Chapter 1

Developing a simple counter

We want you to implement a simple counter by following the steps given below. In this exercise you will learn how to create classes, method, instances. You will learn how to define tests and many more.

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. The companion video follows also such programming development flow.

Now in Pharo, developers often follows a totally different style (that we call more live coding) where they execute an expression that will raise errors and they will code in the debugger or let the system define some instance variables and methods on the fly for them. Once you will have finish this tutorial we will ask you to try the other style by following a video showing such different development practice.

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.

```
| 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.

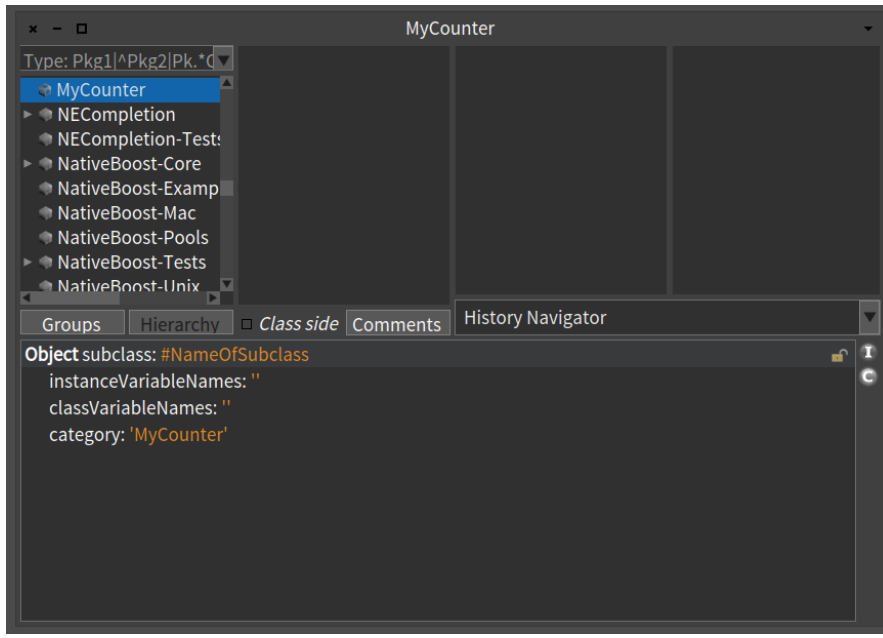

Figure 1.1: Package created.

1.2 Create your own class

In this part you will create your first class. In Pharo a class is defined in a package. The steps we will do are the same ones every time you create a class, so memorize them well. We are going to create a class `Counter` in a package called `MyCounter`. Figure 1.1 shows the result of creating such a package.

Create a package

In the Browser create a package. The system will ask you a name, write `Counter`. open This new package will be created and added to the list.

Create a class

Creating a class requires five steps. They consist basically of 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`.

- 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`

- 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: ''`.

- Compile. That's 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. The class `Counter` is now compiled and immediately added to the system.

You should get the following class definition.

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

Figure 1.2 illustrates the resulting situation that the browser should show.

As we are disciplined developers, we provide a comment to `Counter` class by clicking Comment button. You can write the following comment:

