meringue Documentation

Vadim Samokhin

Dec 02, 2020

Contents

I published this entry both in my old and new blogs, and I repost it here in its entirety.

CHAPTER 1

Short intro in metaphysics

From ancient times-I mean, really ancient, all the way before Plato, Aristotle and all, people have been entertaining themselves with a fancy question: what does the world consist of? What is the reality surrounding us? What is there? What is it like? Why?

1.1 Objects

There are apples and oranges, and there is the apple in front of me. There are Tuesdays and sevens, and today is Tuesday and there are seven avocados in my refrigerator. There is honesty and there are specific persons who are honest.

1.2 Properties

It seems that there are concrete objects, like that apple in front of me, and there are ways those objects are, like being red, being sweet, and laying on the table. Usually, properties take a predicate position in a sentence.

If I had two apples in front of me, and both of them were red, what makes them both red? There must be something that explains the seemingly simple fact that many objects share the same property. What is this, that is shared, that ground the redness of many apples? Is it an object? Is it tangible?

There are at least two major views on that. The first one says that, yes, there is a special "thing" called a Universal, and all the concrete objects have their properties in virtue of sharing specific Universals. Universal Red is like a single source of truth for what it means to be red. Besides, this entity grounds the similarity between a red apple and a red firetruck.

The second view is that two rednesses of two apples are distinct tangible entities called tropes, which represent the property of being red. What makes those entities similar? It's either the fact of sharing a universal Red or a mysterious entity representing this similarity relation-trope again.

Both views have their supporters, arguing furiously for a good couple of thousand years. Both sides have good arguments for their supported view and against opponents' view. But this debate seems to be far from being resolved; I doubt it can be resolved ever. This problem even has a name: Problem of Universals.

1.3 Natural Kinds

Besides, there is a specific kind which seems to be natural, the one where all the apples belong: the kind called "Apple". What makes it "natural"? Its ubiquity, universality, and, uhm, naturalness. Do all the green things form a natural kind of "Green thing"? Some say yes, some say otherwise. Again, both sides have good arguments.

1.4 Abstract Objects

Finally, there are intangible objects called "abstract", like numbers or feelings. They are definitely out there, and they seem to be close to both properties and natural kinds. For example, linguistics says that there are several ways to nominalize a predicate. For example, there is a person who is brave, and there is bravery. If predicates are to subjects what properties are to objects, some properties can turn into abstract objects. It works the other way round as well: some abstract objects can be "unnominalized". But we can't do that with Thursdays, and there is no straightforward accepted way to do that with numbers.

Up to this point, things might seem quite complicated. But the world's ontology is finite, and I've already outlined its main constituents. For most contemporary philosophers, the ontology is even more parsimonious.

CHAPTER 2

Domain modeling with metaphysics

2.1 Correspondence between domain model entities and code entities

I want my code to reflect the model of the desired domain expert's reality, which is currently only in her head, and which she expresses verbally. I want to have a one-to-one correspondence (bijection) between a model and a codebase. That is, every entity from a model should correspond to an entity from a codebase.

Why? Because if a model changes, I know exactly where to fix it. If some entity is added, I know exactly where to add it in my code.

It's only natural, and it's used in a way more mature area such as engineering. Simplified, a process of manufacturing pretty much anything looks like the following. An engineer sits and proceeds to drawing. When he's done, he gives that drawing to a factory worker who starts the manufacturing thing. If an engineer modifies some tiny part of an entire drawing, a factory worker knows exactly which manufactured part should be redone.

Wouldn't it be wonderful to have this state of affairs in software engineering?

2.2 What metaphysics has to offer

Since I want my codebase to reflect a domain model, it seems reasonable to enjoy the same ontology that constitutes reality. Pretty damn smart folks have been engaging in figuring out what reality consists of for, like, two and half thousand years. Why not take advantage of their results?

So far, there are objects, both concrete and abstract; natural kinds they fall under; and properties they possess.

Natural kinds stand out in certain respects. First, it seems plausible that there are much fewer kinds than objects. Second, objects come and go, but natural kinds stay the same. They are immutable pieces of knowledge. That's why they are extremely stable. And that's the reason why they are universally ubiquitous. This, in turn, takes a very small cognitive load to grasp them and operate them. And that's the primary reason why you should concentrate your efforts on discovering natural kinds.

2.3 Abstracting towards natural kinds

Having a few natural kinds and plenty of objects falling under them is often a benchmark of a domain model well thought.

In my personal software development experience, I often find entities that I earlier thought of as single concrete objects not belonging to any kind, but which turn out to be kinds later. It's very likely that those entities are already kinds in domain expert's head; the sooner we identify them, the better for us.

