CHAPTER 1

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, method, 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 that illustrates this tutorial in a more lively manner.

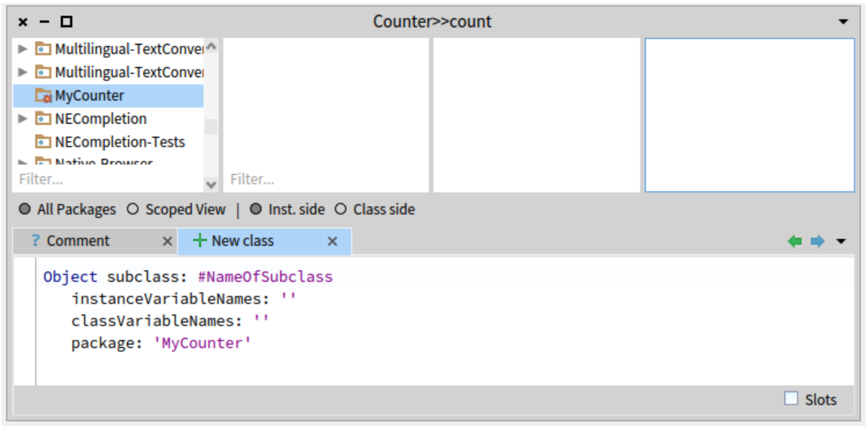
Note that the development flow promoted by this little tutorial is *traditional* in the sense that you will define a package, a class, *then* define its instance variable *then* define its methods *and* finally execute it. We show also how you can save your code on git hosting services such as github using Iceberg. The companion video follows also such programming development flow. Now in Pharo, developers often follows a *totally* different style (that we call live coding or Xtreme TDD) where they execute an expression that raises errors and they code 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 by following the second video showing such different development practices.

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.

```
| counter |
counter := Counter new.
```



```
Object subclass: #NameOfSubclass
	instanceVariableNames: ''
	classVariableNames: ''
	package: 'MyCounter'
```

Figure 1.1 Package created and class creation template.

```
counter increment; increment.
counter decrement.
counter count = 1
```

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

1.2 Create your own class

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.

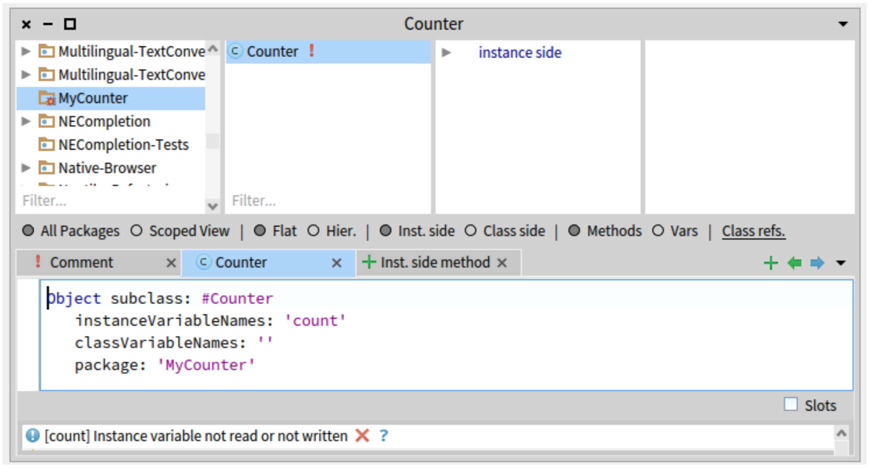
Create a package.

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 `NameOfSuperclass` with the word `Object`. Thus, you specify the superclass of 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`.



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

[count] Instance variable not read or not written

Figure 1.2 Class created: It inherits from `Object` class and has one instance variable named `count`.

