# Learning Object-Oriented Programming, Design and TDD with Pharo

Stéphane Ducasse

March 11, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

# Contents

# Illustrations

CHAPTER 1

# Temperature Logger (under writing)

In this chapter you will implement a little temperature logger. It is really simple that it will help us to get started with Pharo and also with test driven development. We will collect temperate measurements and compute temperature average and display the temperatures in a graph. Near the end of the chapter, we will add the possibility to convert between celcius and fahrenheit. For this you will create a simple class and its tests.

We will show how to write test to specify the expected results. Writing tests is really important. It is one important tenet of Agile Programming and Test Driven Development (TDD). We will explain later why this is really good to have tests. For now we just implement them. We will also discuss a bit the fundamental aspects of float comparison and we will also present some loops.

## 1.1 First a test

Imagine that you want to monitor the different temperatures between the locations where you live and where you work. (This is a real scenario since the building where my office is located got its heating broken over winter and I wanted to measure and keep trace of the different temperatures in both locations.)

We will define tests to capture the expected behavior. First we define a test class named `TemperatureLoggerTest` within the package `TemperatureLogger`. It inherits from the class `TestCase`. This class is special, any method starting with `'test'` will be executed automatically, one by one each time on a new instance of the class (to make sure that tests do not interfere with each others).

```
TestCase subclass: #TemperatureLoggerTest
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'TemperatureLogger'
```

Here is a test method named `testRecords` representing a typical case. First, since we want to distinguish measurements based on the locations, we added the possibility to specify a location. Then we record temperatures.

```
TemperatureLoggerTest >> testRecords

    | office |
    office := TemperatureLogger new location: 'Office'.
    "Perform two measures that are logged"
    office recordTemperature: 19.
    office recordTemperature: 21.

    "We got effectively two measures"
    self assert: office recordCount equals: 2.
```

The test is structured the following way:

- Its selector starts with `test`, here the method is named `testRecords`.
- It creates a new instance of `TemperatureLogger` (it is called the *context* of the test or more technically its fixture). Note that for now the class `TemperatureLogger` is not defined yet but this is not a problem.
- Then we check using the message `assert:equals:` that the expected behavior is really happening. Here we test that we effectively recorded two measures.

## 1.2 Making the test pass

The first thing that we do is to define a class named `TemperatureLogger`. We add two instance variables to our class: `location` that will hold the name of the location of our measure and `measures` a collection that will hold all the recorded temperatures.

It means that each instance of the class `TemperatureLogger` will be able to have its own location and its own collection of measures.

```
Object subclass: #TemperatureLogger
    instanceVariableNames: 'location measures'
    classVariableNames: ''
    package: 'Converter'
```

We defined the method `initialize` to initialize our logger so that it can be correctly used. Such `initialize` method is executed each time new instance is created. We initialize such instance variables with the following values: we set by default the location to be 'home' and we initialize the `measures` instance variables to hold a collection in which we will add measurements. An ordered collection is a collection of objects that is ordered and can grow over time.

```
TemperatureLogger >> initialize
    location := 'Home'.
    measures := OrderedCollection new
```

**Note** The method `initialize` is executed each time we create a new object using the message `new` (sent to the class).

We add a method to set the location:

```
TemperatureLogger >> location: aNumber

    location := aNumber
```

Now we are ready to record a temperature in celsius.

## 1.3 Record temperature

What we do is to add a measure to our collection of measures.

```
TemperatureLogger >> recordTemperature: aNumber
    "Add the temperature to the list of recorded measures"

    measures add: aNumber
```

Then you should define the method `recordCount` which returns the number of measures done so far. Its implementation is to return the size of the collection holding the recorded temperatures.

```
TemperatureLogger >> recordCount

    ...
```

Our test should now pass. We suggest you save your code.

