How much time does it take to write tests? A case study

A presentation given at EuroPython, Dublin, 13 July 2022

Antonis Christofides, antonis@antonischristofides.com

1. Introduction

There was a URL, http://openmeteo.org/api/tsdata/855/, where you could get some data. It doesn't work now, because the application has changed since then and uses different URLs now. But at that time, four years ago, that was the URL, and this is the code that was running behind the scenes in order to give you the data:

```
def get(self, request, pk, format=None):
    timeseries = models.Timeseries.objects.get(pk=int(pk))
    self.check_object_permissions(request, timeseries)
    response = HttpResponse(content_type='text/plain')
    pd2hts.write(timeseries.get_data(), response)
    return response
```

For some time this worked, until eventually a user got an internal server error. Django emailed the error and traceback to me. The error was "Timeseries does not exist", and the location was the second line above. Diagnosis was trivial, and for people who have some experience with Django, the fix is also trivial:

```
def get(self, request, pk, format=None):
    try:
        timeseries = models.Timeseries.objects.get(pk=int(pk))
    except Timeseries.DoesNotExist:
        raise 404
    self.check_object_permissions(request, timeseries)
    response = HttpResponse(content_type='text/plain')
    pd2hts.write(timeseries.get_data(), response)
    return response
```

Attentive readers may notice another issue, that "pk" might not necessarily be an integer, which might cause a "ValueError". Either this has already been checked elsewhere by the time we get here, or we didn't notice it. It doesn't matter, this code is obsolete anyway. The point I want to make is that this was a very easy fix. It likely took less than 5 minutes to diagnose and fix. But we have the rule that we pretty much always write tests for our fixes.

So we wrote this test:

```
def test_get_nonexisting_timeseries_data(self):
    response = self.client.get("/api/tsdata/9999999/")
    self.assertEqual(response.status_code, 404)
```

It looks easy alright. However, those of you who have done this know the frustration of having to write tests for such a trivial fix. You really need to do some context switching in your brain and put it in testing mode. You need to figure out the details of how to test. Which file is the test going to be in? Can you add some tests to an existing test case? Or do you need to add a new test case? Where is it going to inherit from? Do you need test data? How will you create them? Do you already have some test data that you can use? How are you going to import it? And so on. In fact even this test actually has a tricky decorator that I removed for this presentation. And actually it is an unusually easy case. In fact I had assigned it as a first assignment to a programmer who had no experience writing tests.

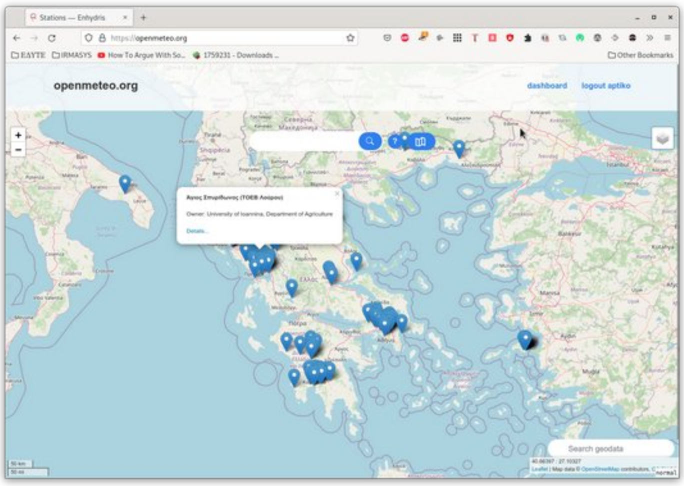
Some people will tell me we should write the tests first, run them and verify they fail, then write the fix, then check that the tests pass. I agree. But the point is the same. You have a trivial fix that takes you five minutes, and you might need to spend 30 minutes or an hour to write the tests, regardless whether this is before or after the main code. So, for some years, I have been wondering: Am I spending too much time writing tests? Am I spending twice as much time writing tests as writing main code? Am I overdoing it? Am I too slow because of this?

