can then extend that approach for many other dimensions of environmental impact accounting, more details on accuracy, estimated vs measured vs calculated values, traceability of the numbers etc., just by expanding the concepts at each level, while still keeping it all as nested mathematical structures that compose perfectly. This approach scales for high complexity, and for high cardinality as well.

Domain-Driven Design loves monoids

Domain-Driven Design leans towards a functional programming style in various aspects. The most visible is the obvious Value Object tactical pattern, but in the Blue Book you can also find the patterns Side-Effect-Free Functions, Closure of Operations, Declarative Design and Drawing on Established Formalisms.

It turns out that if you put all of them together, you end up with something like monoids.

Monoids are everywhere, even in Machine Learning, with the ubiquitous matrices and tensors, and with the key trick of composing derivatives together thanks to the Chain Rule.

Once you've used monoids a few times you can't but fall in love with them. As a consequence, you try to make everything into a monoid. For example with my friend Jeremie Chassaing we've discussed monoids and Event Sourcing, and he kept investigating how to make Monoidal Event Sourcing (see his related blog post).

The code for the code snippets in this text are all online as Github gists: https://gist.github.com/cyriux

Many thanks to my colleague Mathieu Eveillard for reviewing an early draft, and to Eric Evans, Mathias Verraes, Yvan Phelizot for the constructive reviews and feedbacks.