

Pharo by Example 9.0

Stéphane Ducasse, Sebastijan Kaplar, Gordana Rakic and Quentin Ducasse

July 3, 2021

Copyright 2017 by Stéphane Ducasse, Sebastijan Kaplar, Gordana Rakic and Quentin Ducasse.

The contents of this book are protected under the Creative Commons Attribution-ShareAlike 3.0 Unported license.

You are **free**:

- to **Share**: to copy, distribute and transmit the work,
- to **Remix**: to adapt the work,

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page:

<http://creativecommons.org/licenses/by-sa/3.0/>

Any of the above conditions can be waived if you get permission from the copyright holder. Nothing in this license impairs or restricts the author's moral rights.



Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):

<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

Contents

Illustrations	ii
1 Developing a simple counter	1
1.1 Our use case	1
1.2 Create a package and class	2
1.3 Define protocols and methods	5
1.4 Create a method	5
1.5 Adding a setter method	7
1.6 Define a Test Class	7
1.7 Saving your code on git with Iceberg	8
1.8 Adding more messages	12
1.9 Instance initialization method	12
1.10 Define an initialize method	13
1.11 Define a new instance creation method	14
1.12 Better object description	15
1.13 Saving your code on a remote server	15
1.14 Conclusion	18
Bibliography	19

Illustrations

1-1	Package created and class creation template.	2
1-2	Class created: It inherits from <code>Object</code> class and has one instance variable named <code>count</code>	4
1-3	Counter class has now a comment! Well done.	4
1-4	The method editor selected and ready to define a method.	6
1-5	The method <code>count</code> defined in the protocol <i>accessing</i>	6
1-6	A first test is defined and it passes.	8
1-7	Iceberg <i>Repositories</i> browser on a fresh image indicates that if you want to version modifications to Pharo itself you will have to tell Iceberg where the Pharo clone is located. But you do not care.	9
1-8	Add and create a project named <code>MyCounter</code> and with the <code>src</code> subfolder. . .	10
1-9	Selecting the Add package iconic button, add your package <code>MyCounter</code> to your project.	10
1-10	Now Iceberg shows you that you did not commit your code.	11
1-11	Iceberg shows you the changes about to be committed.	11
1-12	Once you save your change, Iceberg shows you that	12
1-13	Class with more green tests.	13
1-14	Counter instance better description.	15
1-15	A <i>Repository</i> browser opened on your project.	16
1-16	GitHub HTTPS address our our project.	17
1-17	Using the GitHub HTTPS address.	17
1-18	Commits sent to the remote repository.	17

Developing a simple counter

To get started in Pharo, we invite you to implement a simple counter by following the steps given below. In this exercise you will learn how to create packages, classes, methods, and instances. You will learn how to define tests and more. This simple tutorial covers most of the important actions that we do when developing in Pharo. You can also watch the companion videos available in the Pharo mooc at <http://mooc.pharo.org>: they illustrate this tutorial in a more lively manner. Just pay attention that the videos are for Pharo 80 and some details are different in Pharo 90.

Note that the development flow promoted by this little tutorial is *traditional* in the sense that you will define a package, a class, *then* define its instance variable *then* define its methods *and* finally execute it. We show also how you can save your code with git hosting services such as github using Iceberg. Now in Pharo, developers often follows a *totally* different style that we call Xtreme TDD: they execute an expression that raises errors. Such errors are caught by the debugger and the developer can code directly in the debugger and let the system define some instance variables and methods on the fly for them.

Once you will have finish this tutorial, you will feel more confident with Pharo and we strongly suggest you to try the other style. Again there is a video showing this powerful alternate flow of coding.

1.1 Our use case

Here is our use case: We want to be able to create a counter, increment it twice, decrement it and check that its value is correct. It looks like this little use case will fit perfectly a unit test - you will define one later.

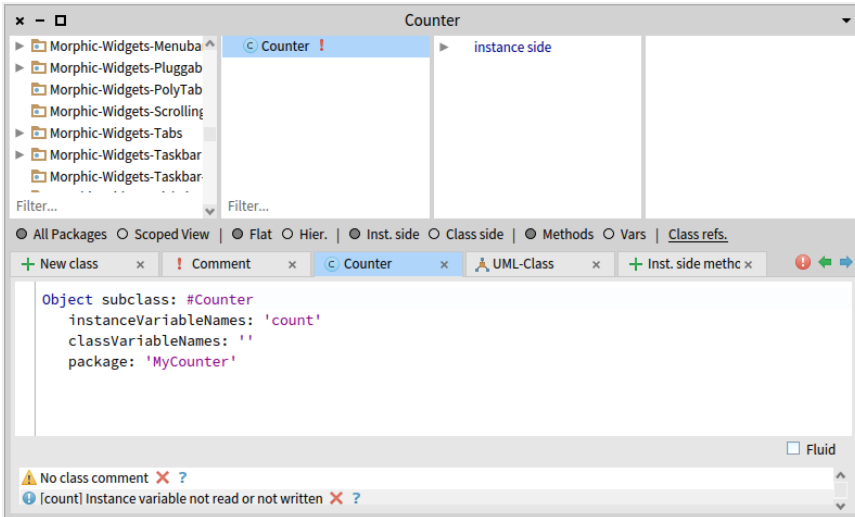


Figure 1-1 Package created and class creation template.

```
| counter |
counter := Counter new.
counter increment; increment.
counter decrement.
counter count = 1
```

Now we will develop all the mandatory class and methods to support this scenario.