So I measured it. When I submitted the application for this talk I hadn't measured yet and I didn't know what the results would be. I can tell you I was surprised. Things were not what I expected.

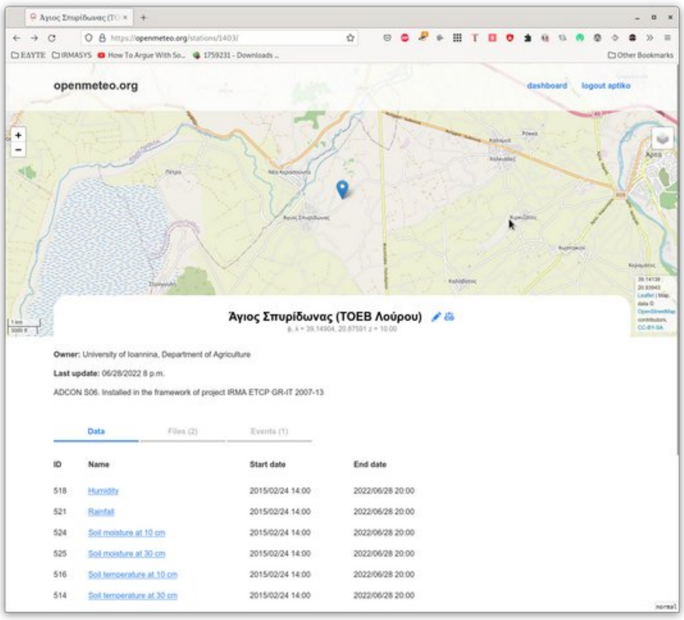
One of the surprises had to do with the example I showed you. Trying to find such a simple example I searched and searched and searched the repository history of several pieces of software. I thought I would find many examples of trivial fixes where testing would be the hardest part. But, as you noticed, I ended up showing you some old obsolete code, because there aren't nearly as many examples as I thought. Reality is that, even trivial fixes often need some refactoring if you want to keep the code readable. Even in this case, which is almost as trivial as it gets, someone could argue you'd benefit from some refactoring. So although the diagnosing and the fixing took maybe less than five minutes, refactoring the code would take another ten perhaps. So now the amount of time for the test becomes smaller as a percentage of the total time, not to mention the fact that you practically can't refactor the code without good tests.

2. The case study

The software is called Enhydris. It's available at https://openmeteo.org/. Here is a screenshot:



There are meteorological stations on the map, and each station makes measurements. If you go to the detail page for a station, you can see the variables for which it has data:



Άγιος Σπυρίδωνας (ΤΟΕΒ Λούρου)

φ, λ = 39.14904, 20.87591 z = 10.00

Owner: University of Ioannina, Department of Agriculture

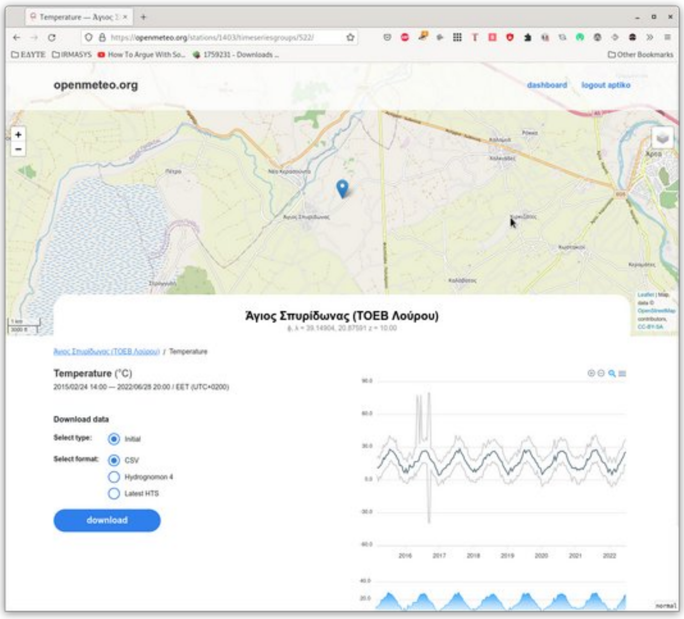
Last update: 06/28/2022 8 p.m.