## 1.4 Logger recording improvements

Our previous test did not cover well the order in which the measures are added. We should make sure that the latest added record is really the correct one. Since in the future we will want to have summary for the measures have our measures nicely ordered may simplify our future tasks. We write a test to specify this point.

```
TemperatureLoggerTest >> testLatestRecord

    | office |
    office := TemperatureLogger new location: 'Office'.
    office recordTemperature: 19.
    office recordTemperature: 18.
    self assert: office latestRecord equals: 18.
    office recordTemperature: 21.
    self assert: office latestRecord equals: 21.
```

To be able to implement the method `latestRecord` we should understand where to the ordered collection add its elements. Let us have a look at the `add:` method of the class `OrderedCollection` implemented.

```
OrderedCollection >> add: newObject

    ^ self addLast: newObject
```

It means that the elements are added at the end of the collection, therefore the definition of the method `latestRecord` can be defined as follows:

```
TemperatureLogger >> latestRecord

    ^ measures last
```

Our tests should pass.

## 1.5 Understanding design decision

We want to show you that an internal representation choice such as the ordering of the elements can make a difference when implementing certain behavior. In addition we want to show you that with a good encapsulation we can change the internal representation and still get tests pass. Note that the tests also give us this agility to change and control the changes.

```
TemperatureLoggerTest >> testLatestRecords

    | office |
    office := TemperatureLogger new location: 'Office'.
    office recordTemperature: 16.
    office recordTemperature: 17.
    office recordTemperature: 19.
    office recordTemperature: 18.
    office recordTemperature: 21.
    self assert: (office latestRecords: 3) equals: #(21 18 19)
     asOrderedCollection
```

```
TemperatureLogger >> latestRecords: aNumber

    | records reversed |
    records := OrderedCollection new.
    reversed := measures reversed.
    1 to: aNumber do: [ :each |
            records add: ( reversed at: each ) ].
    ^ records
```

It is not really nice since reversed copy the collection of measures. We can proposer another solution that goes from the end of the collection and access the correct number of elements.

recordTemperature: aNumber ”Add the temperature to the list of recorded measures” measures addFirst: aNumber

```
TemperatureLogger >> latestRecords: aNumber
    | records |
    records := OrderedCollection new.
    1 to: aNumber do: [:i | records add: (measures at: i)  ].
    ^ records
```

## 1.6 Adding average computation

```
TemperatureLoggerTest >> testAverageTemperature

    | office |
    office := TemperatureLogger new location: 'Office'.
    office recordTemperature: 19.
    office recordTemperature: 21.
    office recordTemperature: 17.

    self assert: office average equals: 19
```

There are multiple ways to implement `average`: one is to use a temporary variable representing the sum, to initialize it to zero and to loop over the measures and add to the sum each measure and finally to divide the sum by the number of measures.

```
TemperatureLogger >> average
   ...
```

## 1.7 Solutions

### First test

```
TemperatureLogger >> recordCount

    ^ measures size
```

### Average

```
TemperatureLogger >> alternateAverage

        | sum |
        sum := 0.
        measures do: [ :aMeasure | sum := sum + aMeasure ].
        ^ sum / measures size
```

```
TemperatureLogger >> average

    ^ measures average
```

It is worth looking at the implementation of `average`

```
Collection >> average
    ^ self sum / self size
```

What we do is that we add pair with the time and the value to our collection of measures.

Since the temperature often depends on the moment during the day I want to log the date and time with each measure.

Then we can request a converter for all the dates (message `dates`) and temperatures (message `temperatures`) that it contains.

```
TemperatureConverterTest >> testLocationAndDate

    | office |
    office := TemperatureConverter new location: 'Office'.
    "perform two measures that are logged"
    office measureCelsius: 19.
    office measureCelsius: 21.

    "We got effectively two measures"
    self assert: office measureCount = 2.

    "All the measures were done today"
    self assert: office dates equals: {Date today . Date today}
    asOrderedCollection.

    "We logged the correct temperature"
    self assert: office temperatures equals: { 19 . 21 }
    asOrderedCollection
```

```
TemperatureConverter >> measureCelsius: aTemp
    measures add: DateAndTime now -> aTemp
```

To support tests we also define a method returning the number of current measure our instance holds.

```
TemperatureConverter >> measureCount
    ^ measures size
```

We now define three methods returning the sequence of temperatures, the dates and the times. Since the time has a microsecond precision it is a bit difficult to test. So we only test the dates.

```
TemperatureConverter >> temperatures
    ^ measures collect: [ :each | each value ]
```

```
TemperatureConverter >> dates
    ^ measures collect: [ :each | each key asDate ]
```

```
TemperatureConverter >> times
    ^ measures collect: [ :each | each key asDate ]
```