In this part, you will create your first class. In Pharo, a class is defined in a package. You will create a package then a class. The steps we will do are the same ones every time you create a class, so memorize them well.

1.2 Create a package and class

Using the Browser create a package. The system will ask you a name, write `MyCounter`. This new package is then created and added to the list. Figure 1-1 shows the result of creating such a package.

Create a class.

Creating a class requires five steps. They consist basically in editing the class definition template to specify the class you want to create.

- **Superclass Specification.** First, you should replace the word `NameOf-Superclass` with the word `Object`. Thus, you specify the superclass of

1.2 Create a package and class

the class you are creating. Note that this is not always the case that Object is the superclass, since you may to inherit behavior from a class specializing already Object.

- **Class Name.** Next, you should fill in the name of your class by replacing the word NameOfClass with the word Counter. Take care that the name of the class starts with a capital letter and that you do not remove the #sign in front of NameOfClass. This is because the class we want to create does not exist yet, so we have to give its name, and we use a Symbol (a unique string in Pharo) to do so.
- **Instance Variable Specification.** Then, you should fill in the names of the instance variables of this class. We need one instance variable called count. Take care that you leave the string quotes!
- **Class Variable Specification.** As we do not need any class variable make sure that the argument for the class instance variables is an empty string classInstanceVariableNames: ''.

You should get the following class definition.

```
Object subclass: #Counter
  instanceVariableNames: 'count'
  classVariableNames: ''
  package: 'MyCounter'
```

Now we should compile it. We now have a filled-in class definition for the class Counter. To define it, we still have to *compile* it. Therefore, select the accept menu item. The class Counter is now compiled and immediately added to the system.

Figure 1-2 illustrates the resulting situation that the browser should show.

The tool runs automatically some code critic and some of them are just inaccurate, so do not care for now.

As we are disciplined developers, we add a comment to Counter class by clicking Comment pane and the Edit / View comment toggle. You can write the following comment:

```
`Counter` is a simple concrete class which supports incrementing and
  decrementing.
Its API is
- `decrement` and `increment`
- `count`
Its creation message is `startAt:`
```

The comment are written in Microdown a markdown dialect and they are nicely rendered.

Select menu item 'accept' to store this class comment in the class.

Figure 1-3 shows the class with its comment.

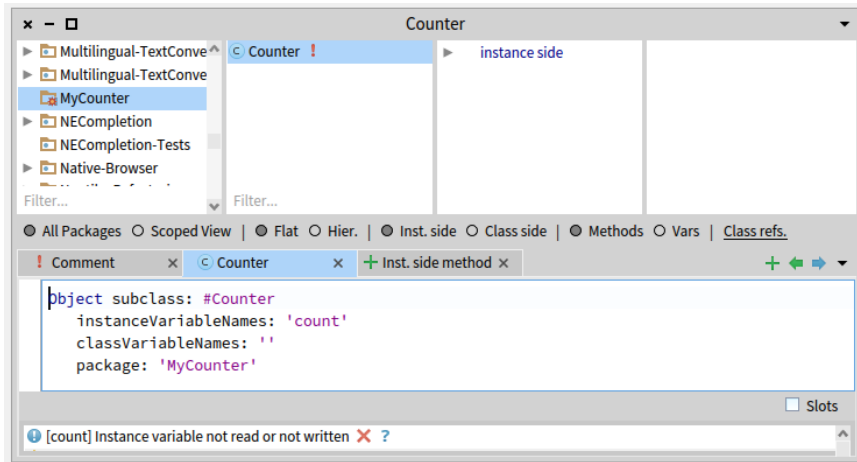


Figure 1-2 Class created: It inherits from Object class and has one instance variable named count.