ADCON S06. Installed in the framework of project IRMA ETCP GR-IT 2007-13

Data | Files (2) | Events (1)

ID	Name	Start date	End date
518	Humidity	2015/02/24 14:00	2022/06/28 20:00
521	Rainfall	2015/02/24 14:00	2022/06/28 20:00
524	Soil moisture at 10 cm	2015/02/24 14:00	2022/06/28 20:00
525	Soil moisture at 30 cm	2015/02/24 14:00	2022/06/28 20:00
516	Soil temperature at 10 cm	2015/02/24 14:00	2022/06/28 20:00
514	Soil temperature at 30 cm	2015/02/24 14:00	2022/06/28 20:00

And you can also find out what it has measured for each variable:



Άγιος Σπυρίδωνας (ΤΟΕΒ Λούρου)

φ, λ = 39.14904, 20.87591 z = 10.00

Άγιος Σπυρίδωνας (ΤΟΕΒ Λούρου) / Temperature

Temperature (°C)

2015/02/24 14:00 — 2022/06/28 20:00 / EET (UTC+0200)

Download data

Select type: [✓ Initial]

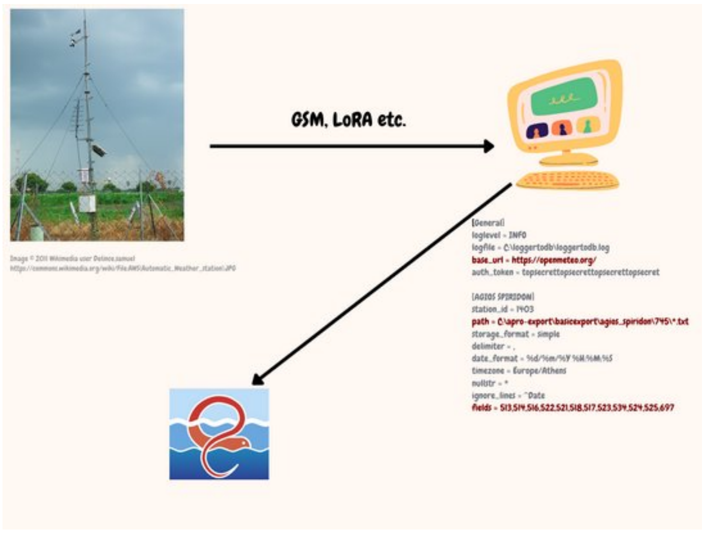
Select format: [✓ CSV] Hydrognomon 4 Latest HTS

It's that simple to begin with, although, as it often happens, the devil is in the details, so you can't rewrite it overnight.

Enhydris is free software, and all the code I'll show you is available at https://github.com/openmeteo/enhydris.

Now, the question in our case is how to automatically update the data, how to acquire the data, that is.

Meteorological stations are often installed in remote areas and they aren't connected to the power network or to the fixed telephone network. They are usually powered with a battery and a solar panel, and they have some means of communication. At the other end of this communication there's usually a computer, usually with Windows, that runs some kind of software, usually provided by the manufacturer of the station, and this periodically downloads the data from the station and stores it in files.