Now we can get two converters each with its own location and measurement records. The following tests verifies that this is the case.

```
TemperatureConverterTest >> testTwoLocations

    | office home |
    office := TemperatureConverter new location: 'office'.
    office measureFahrenheit: 19.
    office measureFahrenheit: 21.
    self assert: office location equals: 'office'.
    self assert: office measureCount equals: 2.
    home := TemperatureConverter new location: 'home'.
    home measureFahrenheit: 22.
    home measureFahrenheit: 22.
    home measureFahrenheit: 22.
    self assert: home location equals: 'home'.
    self assert: home measureCount equals: 3.
```

## 1.8 The class TemperaturLogger

The class `TemperaturLogger` is defined as shown below. You could have define it before defining the class `TemperaturLoggerTest` using the class definition below:

```
Object subclass: #TemperaturLogger
    instanceVariableNames: ''
    classVariableNames: ''
    package: 'Converter'
```

This definition in essence, says that:

- We want to define a new class named `TemperaturLogger`.
- It has no instance or class variables (`''` means empty string).
- It is packaged in package `Converter`.

Usually when doing Test Driven Development with Pharo, we focus on tests and lets the system propose us some definitions. Then we can define the method as follows.

The system may tell you that the method is an utility method since it does not use object state. It is a bit true because the converter is a *really* simple object. For now do not care.

Your test should pass. Click on the icon close to the test method to execute it.

## 1.9 Converting Fahrenheit to Celcius

Converting from Fahrenheit to Celsius is done with a simple linear transformation. The formula to get Fahrenheit from Celsius is F = C * 1.8 + 32. Let us write a test covering such transformation. 30 Celsius should be 86 Fahrenheit.

```
testCelsiusToFahrenheit

    | converter |
    converter := TemperatureConverter new.
    self assert: ((converter convertCelsius: 30) = 86.0)
```

The message `assert:` expects a boolean. Here the expression `((converter convertCelsius: 30) = 86.0)` returns a boolean. `true` if the converter returns the value 86.0, `false` otherwise.

The testing frameworks also offers some other similar methods. One is particularly interesting: `assert:equals:` makes the error reporting more user friendly. The previous method is strictly equivalent to the following one using `assert:equals:`.

```
testCelsiusToFahrenheit

    | converter |
    converter := TemperatureConverter new.
    self assert: (converter convertCelsius: 30) equals: 86.0
```

The message `assert:equals:` expects an expression and a result. Here `(converter convertCelsius: 30)` and `86.0`. You can use the message you prefer and we suggest to use `assert:equals:` since it will help you to understand your mistake by saying: 'You expect 86.0 and I got 30' instead of simply telling you that the result is false.

Now we can define the method `convertCelcius:` as follows:

```
TemperatureConverter >> convertCelsius: anInteger
    "Convert anInteger from celsius to fahrenheit"

    ^ ((anInteger * 1.8) + 32)
```

## 1.10 Define the test method (and more)

While defining the method `testCelsiusToFahrenheit` using the class browser, the system will tell you that the class `TemperatureConverter` does not exist (This is true because we did not define it so far). The system will propose to create it. Just let the system do it.

Once you are done. You should have two classes: `TemperatureConverterTest` and `TemperatureConverter`. As well as one method: `testCelsiusToFahrenheit`. The test does not pass since we did not implement the conversion method (as shown by the red color in the body of `testCelsiusToFahrenheit`).

Note that you entered the method above and the system compiled it. Now in this book we want to make sure that you know about which method we are talking about hence we will prefix the method definitions with their class. For example the method `testCelsiusToFahrenheit` in the class `TemperaturLoggerTest` is defined as follows:

```
TemperaturLoggerTest >> testCelsiusToFahrenheit

    | converter |
    converter := TemperatureConverter new.
    self assert: (converter convertCelsius: 30) equals: 86.0
```

## 1.11 Converting from Fahrenheit to Celsius

Now you got the idea. Let us define a test for the conversion from Fahrenheit to Celsius.

```
TemperatureConverterTest >> testFahrenheitToCelsius

    | converter |
    converter := TemperatureConverter new.
    self assert: ((converter convertFarhenheit: 86) equals: 30.0).
    self assert: ((converter convertFarhenheit: 50) equals: 10)
```

Define the method `convertFarhenheit: anInteger`

```
TemperatureConverter >> convertFarhenheit: anInteger
    "Convert anInteger from fahrenheit to celsius"

    ^ ((anInteger - 32) / 1.8)
```

Run the tests they should all pass.