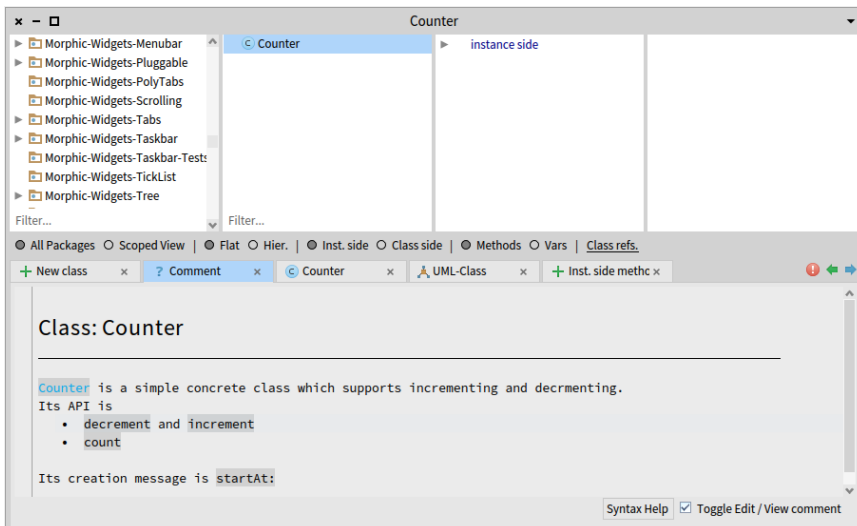


Figure 1-3 Counter class has now a comment! Well done.

1.3 Define protocols and methods

In this part you will use the browser to learn how to add protocols and methods.

The class we have defined has one instance variable named `count`. You should remember that in Pharo, (1) everything is an object, (2) that instance variables are private to the object, and (3) that the only way to interact with an object is by sending messages to it.

Therefore, there is no other mechanism to access the instance variable values from outside an object than sending a message to the object. What you can do is to define messages that return the value of the instance variable. Such methods are called *accessors*, and it is a common practice to always define and use them. We start to create an accessor method for our instance variable `count`.

A method is usually sorted into a protocol. These protocols are just a group of methods without any language semantics, but convey important navigation information for the reader of your class. Although protocols can have any name, Pharo programmers follow certain conventions for naming these protocols. If you define a method and are not sure what protocol it should be in, first go through existing code and try to find a fitting name.

1.4 Create a method

Now let us create the accessor methods for the instance variable `count`. Start by selecting the class `Counter` in a browser, and make sure the you are editing the instance side of the class (i.e., we define methods that will be sent to instances) by deselecting the Class side radio button.

Click on the instance method tab and define your method.

Figure 1-4 shows the method editor ready to define a method.

As a general hint, double click at the end of or beginning of the text and start typing your method: this automatically replace your Replace the template with the following method definition:

```
[ count
  ^ count
```

This defines a method called `count`, taking no arguments, having a method comment and returning the instance variable `count`. Then choose *accept* in the menu to compile the method. The method is automatically categorized in the protocol *accessing*.

Figure 1-5 shows the state of the system once the method is defined.

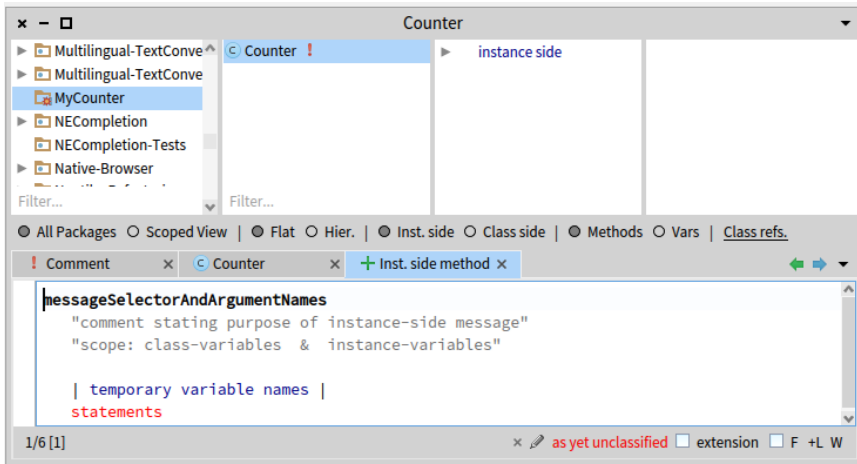


Figure 1-4 The method editor selected and ready to define a method.

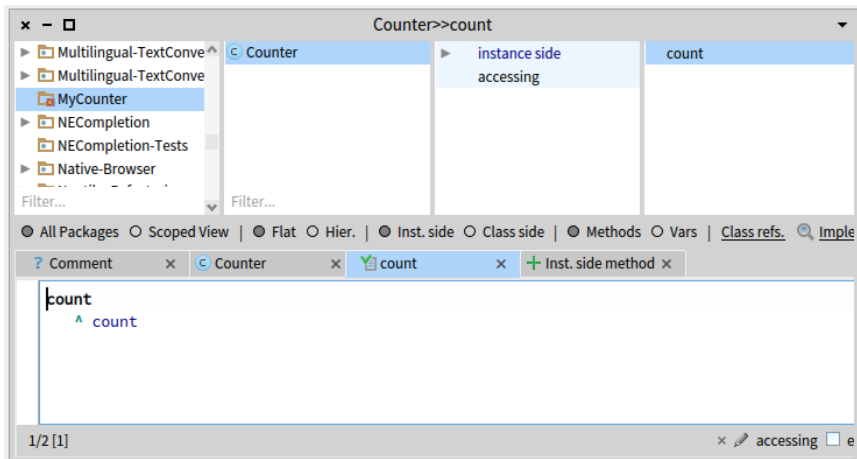


Figure 1-5 The method `count` defined in the protocol `accessing`.