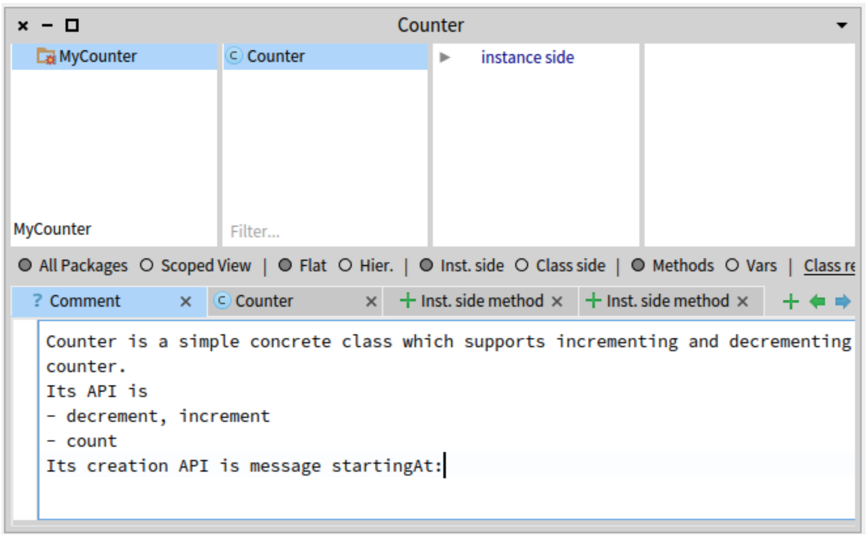
- Class Name. Next, you should fill in the name of your class by replacing the word `NameOfClass` with the word `Counter`. Take care that the name of the class starts with a capital letter and that you do not remove the #sign in front of `NameOfClass`. This is because the class we want to create does 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.



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

Figure 1.3 Counter class has now a comment! Well done.

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 button. You can write the following comment:

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

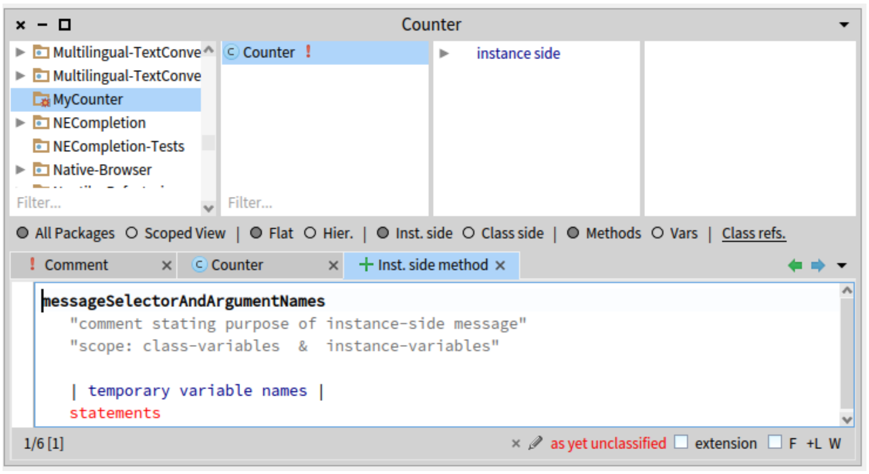
Select menu item 'accept' to store this class comment in the class.

Figure 1.3 shows the class with its comment.

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.



```
messageSelectorAndArgumentNames
    "comment stating purpose of instance-side message"
    "scope: class-variables  &  instance-variables"

    | temporary variable names |
    statements
```

Figure 1.4 The method editor selected and ready to define a method.

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.

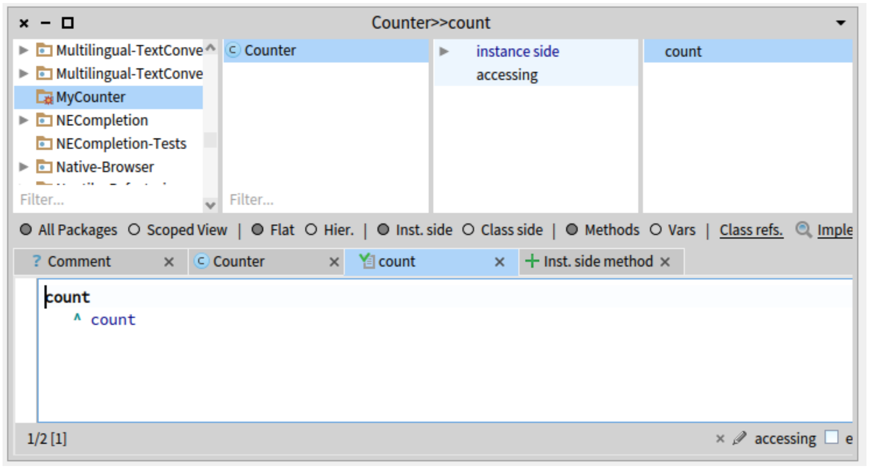
Create a method.