## 1.12 About floats

The conversions method we wrote returns floats. Floats are special objects in computer science because it is complex to represent infinite information (such as all the numbers between two consecutive integers) with a finite space (numbers are often represented with a fixed number of bits). In particular we should pay attention when comparing two floats. Here is a surprising case: we add two floats and the sum is not equal to their sums.

```
(0.1 + 0.2) = 0.3
> false
```

This is because the sum is not just equal to `0.3`. The sum is in fact the number `0.30000000000000004`

```
(0.1 + 0.2)
> 0.30000000000000004
```

To solve this problem in Pharo (it is the same in most programming languages), we do not use equality to compare floats but alternate messages such as `closeTo:` or `closeTo:precision:` as shown below:

```
(0.1 + 0.2) closeTo: 0.3
> true
(0.1 + 0.2) closeTo: 0.3 precision: 0.001
> true
```

To know more, you can have a look at the Fun with Float chapter in Deep Into Pharo (http://books.pharo.org)). The key point is that in computer science you should always avoid to compare the floats naively.

So let us go back to our conversion:

```
((52 - 32) / 1.8)
> 11.11111111111111
```

In the following expression we check that the result is close to 11.1 with a precision of 0.1. It means that we accept as result 11 or 11.1

```
((52 -  32) / 1.8) closeTo: 11.1 precision: 0.1
> true
```

We can use `closeTo:precision:` in our tests to make sure that we deal correctly with the float behavior we just described.

```
((52 -  32) / 1.8) closeTo: 11.1 precision: 0.1
> true
```

We change our tests to reflect this

```
TemperatureConverterTest >> testFahrenheitToCelsius

    | converter |
    converter := TemperatureConverter new.
    self assert: ((converter convertFarhenheit: 86) closeTo: 30.0
     precision: 0.1).
    self assert: ((converter convertFarhenheit: 50) closeTo: 10
     precision: 0.1)
```

## 1.13 Printing rounded results

The following expression shows that we may get converted temperature with a too verbose precision.

```
(TemperatureConverter new convertFarhenheit: 52)
>11.11111111111111
```

Here just getting 11.1 is enough. There is no need to get the full version. In fact, we can manipulate floats in full precision but there are case like User Interfaces where we would like to get a shorter sort of information. Typically as user of the temperature converter, our body does not see the difference between 12.1 or 12.2 degrees. Pharo libraries include the message `printShowingDecimalPlaces: aNumberOfDigit` used to rounds the *textual* representation of a float.

```
(TemperatureConverter new convertFarhenheit: 52)
    printShowingDecimalPlaces: 1
>11.1
```

## 1.14 Building a map of degrees

Often when you are travelling you would like to have kind of a map of different degrees as follows: Here we want to get the converted values between 50 to 70 farhenheit degrees.

```
(TemperatureConverter new convertFarhenheitFrom: 50 to: 70 by: 2).
> { 50->10.0.
    52->11.1.
    54->12.2.
    56->13.3.
    58->14.4.
    60->15.6.
    62->16.7.
    64->17.8.
    66->18.9.
    68->20.0.
    70->21.1}
```

What we see is that the method `convertFarhenheitFrom:to:by:` returns an array of pairs.

A pair is created using the message `->` and we can access the pair elements using the message `key` and `value` as shown below.

```
| p1 |
p1 := 50 -> 10.0.
p1 key
> 50
p1 value
> 10.0
```

Let us write a test first. We want to generate map containing as key the farhenheit and as value the converted celsius. Therefore we will get a collection with the map named `results` and a collection of the expected values that the value of the elements should have.