The crucial part of honing a model is asking what-kind of questions. You abstract away from details, focusing on essential properties: what is this anyway? That is, what kind is it? Different things fall into the same category. Red apple, red firetruck, and blood fall under the category of red things. Does it make sense in your domain? If yes, great, you arrived with a natural kind. If no, go on discovering. Small green sour apple and big sweet red apple, what are they? They are apples! Does a concept of apple seem plausible for a domain expert? If it doesn't, this mind entity doesn't reflect a domain expert's way of thinking; it's not a viable option to proceed to code at this stage.

As a quick and cheap way of checking its vitality is to make sure that a concept is not a made-up word that suits some particular implementation, it's present in a language. After all, language is a reflection of reality; it reflects mostly what exists (notable counterexample are dragons), at least as ubiquitous mental concepts. If you use a word no one ever heard before, chances are you have a domain model entities-code entities mismatch, akin to object-relational impedance mismatch in some sense.

Useful abstractions resulting from this train of thought are natural kinds inherent to a domain.

2.4 Objects

Useful abstractions are rarely discovered in isolation. More often than not, they follow inductive reasoning. You encounter one object, note its properties; then you stumble upon another one, find out that there are some properties in common; then you see the third instance and find out it has mutual properties with the first two. Okay, probably all three are of the same kind; it might turn out otherwise later though. Little by little, essential properties of some concept are fleshed out. Thus, a basic object set and a concept they exemplify, both emerge at the same time. In general, it looks like a following endless cycle: pose a hypothesis, encounter new evidence, modify initial hypothesis if needed; then repeat, on and on. This idea is universal. That's how TDD works; that's how a product is developed in a lean startup way; that's what growth hacking marketing is mostly about. That's the process known as a scientific method, and that's how science has been working for about three hundred years.

2.5 Properties

There are objects, and there are ways those objects are. What are they like? Where are they? When? What state are they in? What are they doing? How do they feel? How many of them are there? What relations do they hold with other objects? There are a lot of possible properties, but there can not be properties soaring in the air. They are strongly dependent on objects.

A set of essential properties forms a natural kind. If you have some objects with identified properties, but struggle to come up with a consistent natural kind, keep looking. A reward brings clarity to both a domain model and a codebase, and satisfaction to your soul. There is no recipe here actually, it's more of an art rather than craft.

CHAPTER 3

What my code looks like

I use interfaces or abstract classes for natural kinds and classes implementing or extending them for any type of object. If I need to create very similar objects which are different in some respects, it might be reasonable to do so with a single class. I parameterize it with properties that may vary from instance to instance. If I have an Apple kind, probably I have a `Jonagold` class. I'd likely want to have them of different colors, so I pass `Color` in a `Jonagold`'s constructor.

As another example, an `ISO8601DateTime` abstract class from my meringue datetime library. It represents a datetime in ISO8601 standard. A handful of concrete datetime objects belong to this kind. What do you mean when you say "yesterday", anyway? You mean a specific date(time). The same with "tomorrow". The same with "the first day of several months later". And the same with the "maximum of seven given dates". And so on. That's why I have an `ISO8601DateTime` abstract class. Among others, I have `ISO8601Interval` and `Schedule` concepts. Check out the code if you fancy learning some more.

The transition of domain model entities into code is actually a pretty straightforward enterprise. Coming up with a consistent model, not so much.

CHAPTER 4

What's next

I find metaphysics to fit enterprise software development discovery and implementation needs perfectly. I've been reading some stuff for quite a while now, and recently I've started to blog about it. At the same time, I gradually start to opensource some of the libraries that I've created with this approach.

I decided to make a new digital home for everything I write. Meet my new blog! In the coming months, I'm planning to dive a bit deeper into the main metaphysical areas. After that, I'll write some more about how it can be applied to software development.

CHAPTER 5

Quick start

If you haven't yet read *meringue philosophy* entry, now is a good time to do it.

5.1 Short recap

There is a small set of basic abstractions, and quite a few of their implementations. It's like Lego bricks. There are plenty of them, but few of their forms. An amount of possible combinations explodes combinatorially. You can create as complex objects as you need, but you don't lose grip of your code entities – they stay small and simple.

Any class represents a specific implementation of a kind denoted by an interface (*meringue philosophy*, again). So any class' declaration, like `class SomeSpecificImplementation implements Concept` is read like "*Some specific implementation* is a *concept*". In other words, a class *falls under* a category represented by an interface. What is this specific red and sweet thing right in front of me? It's an apple. What is that green and sour thing on the table? That's an apple either. Both are apples, but the former is red and sweet, and the latter and green and sour. They have the same essential properties, ones that make them both apples, but their accidental properties vary. So, you just need to find abstractions that work for your domain.

Just focus on **what** you need as a result, instead of **how** you want to get there.

5.2 Creating datetimes

So, what do you want? Say, you want to output a current datetime. So, current datetime is what you need. There is a class that represents current datetime: it's called `Now`.