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
```

Figure 1.5 The method `count` defined in the protocol *accessing*.

```
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.

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.

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.

1.4 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: 'Counter'
```

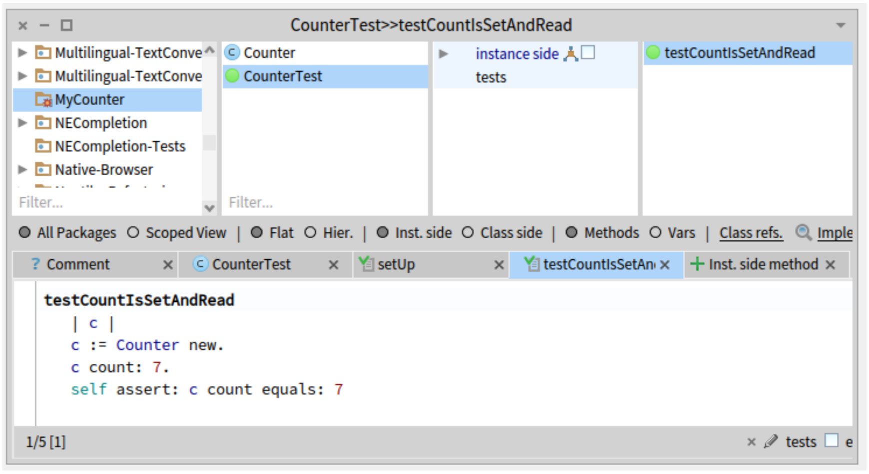
Now we can write a first test by defining one method. Test methods should start with *text* to be automatically executed by the TestRunner or when you press on the icon of the method. Now to make sure that you understand in which class we define the method we prefix the method body with the class name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

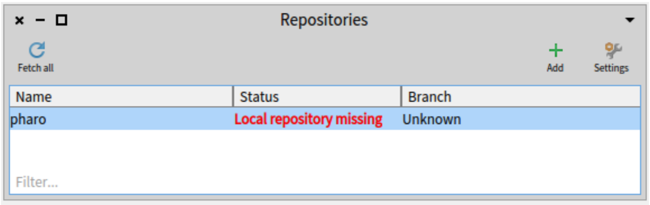
```
CounterTest >> testCountIsSetAndRead
   | c |
   c := Counter new.
   c count: 7.
   self assert: c count equals: 7
```

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.