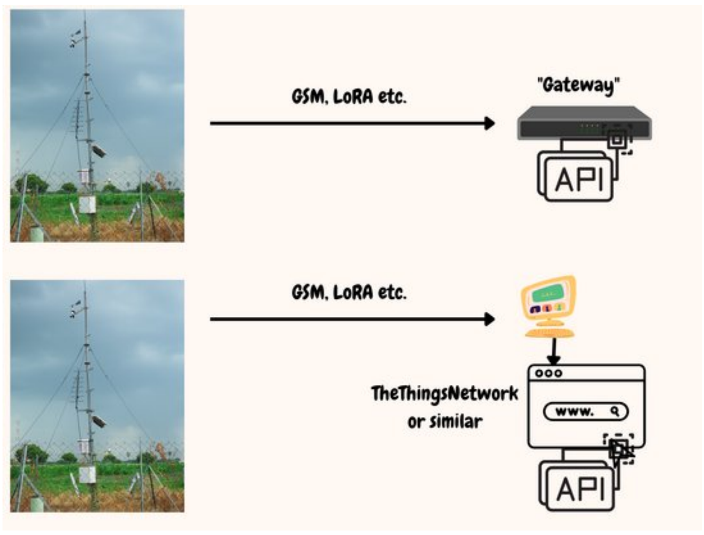

GSM, LoRA etc.
Image © 2011 Wikimedia user Deinos.samuel
https://commons.wikimedia.org/wiki/File:AWS/Automatic_Weather_station.JPG

```
[General]
loglevel = INFO
logfile = C:\loggertodb\loggertodb.log
base_url = https://openmeteo.org/
auth_token = topsecrettopsecrettopsecrettopsecret

[AGIOS SPIRIDON]
station_id = 1403
path = C:\apro-export\basicexport\agios_spiridon\745\*.txt
storage_format = simple
delimiter = ,
date_format = %d/%m/%Y %H:%M:%S
timezone = Europe/Athens
nullstr = *
ignore_lines = ^Date
fields = 513,514,516,522,521,518,517,523,534,524,525,697
```

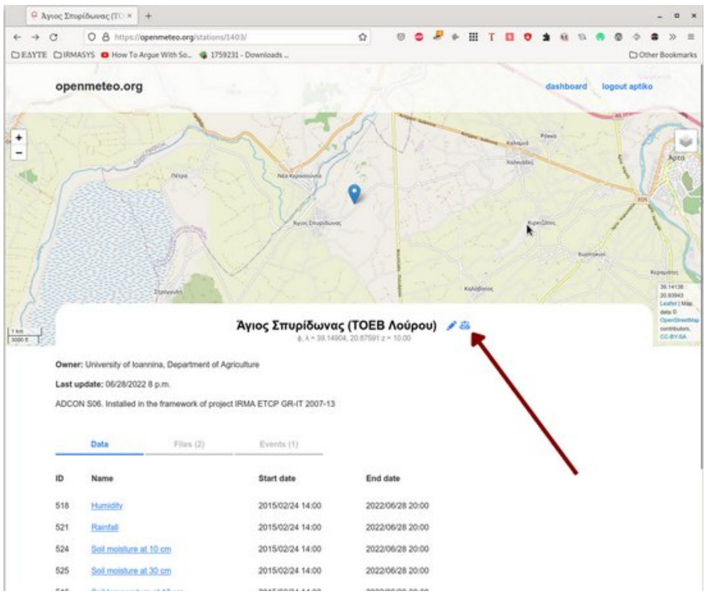
So we've written accompanying command line software and we run it on that computer with cron or with Windows Scheduler. We give it a configuration file that tells it where the data files are and to which station and time series they correspond, so that it can upload them. You might not be able to read the configuration file if it's too small, but it doesn't really matter; it's in INI format. It specifies the URL to which the data should be uploaded, with what credentials, which file contains the data, the correspondence of columns in the file to time series ids in the database, and some details about how the file is formatted.

But this way of communication has been changing in the last few years. More and more often, the data somehow end up being served by an API.



In some setups there's a specialized computer, the size of an aDSL router, at the end of the link, and that computer offers the data with an API. In some other setups it's more or less the same as the old setup, but the manufacturer's software that runs on the computer will upload the data to a service that in turns offer the data with an API. So it's going to be both more reliable and simpler for users if we just give them a form where they can configure such API access and have Enhydris occasionally connect to the API and pull the data, rather than having software that must be installed elsewhere and push the data to Enhydris.