Or, say you want to add a few seconds, minutes, hours, days, or months, or years. In fact, what you need is some datetime in the future. It's some interval away from your given datetime. It doesn't matter how to get there, the only thing that matters is what you need, not how – remember? So that's how it looks in meringue:

```
$f = new Future(new Now(), new NHours(2));
```

The code above creates an object which *is* a datetime, namely a datetime 2 hours later from now.

Or, you might want to create an arbitrary datetime. If you have an ISO8601 string, you can do it with `new FromISO8601($yourString)`. The name of this class is read like *an object is created from an ISO8601 string*.

In the same vein, you can create a past some time earlier. This could look like

```
$f =
   new Past(
      new FromISO8601('2020-04-05 12:26:04+03'),
      new NHours(2)
   );
```

It's possible to create any datetime in the past or future with any custom intervals as well.

5.3 Creating intervals

Meringue intervals go in two flavors. The first one is a floating interval not tied to any specific start date, and the second one is with fixed start datetime. Hence the names: `Floating` and `WithFixedStartDateTime`.

Floating interval should be used when you don't care when it starts. How long does it take to cook a cake? It takes two hours. No matter when you start, it's just two hours. There are classes representing intervals used most often: one second, *n* seconds, one minute, *n* minutes, and so on. Besides, you can use ISO8601 format for intervals. It starts with *P*, then amount of years followed with *Y*, amount of months followed with *M*, then date, indicated with *M*. After that, *T* separator marking the beginning of time notation. *H* is for hours, *M* for minutes, *S* for seconds. So a floating interval of one year, three days and seventy four seconds looks like that:

```
$f =
   new Future(
      new FromISO8601('2020-04-05 12:26:04+03'),
      new FromISO8601('P1YT3D74S')
   );
```

When a start date matters, the corresponding class is used. For example, how many days are in one month? It depends on the month, doesn't it? So we can't answer this question having a floating interval. There are two way we can get it: from a start and finish datetime range, and from a start date and an interval. Hence two classes: `FromRange` and `FromStartDateTimeAndInterval`.

5.4 Formatting datetimes

First, I was pondering about writing some custom code for handling that. But I ended up with using a default IntlDateFormatter class in my projects. It represents an ICU initiative, combining internationalization and localization efforts. Most of the time, no extra layer of complexity is needed.

If you need some simple formatting or ISO8601 support, there are a few classes supporting that.

5.5 Formatting intervals

Due to reasons outlined above, only intervals with fixed start date can be formatted.

Mainly, you need to express an interval in some units: seconds, minutes, etc. They can be rounded either up or down. There is a bunch of classes doing exactly that: `TotalCeiledDays` (rounded up, after a `ceil` php function), `TotalFullMinutes` (rounded down), etc.

If you need a human readable version, there is a `HumanReadable` class. I doubt it suits your needs though; more often than not, all the formatting facilities are unique to an application, and trying to stick it into a single library only makes things more complicated.

5.6 Schedule

Sometimes you need to represent a concept of a schedule, be it a grocery store, a restaurant, or your morning jogging. More often than not, it depends on the week day. Here is how it looks in meringue:

```
$schedule =
   new ByWeekDays(
      new Daily(
         new DefaultTime(6, 0, 0),
         new DefaultTime(20, 30, 0)
      ),
      new Daily(
         new DefaultTime(6, 0, 0),
         new DefaultTime(20, 30, 0)
      ),
      new Daily(
         new DefaultTime(6, 0, 0),
         new DefaultTime(20, 30, 0)
      ),
      new Daily(
         new DefaultTime(6, 0, 0),
         new DefaultTime(20, 30, 0)
      ),
      new Daily(
         new DefaultTime(6, 0, 0),
         new DefaultTime(2, 0, 0)
      ),
      new Daily(
         new DefaultTime(6, 0, 0),
         new DefaultTime(23, 00, 0)
      ),
      new Daily(
         new DefaultTime(6, 0, 0),
         new DefaultTime(23, 0, 0)
      )
   );
```

If your restaurant or anything is open twenty four seven, there is a class for that – `TwentyFourSeven`. If you want to take a state holidays into consideration, you can use `Monthly`. It takes two parameters: the first one is a working schedule, the second one is the schedule of holidays.

CHAPTER 6

How to convert a datetime to specific timezone

6.1 What is a timezone?

It's 17:05 right now, as I'm writing this sentence, here in Moscow. At the very same moment, it's 15:05 in London: it's located to the west from Moscow. There is an official explanation for that: we're in different timezones. Timezone is a specific region in the globe that has the same time. Most of them have a single offset relative to a certain timezone with zero offset called UTC.

Here is a full list of all timezones with their offsets.

6.2 What is a daylight saving time?

When I was studying at school, some peculiar events of specific sort usually happened twice a year. The first happened in the end of March, when Moscow switched to summer time. With the intent to make daylight last longer in the evening, we turned clocks one hour forward. So what was 9 pm in the middle of March, became 10 pm couple of weeks later; thus, it was more light at 9 pm. That period is called daylight saving time.