You can now test your new method by typing and evaluating the next expression in a Playground, or any text editor.

```
Counter new count  
>>> nil
```

This expression first creates a new instance of `Counter`, and then sends the message `count` to it. It retrieves the current value of the counter. This should return `nil` (the default value for non-initialised instance variables). Afterwards we will create instances with a reasonable default initialisation value.

1.5 Adding a setter method

Another method that is normally used besides the accessor method is a so-called setter method. Such a method is used to change the value of an instance variable from a client. For example, the expression `Counter new count: 7` first creates a new `Counter` instance and then sets its value to 7:

The snippets shows that the counter effectively contains its value.

```
| c |  
c := Counter new count: 7.  
c count  
>>> 7
```

This setter method does not currently exist, so as an exercise write the method `count:` such that, when invoked on an instance of `Counter`, instance variable is set to the argument given to the message. Test your method by typing and evaluating the expression above.

1.6 Define a Test Class

Writing tests is an important activity that will support the evolution of your application. Remember that a test is written *once and executed million* times. For example if we have turned the expression above into a test we could have checked automatically that our new method is correctly working.

To define a test case we will define a class that inherits from `TestCase`. Therefore define a class named `CounterTest` as follows:

```
TestCase subclass: #CounterTest  
  instanceVariableNames: ''  
  classVariableNames: ''  
  package: 'MyCounter'
```

Now we can write a first test by defining one method. Test methods should start with *text* to be automatically executed by the `TestRunner` or when you press on the icon of the method. Now to make sure that you understand in which class we define the method we prefix the method body with the class



Figure 1-6 A first test is defined and it passes.

name and >>. CounterTest>> means that the method is defined in the class CounterTest.

Figure 1-6 shows the definition of the method testCountIsSetAndRead in the class CounterTest.

Define the following method. It first creates an instance, sets its value and verifies that the value is correct. The message `assert:equals:` is a special message verifying if the test passed or not.

```

CounterTest >> testCountIsSetAndRead
| c |
c := Counter new.
c count: 7.
self assert: c count equals: 7

```

Verify that the test passes by executing either pressing the icon in front of the method (as shown by Figure 1-6) or using the TestRunner available in the Tools menus (selecting your package). Since you have a first green test. This is a good moment to save your work.

1.7 Saving your code on git with Iceberg

With Iceberg, we will show you how to save your code locally then later we will push it to GitHub.

1.7 Saving your code on git with Iceberg

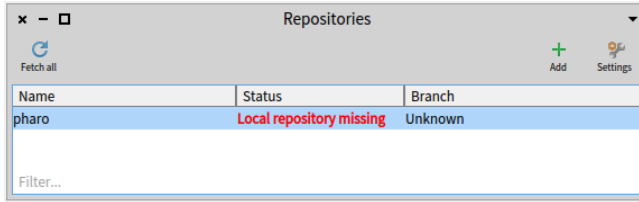


Figure 1-7 Iceberg *Repositories* browser on a fresh image indicates that if you want to version modifications to Pharo itself you will have to tell Iceberg where the Pharo clone is located. But you do not care.

Open Iceberg.

You should the situation depicted by Figure 1-7 which shows the top level Iceberg pane. It shows that for now you do not have defined nor loaded any project. It shows the Pharo project and indicates that it could not find its local repository by displaying 'Local repository missing'. You do not have to worry about the Pharo project or repository if you do not want to contribute to Pharo. So just go ahead. Since you do not plan to modify and version the Pharo system code, you do not have to worry.

Add and configure a project.

Press the iconic button Add to create a new project. Pick up 'New Repository' and you should get a configuration pane similar to the one of Figure 1-8. Here we define the Project named 'MyCounter', give a directory on our disk and we indicate that the source should be in the subfolder src.

Add your package to the project.

Once added, Iceberg *Working copy* browser should show you an empty pane because you did not add any package to your project. Click on the Add package iconic button and select the package MyCounter as shown by Figure 1-9.

Commit your changes.

Once you package is added, Iceberg shows you that you did not commit your code as shown in Figure 1-11. Press the Commit iconic button. Iceberg will show you all the changes that are about to be saved (Figure 1-11). Enter a commit message and commit

Code saved.

Once you have committed, Iceberg indicates that your system and local repository are in sync.