So, last November I decided to add this feature to Enhydris. I can show you how it ended up. Station administrators got this little button which they can press and configure telemetry:



Άγιος Σπυρίδωνας (ΤΟΕΒ Λούρου)

φ, λ = 39.14904, 20.87591 z = 10.00

Owner: University of Ioannina, Department of Agriculture

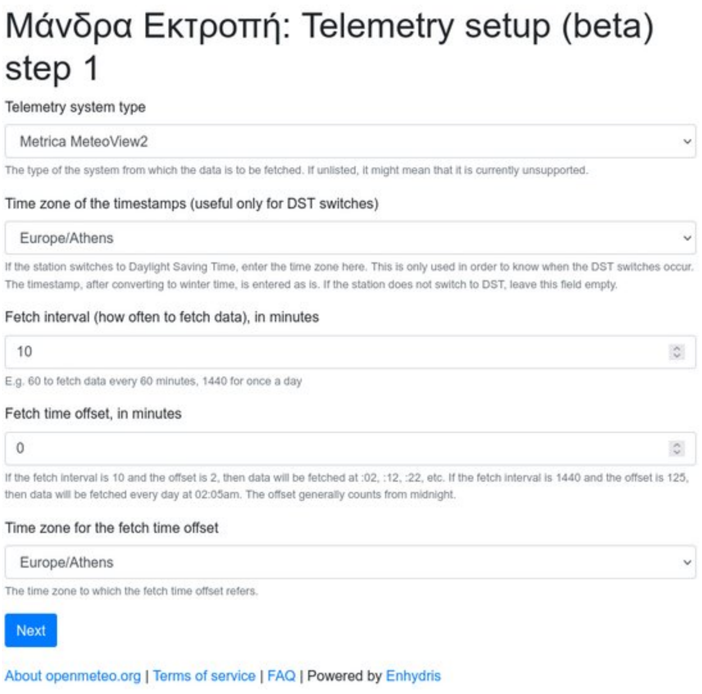
Last update: 06/28/2022 8 p.m.

ADCON S06. Installed in the framework of project IRMA ETCP GR-IT 2007-13

Data | Files (2) | Events (1)

ID	Name	Start date	End date
518	Humidity	2015/02/24 14:00	2022/06/28 20:00
521	Rainfall	2015/02/24 14:00	2022/06/28 20:00
524	Soil moisture at 10 cm	2015/02/24 14:00	2022/06/28 20:00
525	Soil moisture at 30 cm	2015/02/24 14:00	2022/06/28 20:00

Then they get a form:



Μάνδρα Εκτροπή: Telemetry setup (beta) step 1

Telemetry system type

Metrica MeteoView2

The type of the system from which the data is to be fetched. If unlisted, it might mean that it is currently unsupported.

Time zone of the timestamps (useful only for DST switches)

Europe/Athens

If the station switches to Daylight Saving Time, enter the time zone here. This is only used in order to know when the DST switches occur. The timestamp, after converting to winter time, is entered as is. If the station does not switch to DST, leave this field empty.

Fetch interval (how often to fetch data), in minutes

10

E.g. 60 to fetch data every 60 minutes, 1440 for once a day

Fetch time offset, in minutes

0

If the fetch interval is 10 and the offset is 2, then data will be fetched at :02, :12, :22, etc. If the fetch interval is 1440 and the offset is 125, then data will be fetched every day at 02:05am. The offset generally counts from midnight.

Time zone for the fetch time offset

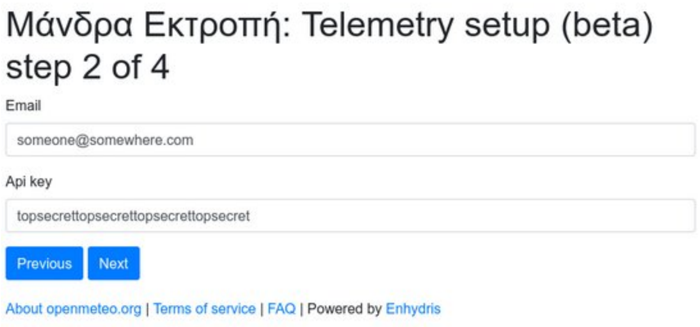
Europe/Athens

The time zone to which the fetch time offset refers.