In autumn it was quite the opposite. The intent was to make mornings lighter, so we turned clocks one hour backwards. If you look out of a window in the middle of October in Moscow at about 6 am, it will be dark. At 7am it's more or less fine. So why not turning 6 am to 5 am? That's exactly what we did for years, at the last Sunday of October. So 6 am becomes what recently was 7 am.

6.3 Timezones in PHP

All good and fine, but it's a source of confusion for most of developers. In Russia, we quit using daylight saving time back in 2014. But in some parts of the World, it's still there.

Luckily, php has our back. For example, if you live in London, you don't have to calculate when you switch to DST every year. You can use Europe/London timezone instead, and daylight saving time is taken care of automatically. That information is distributed within PHP. Here is an entire list. Find a city from this list that has the same timezone

and use it explicitly in your project. That's just more convenient way to work with timezones, instead of specifying manually the offset.

If your government decides to change your timezone rules, the PHP may not have time to catch up. In this case, you can upgrade timezone database via PECL.

6.4 How to deal with timezones in php

Here is how you can adjust your datetime according to specific timezone:

```
(new AdjustedAccordingToTimeZone(
    new FromISO8601('2018-04-25 15:08:01+03:00'),
    new CET()
))
    ->value();
```

You can find some more about an overall approach used in meringue in a *quick start entry*.

CHAPTER 7

How to get the beginning of a day

Just like with any other development task, first I try to identify *what* I need. As the title goes, "how to get *the beginning of a day*". Thus, I need the beginning of any given day. As a result, there is a class which is called exactly like that: `TheBeginningOfADay`. In turn, what is it? It's a specific datetime actually. Hence no wonder that this class extends `ISO8601DateTime` abstract class.

So, obtaining the beginning of any given day from a datetime goes like the following:

```
(new TheBeginningOfADay(
    new FromISO8601('2014-11-21 08:12:54-11:30')
))
    ->value();
```

If you need the beginning of *a current datetime*, it goes like that:

```
(new TheBeginningOfADay(
    new Now()
))
    ->value();
```

If you want to get just a date from a datetime, things change. What you need is a *date*, not a datetime. Thus, the class you should use definitely implements a `Date` interface (or a corresponding abstract class). The class we need in our case is a `FromISO8601DateTime`. It's read as `A date obtained from an ISO8601 datetime`, and an abstract class implies that there is a single property that an object can tell you about: it is its textual representation, or `value`, which is just more concise.

Here goes the code:

```
(new FromISO8601DateTime(
    new FromISO8601('2017-07-03T00:00:00+00:00')
))
    ->value(); // results into '2017-07-03' string
```

If you want to format it some peculiar way, you have two options: either built-in php ISO8601 formatting facility, or, if you want to localize your datetime, you'd want to opt into the IntlDateFormatter class.

CHAPTER 8

How to create a datetime from a custom format

With built-in php formatting facilities, you can create a datetime from however crazy sign sequence.

First, consider pretty standard ISO8601 string: `2018-12-31T23:12:59+0200`. It has an `Y-m-d\TH:i:sO` ISO8601 format. So, the following code results into almost identical datetime string (mind last colon sign in timezone offset):

```
(new FromCustomFormat(
    'Y-m-d\TH:i:sO',
    '2018-12-31T23:12:59+0200'
))
    ->value(); // returns a '2018-12-31T23:12:59+02:00' string
```

Now consider a more esoteric sequence, say, `122018--31TT!23:12:59+0200`. If you take a close look, it has a `mY--d\T\T\!H:i:sO` format. Indeed, the following code returns `2018-12-31T23:12:59+02:00`:

```
(new FromCustomFormat(
    'mY--d\T\T\!H:i:sO',
    '122018--31TT!23:12:59+0200'
))
    ->value();
```

After you've obtained an `ISO8601DateTime` object, you can do lots of stuff: *add seconds, minutes, hours, days, months, years* to it, *calculate a difference between datetimes*, *convert it into any other timezone*, and much more. Consider *quick start entry* for more info.

CHAPTER 9

How do you get a current datetime

Following *meringue philosophy*, it's a piece of cake. Just ask the following question: **what** do I need? You need a current datetime, that is, now. Here goes the object – `$currentDatetime = new Now();`.

You can work with this datetime further: you can get datetimes *in the future* or *in the past*, or *format it to your taste*.

Often times, you want to know what time it is in some other timezone. For example, it's 15:08 in Moscow, and you want to know what time it is in Central Europe. In other words, you want to know specific point in time adjusted according to some other timezone, in our case it could be Berlin, Paris or Rome. Hence the class: `AdjustedAccordingToTimeZone`. So here is how you can convert a datetime to another timezone:

```
(new AdjustedAccordingToTimeZone(
    new FromISO8601('2018-04-25 15:08:01+03:00'),
    new CET()
))
    ->value()
```

Currently there are a few of built-in meringue timezones, feel free to add yours.