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

Figure 1.6 A first test is defined and it passes.



Name	Status	Branch
pharo	Local repository missing	Unknown

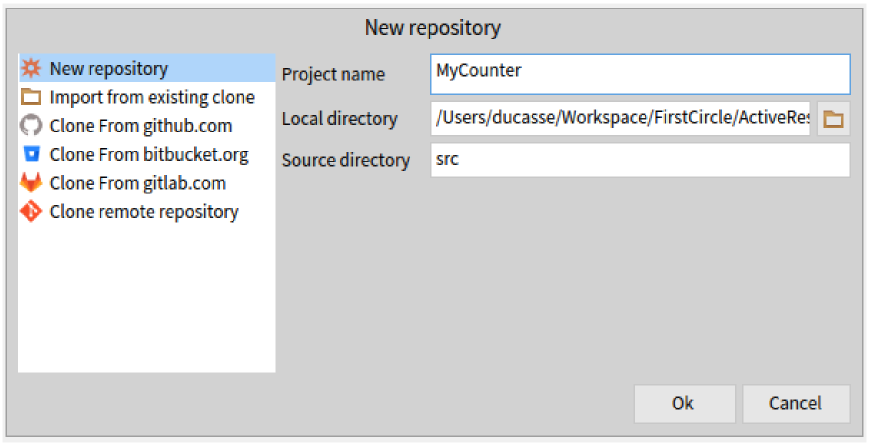
Figure 1.7 Iceberg *Repositories* browser on a fresh image indicates that if you want to version modifications to Pharo itself you will have to tell Iceberg where the Pharo clone is located. But you do not care.

1.5 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.

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.


New repository

- New repository
- Import from existing clone
- Clone From github.com
- Clone From bitbucket.org
- Clone From gitlab.com
- Clone remote repository

Project name: MyCounter
Local directory: /Users/ducasse/Workspace/FirstCircle/ActiveRe:
Source directory: src

Figure 1.8 Add and create a project named MyCounter and with the `src` subfolder.

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 anmd commit

Code saved.

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