Next

About openmeteo.org | Terms of service | FAQ | Powered by Enhydris

Different manufacturers have different APIs. So the first thing you need to do is choose what type of API you have. So far I've implemented only a single API, which is called Meteoview2. And in the first form of this wizard you also tell it some details about how often you want to fetch data and about the time zone of the data. After that you get forms that are different for each API. This one first asks you to login:



Μάνδρα Εκτροπή: Telemetry setup (beta) step 2 of 4

Email

someone@somewhere.com

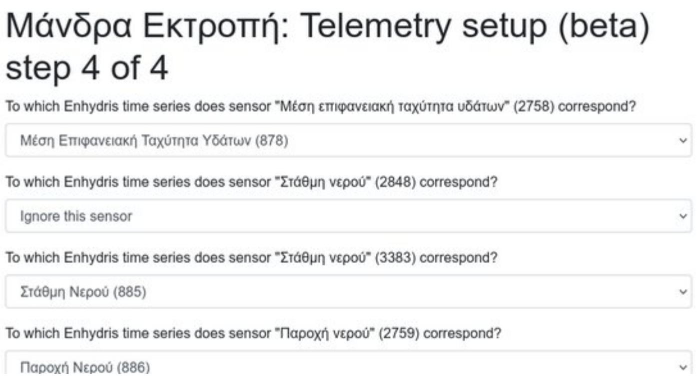
Api key

topsecrettopsecrettopsecrettopsecret

Previous Next

About openmeteo.org | Terms of service | FAQ | Powered by Enhydris

After you login, it finds out what data is available on the API and asks you to match the data on the API to the time series that you have defined on Enhydris:



Μάνδρα Εκτροπή: Telemetry setup (beta) step 4 of 4

To which Enhydris time series does sensor "Μέση επιφανειακή ταχύτητα υδάτων" (2758) correspond?

Μέση Επιφανειακή Ταχύτητα Υδάτων (878)

To which Enhydris time series does sensor "Στάθμη νερού" (2848) correspond?

Ignore this sensor

To which Enhydris time series does sensor "Στάθμη νερού" (3383) correspond?

Στάθμη Νερού (885)

To which Enhydris time series does sensor "Παροχή νερού" (2759) correspond?

Παροχή Νερού (886)

And after you configure all that it's going to work in the background and keep fetching data from the API. That is much easier for the user than having to install and configure the command line program and it's much more reliable.

It doesn't really matter if you don't understand all this clearly. What matters is that I had a specific problem to solve, a specific new functionality to develop. And I also had to write tests for this new functionality.

Now, some people will tell you to use test-driven development. Test-driven development has three rules:

- You may not write production code until you have written a failing unit test.
- You may not write more of a unit test than is sufficient to fail.
- You may not write more production code than is sufficient to pass the currently failing test.

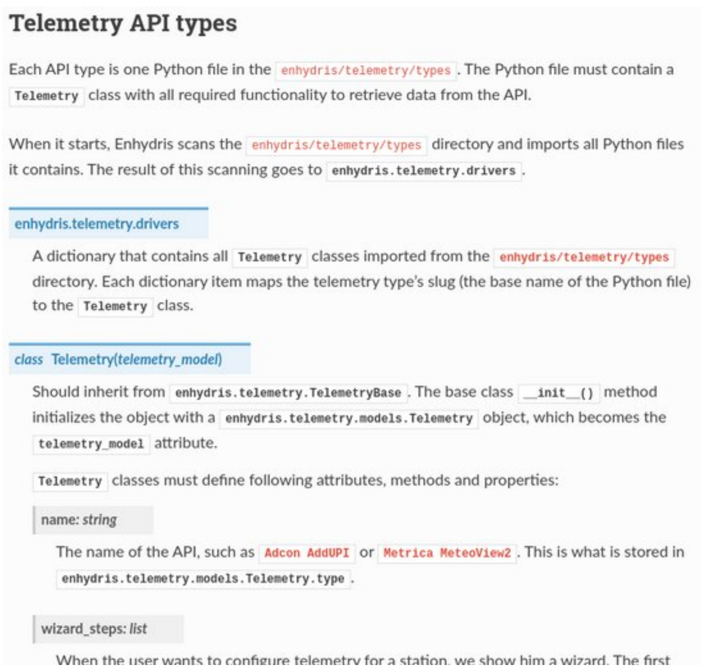
These are from the excellent book "Clean Code" by Robert C. Martin, 2009, page 122. The second rule, rephrased says that the test that you are writing must be the simplest possible and should test just a single thing.