```
Counter is a concrete class which supports incrementing and decrementing a
    counter.
It knows how to increment and decrement its value.
```

Select accept to store this class comment in the class.

1.3 Define protocols and methods

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

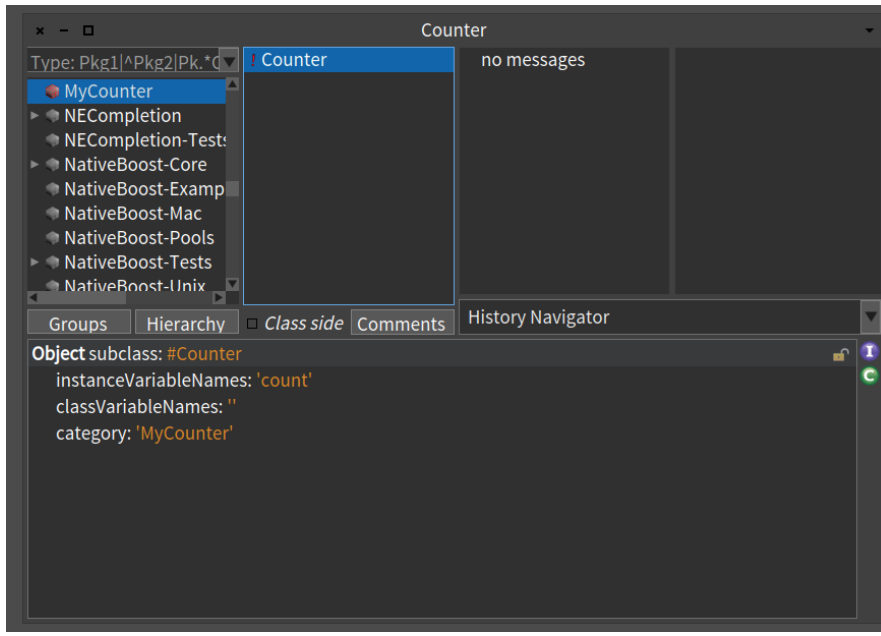

Figure 1.2: Class created.

The class we have defined has one instance variable named `count`. You should remember that in Pharo, everything is an object, that instance variables are private to the object and 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 variables 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 of a class. 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`.

Remember that every method belongs to 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 `value`. 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 radiobutton.

Create a new protocol by bringing the menu of methods protocol list. Select the newly created protocol. Then in the bottom pane, the edit field displays a method template laying out the default structure of a method. Replace the template with the following method definition:

```
count
   "return the current value of the value instance variable"
   ^ 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 operate menu to compile the method. 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 and retrieves the current value of the counter. This should return `nil` (the default value for noninitialised instance variables; afterwards we will create instances with a reasonable default initialisation value.

Remarks

Accessors can be defined in protocols accessing or private. Use the accessing protocol when a client object (like an interface) really needs to access your data. Use private to clearly state that no client should use the accessor. This is purely a convention. There is no simple way in Pharo (as in many dynamically-typed languages) to enforce access rights like private in C++ or Java. Note that this discussion does not seem to be very important in the context of this specific simple example. However, this question is central to the notion of object and encapsulation of the data. An important side effect of this discussion is that you should always ask yourself when you, as a client of an object, are using an accessor if the object is really well defined and if it does not need extra functionality.

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. 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: ''
    category: 'Counter'
```

Now we can write a first test by defining one method. Test methods should start with *text* to be automatically executed by the test runner 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`.

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

```
CounterTest>>testCountIsSetAndRead
    | c |
    c := Counter new.
    c count: 15.
```

```
self assert: c count = 15
```

Verify that the test passes by executing either pressing the icon in front of the method or using the test runner (selecting your package).

1.5 Adding more messages

Before implementing the following messages we will 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 = 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`.

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