On the two last lines of the test method, using the message `with:do:` we iterate on both collections in parallel taking on element of each collection and compare them.

```
TemperatureConverterTest >> testFToCScale

    | converter results expectedCelsius |
    converter := TemperatureConverter new.
    results := (converter convertFarhenheitFrom: 50 to: 70 by: 2).
    expectedCelsius := #(10.0 11.1 12.2 13.3 14.4 15.5 16.6 17.7
    18.8 20.0 21.1).

    results with: expectedCelsius
        do: [ :res :cel | res value closeTo: cel ]
```

Now we are ready to implement the method `convertFarhenheitFrom: low to: high by: step`. Using the message `to:by:`, we create an interval to generate the collection of numbers starting at low and ending up at high using the increment step. Then we use the message `collect:` which applies a block to a collection and returns a collection containing all the values returned by the block application. Here we just create a pair whose key is the farhenheit and whose value is its converted celsius value.

```
TemperatureConverter >> convertFarhenheitFrom: low to: high by: step
    "Returns a collection of pairs (f, c) for all the farhenheit
    temperatures from a low to an high temperature"

    ^ (low to: high by: step)
        collect: [ :f | f -> (self convertFarhenheit: f) ]
```

## 1.15 Spelling Fahrenheit correctly!

You may not noticed but we badly spelled fahrenheit since the beginning of this chapter. Fahrenheit is not spelt Farhenheit but Fahrenheit. Now you may start to think that I'm crazy, because you should rename all the methods you wrote and in addition all the users of such methods and after we should check that we did not break anything. And you can think that this is a huge task.

Well first you should rename the methods because nobody wants to keep badly named code. Second, I'm not crazy at all. Programmers rename their code regularly because they often do not get it right the first time, or even the second time... Often you rewrite your code after thinking more about the interface you finally understand that you should propose. In fact good designer think a lot about names because names convey the intent of a computation. Now we have two super powerful tools to help us: Refactorings and Tests.

We will use the **Rename method** refactoring proposed by Pharo. A refactoring garantees that not only the method but all the places where it is called will also be renamed to send the new message. In addition a refactoring garantees that the behavior of the program is not modified. So this is really powerful.

Select the method `convertFarhenheit:` in the method list and bring the menu, use the **Rename method (all)** item, give a new name `convertFahrenheit:`. The system will prompt you to show you all the corresponding operations. Check them to see what you should have done manually. Imagine the amount of mistakes you could have made and proceed. Do the same for `convertFahrenheitFrom:to:by:`.

Now the key question is if these changes broke anything. Normally everything should work since this is what we expect when using refactorings. But runnning the tests has the final word. So run the tests to check if everything is ok and here is a clear use of tests: they ensure that we can spot fast a regression.

With this little scenario you should have learned two important things:

- Tests are written once and executed million times to check for regression.
- Refactorings are really powerful operations that save us from tedious manual rewriting.

## 1.16

We can add now a new method to convert fahrenheit to celcius and we define a new test first.

```
TemperatureConverterTest >> testLocationAndDateWithConversion

    | converter |
    converter := TemperatureConverter new location: 'Lille'.
    converter measureFahrenheit: 86.
    converter measureFahrenheit: 50.
    self assert: converter measureCount equals: 2.
    self assert: converter dates
        equals: {Date today . Date today} asOrderedCollection.
    self assert: converter temperatures
        equals: { converter convertFahrenheit: 86 .
                converter convertFahrenheit: 50 } asOrderedCollection
```

What we do is that since celsius is the scientific unity for temperature we convert to celsius before recording our temperature.

```
TemperatureConverter >> measureFahrenheit: aTemp
    measures add: DateAndTime now -> (self convertFahrenheit: aTemp)
```

## 1.17 Discussion

From a design perspective we see that the logger behavior is a much better object than the converter. The logger keeps some internal data specific to a location while the converter is stateless. Object-oriented programming is much better for capturing object with state. This is why the converter was a kind of silly objects but it was to get you started. Now it is rare that the world we want to model and represent is stateless. This is why object-oriented programming is a powerful way to develop complex programs.

## 1.18 Conclusion

In this chapter we built a simple temperature converter. We showed how define and execute unit tests using a Test Driven approach. The interest in testing and Test Driven Development is not limited to Pharo. Automated testing has become a hallmark of the *Agile software development* movement, and any software developer concerned with improving software quality would do well to adopt it.

We showed that tests are an important aid to measure our progress and also are an important aid to define clearly what we want to develop.

# Bibliography