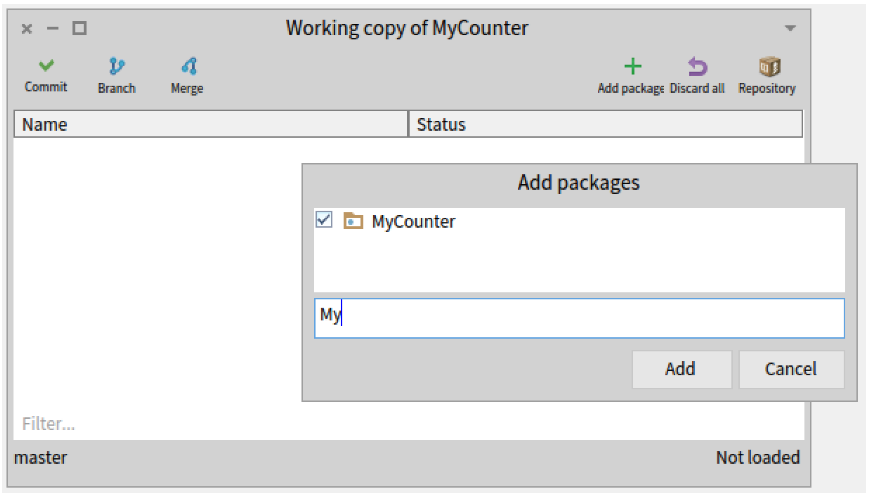


Figure 1.9 Selecting the Add package iconic button, add your package My-Counter to your project.

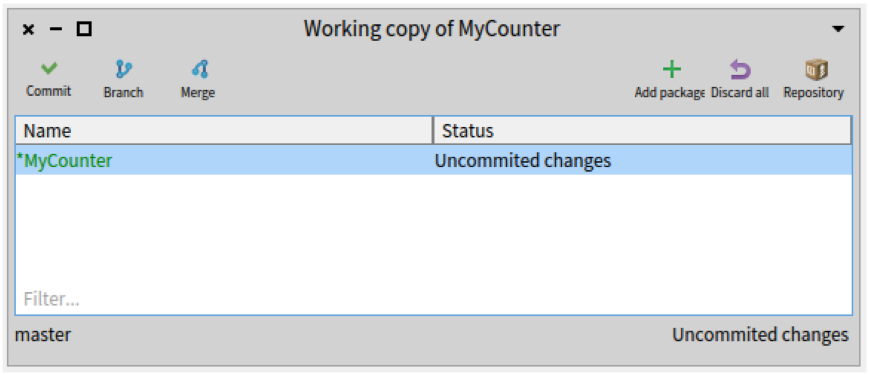
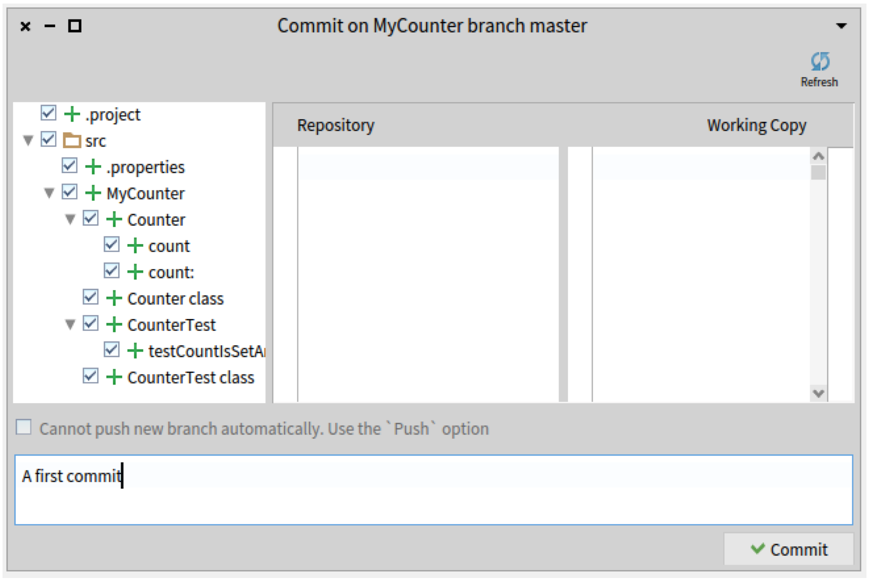


Figure 1.10 Now Iceberg shows you that you did not commit your code.


Commit on MyCounter branch master

Refresh

- .project
- src
  - .properties
  - MyCounter
    - Counter
      - count
      - count:
    - Counter class
    - CounterTest
      - testCountIsSetA
    - CounterTest class

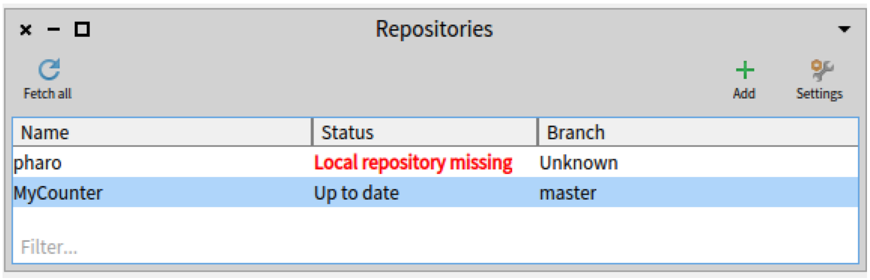
Repository	Working Copy

Cannot push new branch automatically. Use the `Push` option

A first commit

Commit

Figure 1.11 Iceberg shows you the changes about to be commited.

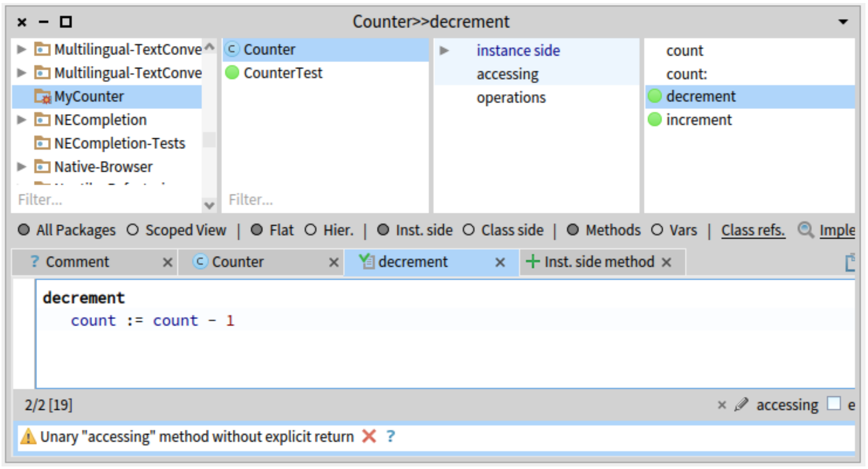

Repositories

Fetch all Add Settings

Name	Status	Branch
pharo	Local repository missing	Unknown
MyCounter	Up to date	master

Filter...

Figure 1.12 Once you save your change, Iceberg shows you that .