It's hard to grasp the implications of these three rules until you follow them. Just try then for a week and you'll see how different your coding is going to be.

I think that test-driven development is a good idea, and I often program like this. But not always. The problem is that test-driven development works when you have a clear idea of what code you are going to write. But very often when I program I don't really know how I'm going to approach the problem. I need to experiment much, and I don't know beforehand how to break the new functionality into units that I can test.

In this case, I needed to develop a system that supports many different APIs. I used the word "driver" here. So I needed to develop a system where drivers for different APIs could be added. How would this work? Would a driver be a subclass of some base class? Would it be a file that offers an entry point? Would it be a module? What would the base class look like? What would the module or the entry point look like? And many more details.

So I thought about it and made some design and wrote it down, like this:



Telemetry API types

Each API type is one Python file in the `enhydris/telemetry/types`. The Python file must contain a `Telemetry` class with all required functionality to retrieve data from the API.

When it starts, Enhydris scans the `enhydris/telemetry/types` directory and imports all Python files it contains. The result of this scanning goes to `enhydris.telemetry.drivers`.

enhydris.telemetry.drivers

A dictionary that contains all `Telemetry` classes imported from the `enhydris/telemetry/types` directory. Each dictionary item maps the telemetry type's slug (the base name of the Python file) to the `Telemetry` class.

class Telemetry(*telemetry_model*)

Should inherit from `enhydris.telemetry.TelemetryBase`. The base class `__init__()` method initializes the object with a `enhydris.telemetry.models.Telemetry` object, which becomes the `telemetry_model` attribute.

`Telemetry` classes must define following attributes, methods and properties:

name: *string*

The name of the API, such as `Adcon AddUPI` or `Metrica MeteoView2`. This is what is stored in `enhydris.telemetry.models.Telemetry.type`.

wizard_steps: *list*

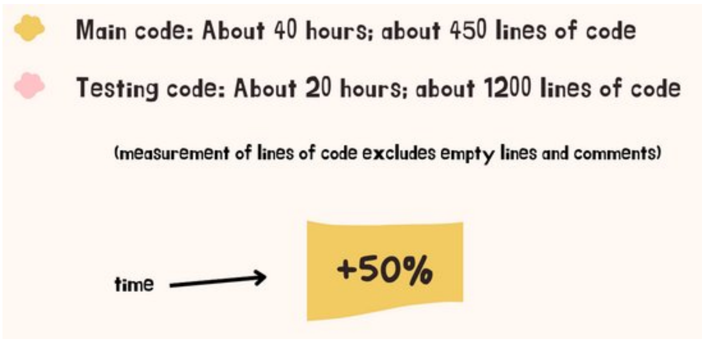
When the user wants to configure telemetry for a station, we show him a wizard. The first

This is part of the finished documentation, it's not what I initially wrote. After I wrote a few lines of code I realized it doesn't work and I changed the entire design. Then I changed it again, and I kept changing it and changing it. I can't imagine how I could have done test-driven development in such a heavily experimental process. So I didn't write any tests at all. I wrote the base stuff, I wrote the first driver, Meteoview2, more or less at the same time, and when I finished and it was working, and I hadn't yet written the unit tests, I wanted to submit a proposal for a talk to this conference, and I thought about delivering this talk. Actually it was a neat trick because I didn't have any deadlines or any kind of pressure about continuing to work on this, so promising to deliver the talk actually helped me sit down and write these tests.

3. Results

As I said, results surprised me.