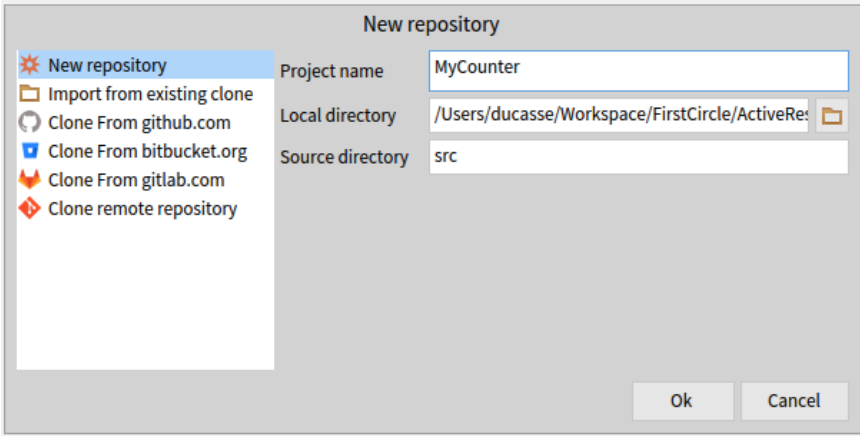


Figure 1-8 Add and create a project named MyCounter and with the src subfolder.

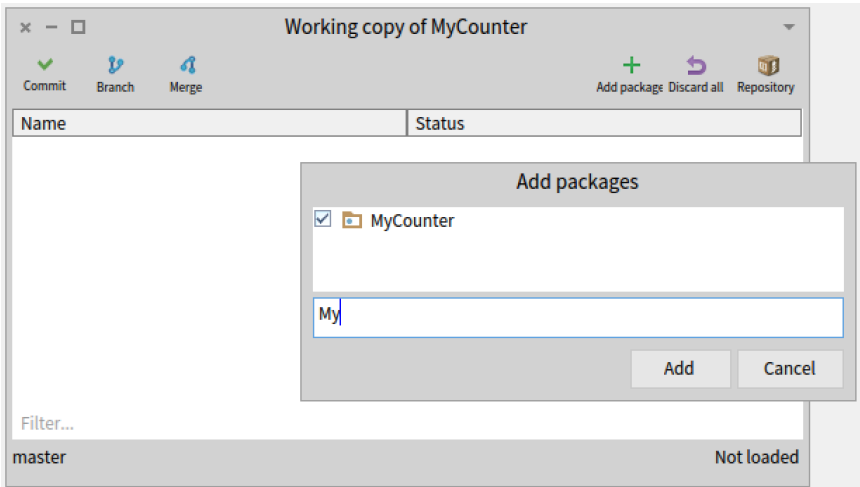


Figure 1-9 Selecting the Add package iconic button, add your package MyCounter to your project.

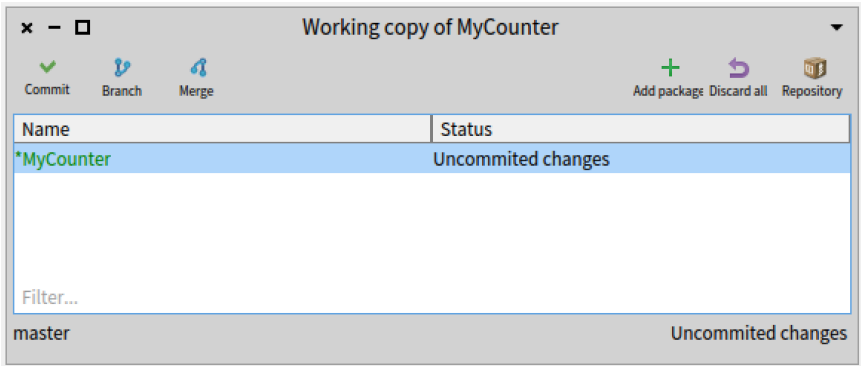


Figure 1-10 Now Iceberg shows you that you did not commit your code.