```
decrement
    count := count - 1
```

Unary "accessing" method without explicit return

Figure 1.13 Class with more green tests.

1.6 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.

1.7 Instance initialization method

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

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

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.

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.

1.8 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 button Class).

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
```

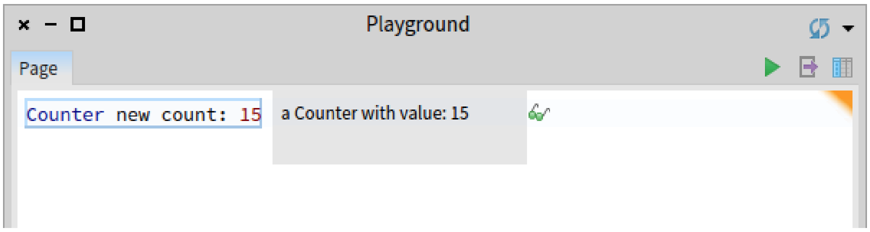
1.9 Better object description

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.
```



```
Counter new count: 15
```

a Counter with value: 15

Figure 1.14 Better description.

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
```

1.10 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.

Create a project on the remote server.

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

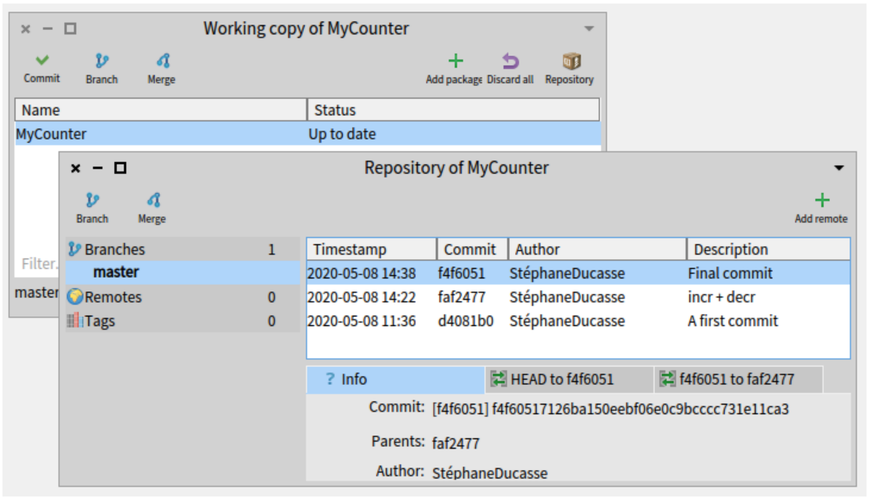
Add a remote repository in HTTPS access

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



Name	Status
MyCounter	Up to date

Branches	1
master	
Remotes	0
Tags	0

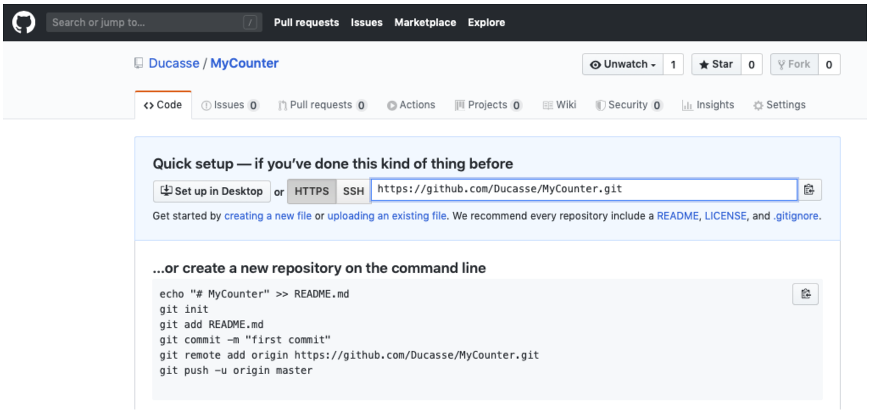
Timestamp	Commit	Author	Description
2020-05-08 14:38	f4f6051	StéphaneDucasse	Final commit
2020-05-08 14:22	faf2477	StéphaneDucasse	incr + decr
2020-05-08 11:36	d4081b0	StéphaneDucasse	A first commit

Commit: [f4f6051] f4f60517126ba150eebf06e0c9bcccc731e11ca3

Parents: faf2477

Author: StéphaneDucasse

Figure 1.15 A *Repository* browser opened on your project.