I spent about a third of my time testing, and the other two thirds on all the rest together, including coding, designing, redesigning and documenting. I thought testing would be more than the rest, but it turns out it was half, and for me this is a real bargain.


Main code: About 40 hours; about 450 lines of code
Testing code: About 20 hours; about 1200 lines of code
(measurement of lines of code excludes empty lines and comments)
time
+50%

Testing was pretty much exhaustive. I did the next best thing to test-driven development. I commented out all my code. Then I followed the three rules of the poor man's test-driven development:

- You may not **uncomment** production code until you have written a failing unit test.
- You may not write more of a unit test than is sufficient to fail.
- You may not **uncomment** more production code than is sufficient to pass the currently failing test.

They are pretty much the same as those of test-driven development, except that I've changed two words—it used to say "write", and I made it "uncomment". They are not as good as the real thing, but they're better than nothing.

Now, there are some caveats. The first is that every case is different. This was the test writing cost in this particular case. I won't be surprised if it's very different in other cases.

The second caveat has to do with how experienced the programmers are. I am experienced enough, but programmers relatively inexperienced in testing might write the main code faster and the tests slower. I have an example from myself. For many years, mocking has been a pain for me. I had understood the concepts of mocking, but each time I was mocking something it refused to work and I was working for hours trying to make the damn thing work. The worst thing is that years went by and things weren't becoming any better. In the end I settled with the fact that I'll always suck in mocking. I was afraid of this case because it's about fetching data from an external API and therefore I'd have lots of mocking to do. But, and here's another surprise, the mocks I wrote for this particular case worked without serious problems. I mean, they didn't work right away, but I was getting errors that I could immediately understand and say "oh, this mock needs a comma there and that mock needs this little fix there", and I made the mocks work reasonably easy. My 2019 self would have had much more trouble writing these mocks. Nothing would work and I would need to read the mocking documentation again, and add breakpoints, and see why the mocks don't do what I thought they should do, and experiment, and start getting nerves, and then break some things, and so on. So maybe at last I'm becoming better at mocking just after I had lost hope. But you see that testing is yet another thing the programmer must learn, and mocking is yet another thing the programmer must learn. So I expect this result to vary based not only on the particulars of each case, but also on the experience and the abilities of each programmer.

But in any case, what is the alternative? Can we do without tests? We've already seen how a straightforward method with five lines of code and no control stuctures became slightly more complicated, and it won't be long before it eventually becomes unreadable and unmanageable.

The next step, now that I have the tests, is to refactor the code. In the right column I show you the worst method in the case study (it's just in order to give you an idea of how bad it is; some parts are redacted to make it not wrap as the width here is too small). There are another two or three methods that need refactoring. It's maybe two hours of work, enabled by the 20-or-so hours spent in writing tests.

So my conclusion is: write tests, it's a bargain.

```
def dispatch_other_step(self):
    station_id = self.station_id
    configuration = self.request.session.get(
        f"telemetry_{station_id}_configuration",
        {"station_id": station_id},
    )
    Form = self.telemetry.wizard_steps[self.seq - 2]
    if self.request.method == "POST":
        form = Form(self.request.POST)
        if form.is_valid():
            configuration.update(form.cleaned_data)
            self.request.session[
                f"telemetry_{station_id}_configuration"
            ] = configuration
            if self.seq <= len(self.telemetry.wizard_steps):
                kwargs = {"station_id": self.[...]}
                target = reverse("telemetry_wizard", [...])
            else:
                T[...].filter(station=self.station).delete()
                kwargs = self.[...]()
                T[...](station=self.station, **kwargs).save()
                msg = _("Telemetry has been configured")
                messages.add_message([...])
                target = reverse("station_detail", [...])
            return HttpResponseRedirect(target)
    else:
        form = Form(initial=configuration)
    return render(
        self.request,
        "enhydris/telemetry/wizard_step.html",
        {
            "station": self.station,
            "form": form,
            "seq": self.seq,
            "prev_seq": self.seq - 1,
            "max_seq": len(self.telemetry.wizard_steps) + 1,
        },
    )
```