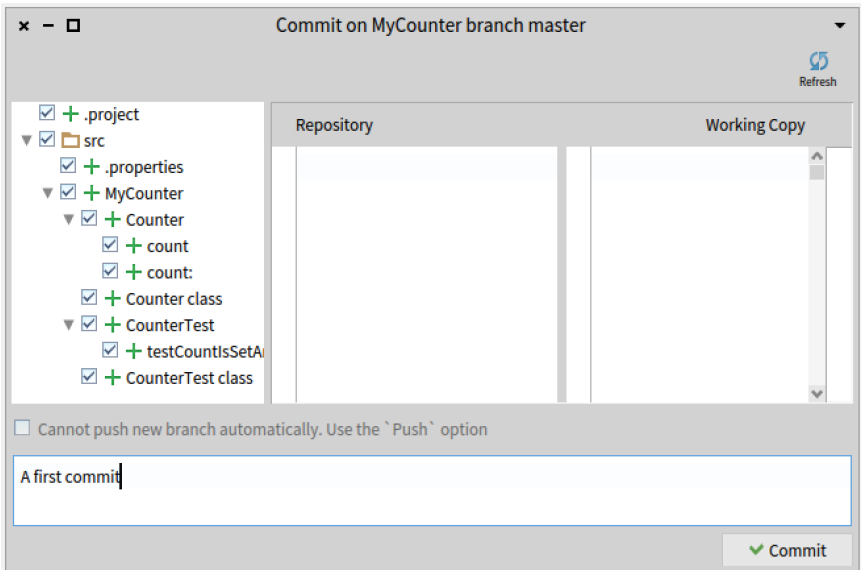


Figure 1-11 Iceberg shows you the changes about to be committed.

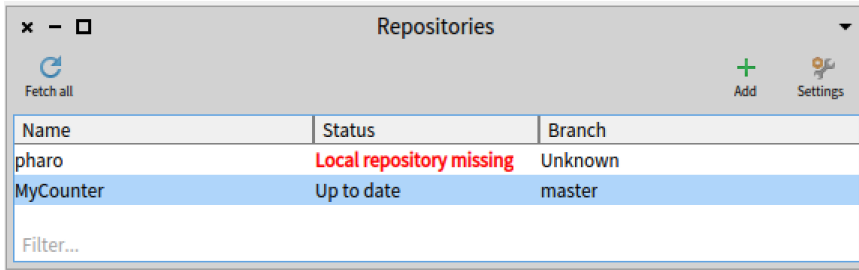


Figure 1-12 Once you save your change, Iceberg shows you that .

1.8 Adding more messages

Before implementing the following messages we define first a test. We define one test for the method `increment` as follows:

```
CounterTest >> testIncrement
| c |
c := Counter new.
c count: 0 ; increment; increment.
self assert: c count equals: 2
```

- Propose a definition for the method `increment`.
- Define a test and method for the method `decrement`.
- Implement the following methods `increment` and `decrement` in the protocol `'operation'`.
- Implement also a new test method for the method `decrement`.

```
Counter >> increment
count := count + 1
```

```
Counter >> decrement
count := count - 1
```

Run your tests they should pass (as shown in Figure 1-13). Again this is a good moment to save your work. Saving at point where tests are green is always a good process. To save your changes, you just have to commit them.

1.9 Instance initialization method

Right now the initial value of our counter is not set as the following expression shows it.

```
Counter new count
>>> nil
```


1.10 Define an initialize method

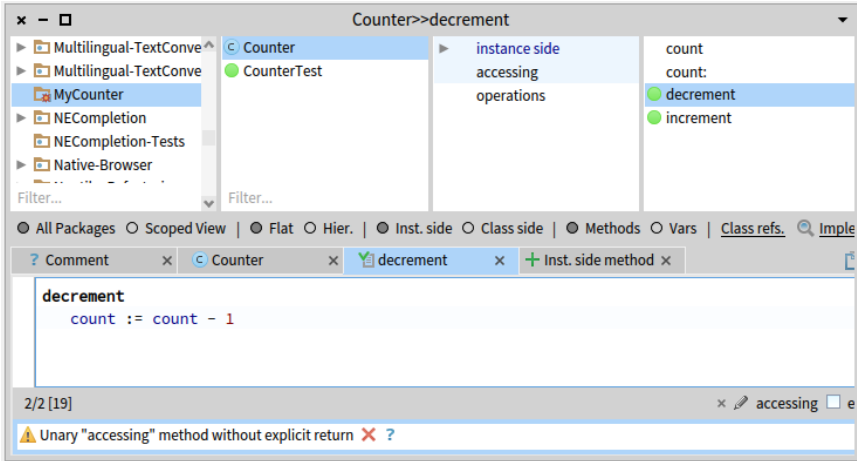


Figure 1-13 Class with more green tests.

Let us write a test checking that a newly created instance has 0 as a default value.

```
CounterTest >> testInitialize
  self assert: Counter new count equals: 0
```

If you run it, it will turn yellow indicating a failure (a situation that you anticipated but that is not correct) - by opposition to an error which is an anticipated situation leading to failed assertion.

1.10 Define an initialize method

Now we have to write an initialization method that sets a default value of the count instance variable. However, as we mentioned the initialize message is sent to the newly created instance. This means that the initialize method should be defined at the instance side as any method that is sent to an instance of Counter (like increment) and decrement. The initialize method is responsible to set up the instance variable default values.

Therefore at the instance side, you should create a protocol initialization, and create the following method (the body of this method is left blank. Fill it in!).

```
Counter >> initialize
  "set the initial value of the value to 0"
  ...
  Fill me please!!!
```

Now create a new instance of class Counter. Is it initialized by default? The following code should now work without problem:

```
[ Counter new increment count
>>> 1
```

and the following one should return 2

```
[ Counter new increment; increment; count
>>> 2
```

But better write a test since we will execute it all the time.

```
[ TestCounter >> testCounterWellInitialized
  self
    assert: (Counter new increment; increment; count)
      equals: 2
```

Again save your work before starting the next step.