CHAPTER 10

How to convert a Unix timestamp to DateTime

Since I always try to find *what* I need instead of *how* to get there, I don't use the lingo from the title, like "how to *do* anything". In case of the current post, I create a datetime from Unix timestamp, or, in other words, a datetime *converted* from Unix timestamp.

10.1 How to create a datetime object from Unix timestamp

First off, Unix timestamp is a number of seconds since January, 1st, 1970, UTC. If you've skimmed through that date, mind that it contains a timezone, and it is UTC. Thus, there is no such thing as Unix timestamp in CET or Los-Angeles timezone. There is always the single Unix timestamp. It's like the single moment in absolute timescale, but it just happens so that different countries have different local time.

To bring my point home, consider a timestamp which equals to `1504534440`. If you wonder what time it is in UTC, here you go:

```
(new FromTimestamp(1504534440))->value(); // returns 2017-09-04T14:14:00+00:00
```

At the same moment in time, most people in Kalinigrad already have had a dinner:

```
(new AdjustedAccordingToTimeZone(
    new FromTimestamp(1504534440),
    new Kaliningrad()
))
    ->value(); // it's 2017-09-04T16:14:00+02:00
```

And still at the same moment in time, it's an early morning in Honolulu:

```
(new AdjustedAccordingToTimeZone(
    new FromTimestamp(1504534440),
    new HawaiiWithNoDST()
))
    ->value(); // 2017-09-04T04:14:00-10:00
```

After you've got an `ISO8601DateTime` object, you can do some more: *subtract seconds, minutes, hours, days, months, years* from it, *calculate a difference between datetimes*, *format it anyway you like*, and much more. Consider *quick start entry* for more info.

10.2 And the other way round

What if you need to *convert a datetime object into Unix timestamp*? Since *what* you need is a number of seconds since January, 1st, 1970 UTC, there is a special class for that: `SecondsSinceJanuary1st1970`. Here is how it's invoked:

```
(new SecondsSinceJanuary1st1970(
    new FromISO8601('2014-11-21T06:04:31+00:00')
))
    ->value(); // returns 1416549871
```

If you wonder why I have so many classes, check out *my philosophy*.

CHAPTER 11

How to compare two dates

You can use built-in `ISO8601DateTime` methods for that: `equalsTo()`, `laterThan()`, and `earlierThan()`.

But, as usual, ask yourself a question what you really need. Chances are you don't need to compare; you need a maximum or minimum date instead. If so, I've got you covered: `Min` and `Max` are what you're looking for:

```
$m =
   new Max(
      new Now(),
      new Future(
         new FromISO8601('1986-05-04 00:30:00+03'),
         new NYears(34)
      )
   );
```

After you got what you need, you can proceed to textual representation. `value()` method returns an ISO8601 string value. There are *fancier ways* to format datetime either.

CHAPTER 12

How to obtain the first day of a week

Start of a week *is a* specific datetime. Looking at this sentence, I can draw two points. The first one is that I need a class called `StartOfTheWeek` which denotes a subject in a proposition `Start of a week is a specific datetime`. The predicate, *is a specific datetime*, clearly tells me that this class should either implement some sort of `DateTime` interface or extend the same sort of abstract class. And you know what? There is one already, and it's called an `ISO8601DateTime`.

Alright, *no more philosophy*, let's just get right into it. Which weekday is the first day of the week? Since this datetime is in ISO8601 format, it dictates us that a week starts on Monday. It might differ depending on cultural traditions of course, but this standard is culture-agnostic, so it's always Monday.

Here is how you can actually get the start of a week in code:

```
(new StartOfTheWeek(
    new FromISO8601('2020-04-23T01:28:04+07:00')
))
    ->value(); // returns 2020-04-20T00:00:00+07:00, which is Monday indeed
```

If you need to obtain the first day of the *current* week, just pass the *current datetime*:

```
(new StartOfTheWeek(
    new Now()
))
    ->value();
```

And if you want just a date of the beginning of the week, instead of a datetime, you can do the following:

```
(new FromISO8601DateTime(
    new StartOfTheWeek(
        new FromISO8601('2020-04-23T01:28:04+07:00')
    )
))
    ->value(); // returns '2017-04-20' string
```

Finally, in case you want to find the last day of the week, you can do simple math. Just get the first day, and then *add six days to it*:

```
(new Future(
    new StartOfTheWeek(
        new FromISO8601('2020-04-23T01:28:04+07:00')
    ),
    new NDays(6)
))
    ->value() // returns 2020-04-26T00:00:00+07:00
```

If you find that having a distinct class for getting the last day of a week would be more convenient, you can write it and create a pull request. If tests are in place and code is OK, I'll merge it pretty soon.