Quick setup — if you've done this kind of thing before

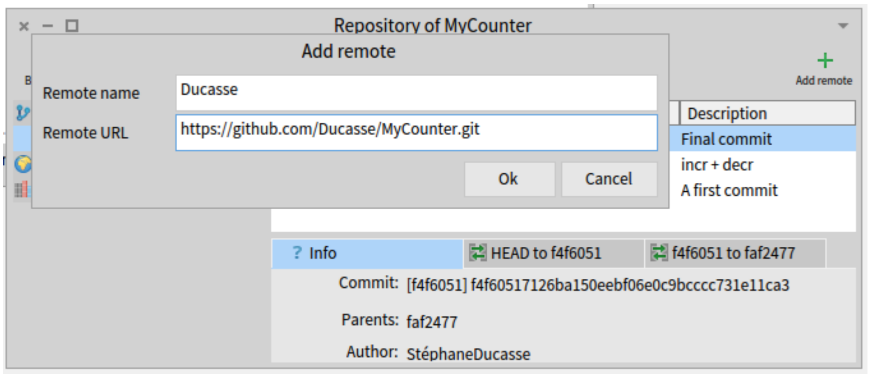
https://github.com/Ducasse/MyCounter.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# MyCounter" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/Ducasse/MyCounter.git
git push -u origin master
```

Figure 1.16 GitHub HTTPS address our our project.



Repository of MyCounter

Add remote

Remote name: Ducasse

Remote URL: https://github.com/Ducasse/MyCounter.git

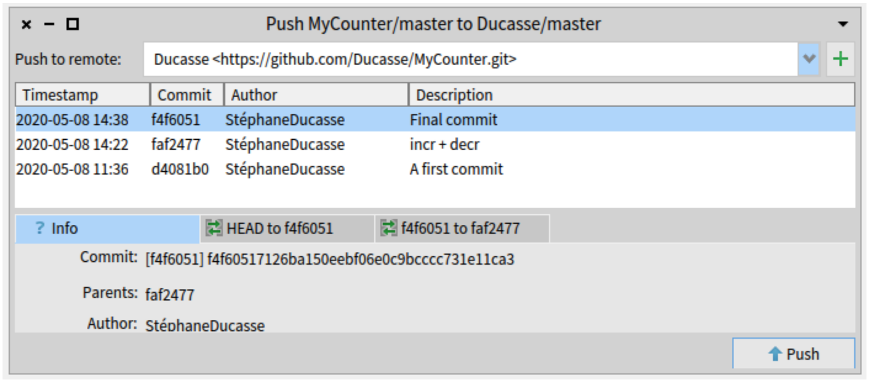
Description
Final commit
incr + decr
A first commit

Commit: [f4f6051] f4f60517126ba150eebf06e0c9bcccc731e11ca3

Parents: faf2477

Author: StéphaneDucasse

Figure 1.17 Using the GitHub HTTPS address.



Push MyCounter/master to Ducasse/master

Push to remote: Ducasse <https://github.com/Ducasse/MyCounter.git>

Timestamp	Commit	Author	Description
2020-05-08 14:38	f4f6051	StéphaneDucasse	Final commit
2020-05-08 14:22	faf2477	StéphaneDucasse	incr + decr
2020-05-08 11:36	d4081b0	StéphaneDucasse	A first commit

Commit: [f4f6051] f4f60517126ba150eebf06e0c9bcccc731e11ca3

Parents: faf2477

Author: StéphaneDucasse

Figure 1.18 Commits sent to the remote repository.

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 collaborately.

1.11 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. In Pharo you can use a different flow that is based on defining a test first, executing it and when the execution raises error to define the corresponding classes, methods, and instance variable often from inside the debugger. We suggest you now to redo the exercise following the second companion video.