1.11 Define a new instance creation method

We would like to show you the difference between an instance method (i.e. sent to instances) and a class method (i.e., to a class). In fact the only difference is the place to define them. An instance method is defined in the instance side of Code Browser while class methods are defined on the class side – pressing the **Class** button.

Define a different instance creation method named `startingAt:`. This method receives an integer as argument and returns an instance of Counter with the specified value.

Let us define a test:

```
[ TestCounter >> testCounterStartingAt5
  self assert: (Counter startingAt: 5) count equals: 5
```

Here the message `startingAt:` is sent to the class Counter itself.

Your implementation should look like

```
[ Counter class >> startingAt: anInteger
  ^ self new count: anInteger.
```

Note that `self` in such method refers to the class Counter itself.

Let us write another test to check that everything is working.

```
[ CounterTest >> testAlternateCreationMethod
  self assert: ((Counter startingAt: 19) increment ; count) equals:
    20
```

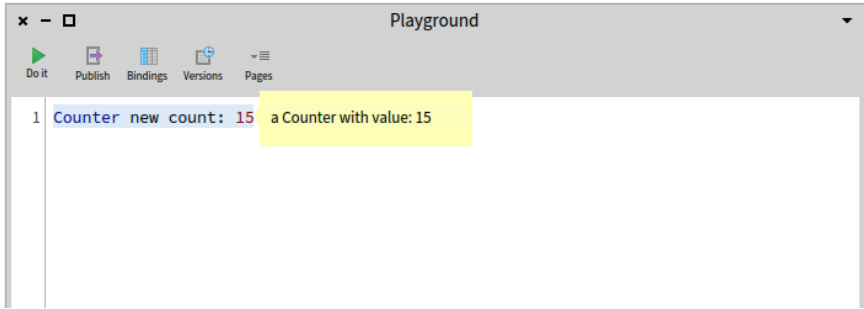


Figure 1-14 Counter instance better description.

1.12 Better object description

When you open an inspect (putting a `self halt` inside a method definition) you obtain an inspector or when you select the expression `Counter new` and print its result (using the Print it menu of the editor) you obtain a simple string 'a Counter'.

```
[ Counter new
  >>> a Counter
```

We would like to get a much richer information for example knowing the counter value. Implement the following methods in the protocol printing

```
[ Counter >> printOn: aStream
  super printOn: aStream.
  aStream nextPutAll: ' with value: ', count printString.
```

Note that the method `printOn:` is used when you print an object using `print it` (See Figure 1-14) or click on `self` in an inspector.

We let you define a method for this method. A tip send the message `printString` to `Counter new` to get its string representation.

```
[ Counter new printString
  >>> a Counter with value: 0
```

1.13 Saving your code on a remote server

Up until now you saved your code on your local disc. We will now show how you can save your code on a remote repository such as the one you can create on GitHub <http://github.com> or Gitlab.

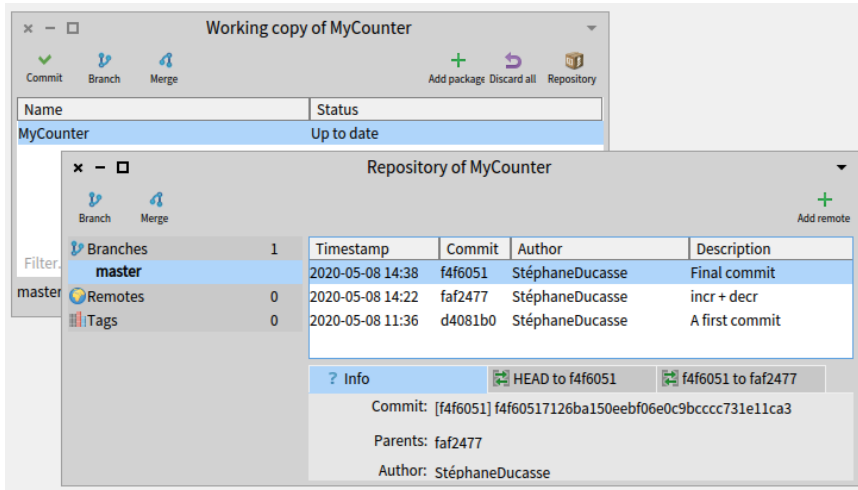


Figure 1-15 A *Repository* browser opened on your project.

Create a project on the remote server.

First you should create a project with the same name than the one of your project.

Add a remote repository in HTTPS access.

Clicking on the *Repository* iconic button of the *Working copy* browser, you get access to the *Repository* browser open on your project as show in Figure 1-15.