CHAPTER 13

How to convert a string to date

So, what you really need is an ISO8601 date. In other words, you need an ISO8601 object. Thus you need a class which creates this object from string. That is a meringue way of thinking, reinforced by *metaphysics*.

So if you have an ISO8601-compliant string, it's as simple as that:

```
$m = new FromISO8601('1986-05-04 00:30:00+03');
```

If you have a string in some arbitrary format, you can use `FromCustomFormat` class. For example,

```
$customFormatDateTime = new FromCustomFormat('mY--d\T\T\!H:i:sO', '122018--31TT!
↪23:12:59+0200');

$this->assertTrue($customFormatDateTime->isValid());
$this->assertEquals(
   '2018-12-31T23:12:59+02:00',
   $customFormatDateTime->value()
);
```

As usual, checking out tests is always a good idea.

CHAPTER 14

How to calculate a difference between two dates

The question in the header implies that the question is formulated correctly. Instead of subtracting two dates, you want a difference. Instead of focusing on how, you focus in what. Instead of posing a a problem in terms of implied solution, you pose a problem in terms of what you really need. And that is a good thing indeed. It even has a name: declarative programming. Sql is known for that, for example. Another my library for declarative validation takes exactly this approach.

Declarative code applied to oop takes roots in *metaphysics*. You formulate **what**'s out there in your domain. You're not interested in how it came to be, or how it will evolve. It doesn't really matter. There are only objects, their properties and relations with each other and natural kinds they belong to.

So, the difference between two datetimes is an interval. You can get it from a range of datetimes. Those sentences transfer directly to php code:

```
class FromRange extends WithFixedStartDateTime
//
```

What is `WithFixedStartDateTime`? Let's take a look at its declaration:

```
abstract class WithFixedStartDateTime implements ISO8601Interval
```

So, `WithFixedStartDateTime` **is a** ISO8601 interval with the following property that I wanted to highlight: it has fixed start datetime.

So, that's how the difference between two datetimes in php looks like:

```
(new FromRange(
    new FromISO8601('2017-07-03T14:27:39+00:00'),
    new FromISO8601('2017-08-28T14:29:47+00:00')
))
    ->value(); // returns P0Y1M25DT0H2M8S
```

Chances are you'd like to convert it to some human-readable format. If so, I got you covered. Here is how this can be done:

```
(new HumanReadable(
    new FromRange(
        new FromISO8601('2017-07-03T14:27:39+00:00'),
        new FromISO8601('2017-07-05T14:27:38+00:00')
    )
))
    ->value(); // returns 1 day, 23 hours, 59 minutes and 59 seconds
```

Or, you can use that difference for getting some datetime *in the past* or *in the future*.

CHAPTER 15

How to add seconds, minutes, hours, days and all to php datetime

As usual, the key is not **how** to do something, but **what** do you want to have. You don't need to add anything, you want some datetime in the future instead. This approach is known as declarative programming. But this declarative nature is just a consequence of a *deeper-rooted philosophy behind meringue library*: focus on abstractions. To do that, keep an eye on linguistic concepts the problem is formulated with. Look for naturally occurring categories: universally accepted and ubiquitous concepts, called natural kinds.

So, if you want some datetime in the future, that is, a future datetime, it's easy to guess the class name: `Future`. If you look at the declaration, you'll see `class Future extends ISO8601DateTime`. It's translated to English as "future **is a** ISO8601 datetime". Or, "future falls under a category known as ISO8601 datetime". So here is how to get a future which is two minutes away from any given datetime:

```
$f =
   new Future(
      new FromISO8601('2020-05-04 18:26:54+03'),
      new NMinutes(2)
   );
```

Methods are mostly text representation of an object. The one of a ISO8601DateTime object is a human-readable string in ISO8601 format. So if you want to output it either for viewing or persisting, just call the `value()` method. If nginx and http had spoken Object language, methods wouldn't have been needed for that purpose.

CHAPTER 16

How to subtract seconds, minutes, hours, days and all from a given datetime

First of all, you don't want to subtract actually. What you need is some point in the past. You can identify it by a datetime that acts as a starting point relative to which you want to calculate the past, and an interval you want to shift your starting point by. Here is how it might look:

```
use Meringue\ISO8601DateTime\FromISO8601 as DateTimeFromISO8601String;
use Meringue\ISO8601Interval\Floating\FromISO8601 as IntervalFromISO8601String;

new Past(
    new DateTimeFromISO8601String('2014-11-21T06:04:31+00:00'),
    new IntervalFromISO8601String('P1Y2M21DT24H56M26S')
);
```

If you don't want any further transformation, you can invoke a `value()` method and output the result:

```
(new Past(
    new DateTimeFromISO8601String('2014-11-21T06:04:31+00:00'),
    new IntervalFromISO8601String('P1Y2M21DT24H56M26S')
))
    ->value(); // returns 2013-08-30T05:08:05+00:00
```