Then you just have to add a remote repository using the *Add remote* iconic button of the *Repository* browser. For this we will use the project identification address given by the remote browser. Since we decided to use HTTPS we use `https://github.com/Ducasse/MyCounter.git` as address as shown in Figure 1-16 and Figure 1-17.

Push.

As soon as you add a valid server address, Iceberg will show a little red indication on the *Push* iconic button. This shows that you have changes in your local repository that have not being pushed to your remote repository. Now you just have to press the *Push* iconic button. Iceberg will show you the commits that will be pushed to the server as shown in Figure 1-18.

Now you fully saved your code and you will be able to reload from another machine or location. This will enable you to work remotely and collaboratively.

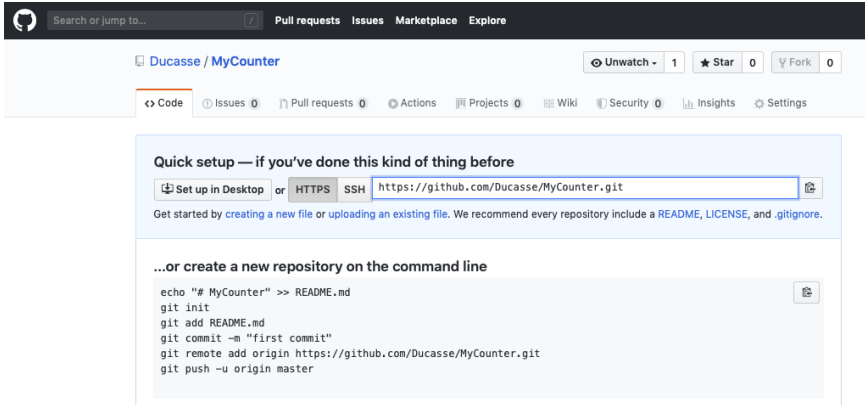


Figure 1-16 GitHub HTTPS address our our project.

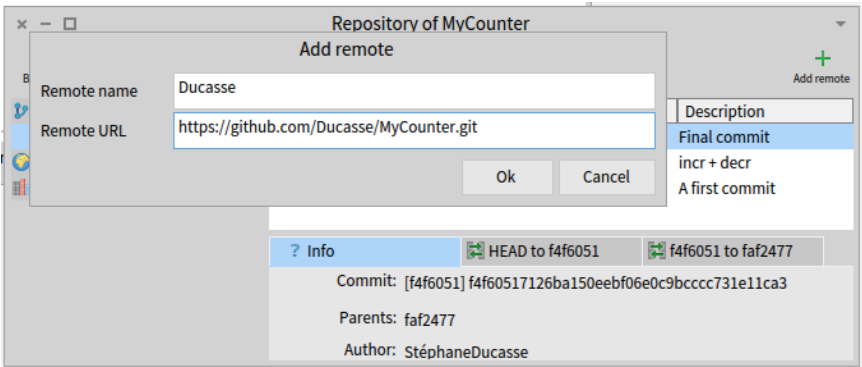


Figure 1-17 Using the GitHub HTTPS address.

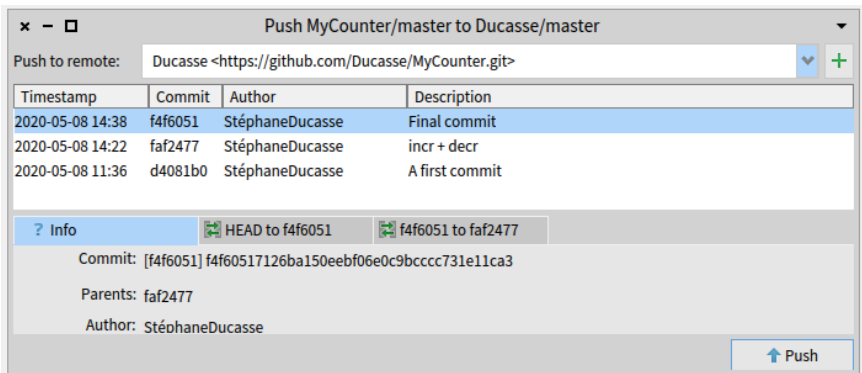


Figure 1-18 Commits sent to the remote repository.

1.14 Conclusion

In this tutorial you learned how to define packages, classes, methods, and define tests. The flow of programming that we chose for this first tutorial is similar to most of programming languages. However in Pharo, smart and agile developers use a different flow: It is based on defining a test first, executing it and when the execution raises an error to define the corresponding classes, methods, and instance variable often from inside the debugger. This is a super powerful flow because you get all the mandatory information and this is a productivity boost.

We suggest you now to redo the exercise by defining a test first, executing it and defining method in the debugger. Watch the second counter video of the Pharo mooc available at <http://mooc.pharo.org> to understand the flow.

Bibliography