There are some shortcuts for most typical intervals. You might benefit from `OneMinute`, `OneHour`, `OneDay`, `OneMonth`, and `OneYear`. Besides, there are `N`-counterparts, just in case you need two years for example:

```
(new Past(
    new DateTimeFromISO8601String('2014-11-21T06:04:31+00:00'),
    new NYears(2)
))
    ->value(); // returns 2012-11-21T06:04:31+00:00
```

There are at least two ways actually to define an interval. The first one is already covered above: you can use standard ISO8601 interval notation, like `P1Y2M21DT24H56M26S`, or shortcut meringue classes. The second option is to *identify an interval by two dates*. For example, I have to significant points in time, I have their absolute values, but I don't know an interval in advance. So I can write:

```
use Meringue\ISO8601Interval\WithFixedStartDateTime\FromRange as IntervalFromRange;

(new IntervalFromRange(
    new FromISO8601('2017-07-03T14:27:39+00:00'),
    new FromISO8601('2017-08-28T14:29:47+00:00')
))
    ->value(); // returns P0Y1M25DT0H2M8S
```

In the same vein, you can *obtain any datetime in the future*.

If you want to *convert it to specific timezone*, you can use `AdjustedAccordingToTimeZone` class:

```
(new AdjustedAccordingToTimeZone(
    new Past(
        new DateTimeFromISO8601String('2014-11-21T06:04:31+00:00'),
        new NYears(2)
    ),
    new CET()
))
    ->value(); // returns 2012-11-21T07:04:31+01:00
```

CHAPTER 17

How to get the last day of a month

If you know *what* it is that you need, you're halfway there. If you want to find the last day of a month, there must be a class of the same name. Besides, this class must implement some kind of `DateTime` interface or extend the same kind of abstract class. This fact indicates that the last day of a month *is a* datetime. And here you go, there is a `TheLastDayOfAMonth` indeed. That's how you can obtain a last day of some datetime's month:

```
(new TheLastDayOfAMonth(
    new FromISO8601('2020-02-21T23:28:04+07:00')
))
    ->value(); // returns 2020-02-29T00:00:00+07:00
```

If you want to find the last day of the current month, simply pass a *current datetime*:

```
(new TheLastDayOfAMonth(
    new Now()
))
    ->value();
```

Besides, you might want to find the first day of a month. It's carried out with `TheFirstDayOfAMonth` and is pretty much the same with the above:

```
(new TheFirstDayOfAMonth(
    new FromISO8601('2020-02-21T23:28:04+07:00')
))
    ->value(); // returns 2020-02-01T00:00:00+07:00
```

Getting the first day of the current month is as easy as

```
(new TheFirstDayOfAMonth(
    new Now()
))
    ->value();
```

There are similar cases covered when you need to find the beginning of something or the end of something. For example, you can *find the beginning of a day* and *a start of a week*. Besides, you can add your own shortcut class for that.

CHAPTER 18

How to calculate a datetime by a day of any week

Say, you want to know what date was on some particular Tuesday. Now the question is, how's that Tuesday identified?

18.1 By specific week and day of week

The first option is that you know that June, 5th, 2020 happened to belong to the week you're interested in. In this case, we could split this problem into two easier tasks: first, we find the specific week you're interested in, and after that, we can find which date was Tuesday. The first task can be carried out by the `Meringue\Date\Week\FromISO8601DateTime` class, which *is a* `Week`. In other words, it represents a week that the passed datetime falls under:

```
(new FromISO8601DateTime(
    new FromISO8601('2020-06-27T15:47:11+07:30')
))
    ->value();
```

The week above starts at 2020-06-22T00:00:00+07:30. Semantics of the `value()` method is exactly that: it's the *beginning of the week*. Actually, it doesn't really matter. What matters is that `FromISO8601DateTime` represents the week we need.

After that, we can find a datetime of that week's Tuesday. Once again, what we need is a datetime which is identified by a week and a day of a week. Here it goes:

```
(new FromWeekAndDayOfAWeek(
    new FromISO8601DateTime(
        new FromISO8601('2020-06-27T15:47:11+07:30')
    ),
    new Tuesday()
))
    ->value(); // voila, 2020-06-23T00:00:00+07:30
```

18.2 By specific week and day of week

The second option is when you know exactly how many weeks ago was your Tuesday. In this case, you can find Tuesday of your current week, and then calculate a datetime which was N weeks ago. For example, today is June 24th, and I want to find out a datetime of Tuesday that was three weeks ago. Here's how it can be done:

```
(new FromWeekAndDayOfAWeek(
    new FromISO8601DateTime(
        new FromISO8601('2020-06-27T15:47:11+07:30')
    ),
    new Tuesday()
))
    ->value(); // voila, 2020-06-23T00:00:00+07:30
```

I'm sure there are many more interesting cases. Most of them though can be solved with basic building blocks from this library.

CHAPTER 19

How to get a timestamp in php

First of all, what is a unix timestamp? It's a number of seconds since 00:00:00, January 1st, 1970 UTC. When you already know **what** you want, instead of **how** you want it, you're good to go and discover if there is a class that serves your needs. In the case of a timestamp, there is one, and it's called `SecondsSinceJanuary1st1970`. If you want to find out current unix timestamp, you can pass current datetime as an argument:

```
$c = new SecondsSinceJanuary1st1970(new Now());
```

In the same vein, you can pass any other ISO8601 datetime:

```
$c =
   new SecondsSinceJanuary1st1970(
      new Max(
         new Now(),
         new Future(
            new FromISO8601('1986-05-04 00:30:00+03'),
            new NYears(34)
         )
      )
   );
```

As usual, if you want a textual representation, `value()` method is just for that.

Also note that timestamp doesn't depend on a timezone, since it's the number of seconds since January 1st midnight 1970 in **UTC** timezone. This timezone is embedded into the very definition of a timestamp.

CHAPTER 20

How to format a date

Meringue datetime formatting relies on ISO8601 standard. It simply mirrors the behavior of built-in php date function. You might ask why on Earth I did this if no new features are introduced. The intention is that I personally didn't want to derail from a customary set of abstractions. It feels natural to work with `ISO8601DateTime` objects across-the-board. I'd like to obtain its value only when I need to output it: either for a `stdout`, or for a database. If I used a `date` function, I'd need to invoke a `ISO8601DateTime`'s `value()` method each time. I wanted to avoid it, so here is how my code looks like:

```
(new ISO8601Formatted(
    new FromISO8601('2017-07-03T14:27:39+04:30'),
    'l jS'
))
    ->value(); // returns Monday 3rd
```

Object composition just looks more neatly to my taste. It adds uniformity.

CHAPTER 21

How to get a current year

As usual, if a current year is what you need, chances are there should be an entity in code which represents exactly this abstraction. Look for an interface or abstract class called `Year`, is there any? In meringue, there is one.

There are currently two ways you can get a year: from some datetime and from integer. Hence, two classes: `FromISO8601DateTime` and `FromInt`. If you're up to find a current year, I believe you could already know what to do: pass a current datetime to `FromISO8601DateTime`:

```
(new FromISO8601DateTime(
    new Now()
))
    ->value();
```

If you want to get a year from some datetime, you can do the following:

```
(new FromISO8601DateTime(
    new FromISO8601('2020-02-11 15:21:47+03')
))
    ->value(); // returns 2020
```

It finds a current year in local timezone, not UTC. For example, the output in the following case would be 2020 already:

```
(new FromISO8601DateTime(
    new FromISO8601('2020-01-01 00:01:47+03')
))
    ->value();
```

If you want to obtain a year in some other timezone, you'd want to *adjust the time to that timezone* first, and then construct a `Year` object.

CHAPTER 22

How to get a month in a datetime in php

Following meringue philosophy, the key is to know *what* you need. With this approach, you end up with objects having counterparts in real life. Usually, the most simple way is to follow the language someone uses to describe a problem. Words refer to existing objects, otherwise, those words wouldn't exist. So listen carefully to the words your domain expert uses.

In the domain of dates and times, it's all quite simple. If you hear "month", most probably you need a class whose objects are referred by this word. Is it a single class or multiple classes? A "month" is a general term. There are a lot of things actually fall under this concept, and they all have different properties. Month can be plainly May or June. Or it can be identified by an ISO8601 datetime. Or by ordinal number. Thus, I've created a *Month* interface indicating this category.

One concrete implementation put in a title of this entry is characterized by the specific datetime. Say, you have an ISO8601 datetime, like 2020-07-20T10:45:21+03:00. And you want to obtain a month from it. the resulting object, falling under the "Month" category, as a very specific property: it's obtained from, or identified by, a datetime. Hence the name: *FromISO8601DateTime*.

If you want to have a current month, you can pass a current datetime to it:

```
(new FromISO8601DateTime(
    new Now()
))
    ->value();
```

Similarly, if you want to create a month from any other datetime, it looks like the following:

```
(new FromISO8601DateTime(
    new FromISO8601('2020-02-11 15:21:47+03')
))
    ->value(); // returns 2
```

In the same vein, you can create a month object not with a datetime, but from its ordinal number:

```
(new FromInt(7))->value(); // no surprises, it's 7
```

Other datetime parts have pretty much the same functionality and approach.

CHAPTER 23

Meringue documentation

Meringue is an object-oriented implementation of datetime functionality in php. It's built atop the few fundamental universal abstractions, so it's minimalistic, intuitive and extendable. At the same time, it allows carrying out complex datetime calculations due to its inherent declarative nature.