```
Counter>>decrement
  self count: self count − 1
```

Run your tests they should pass (as shown in Figure 1.3).

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`

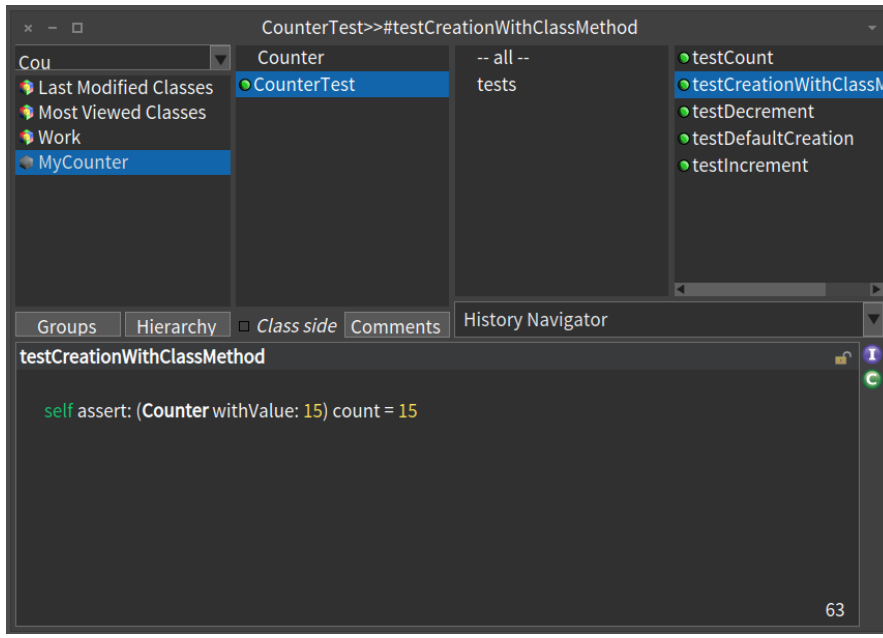

Figure 1.3: Class with green tests.

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

Note that the method `printOn:` is used when you print an object using print it (See Figure 1.4) or click on `self` in an inspector.

1.6 Instance initialization method

Right now the initial value of our counter is not set.

```
    Counter new count
    > nil
```

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

```
Counter>>testValueAtCreationTimeIsZero
```

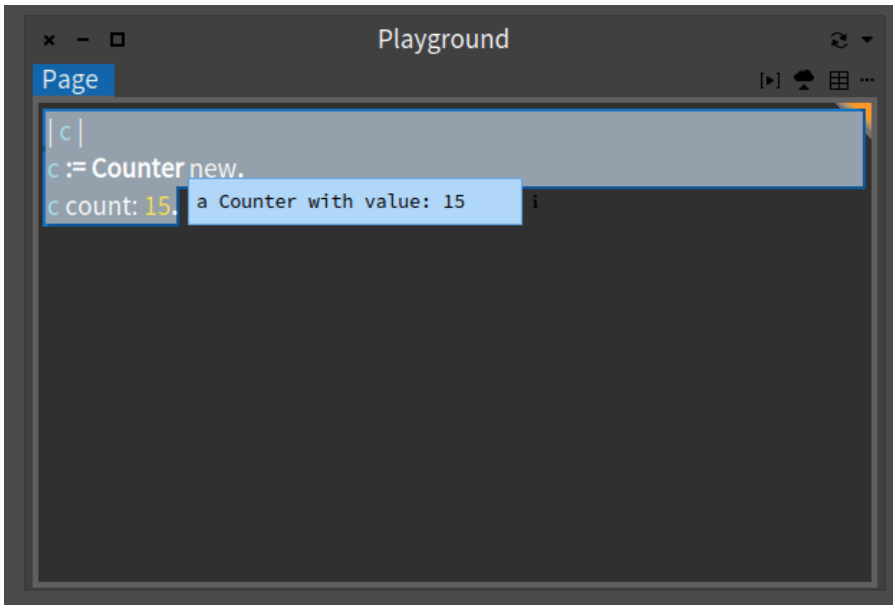

Figure 1.4: Better description.

```
self assert: Counter new count = 0
```

- Define the new test method.

Define 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"
```

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

```
Counter new increment
```

and the following one should return 2

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

1.7 Define a new instance creation method

To be sure that you have really understood the distinction between instance and class methods, you should now define a different instance creation method named `withValue:`. This method receives an integer as argument and returns an instance of `Counter` with the specified value.

Let us define a test:

```
CounterTest>>testAlternateCreationMethod
  self assert: ((Counter withValue: 19) increment ; count) = 20
```

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

1.8 Saving your Work

Several ways to save your work exist: You can

- Save the class by clicking on it and selecting the fileout menu item. You will get a file containing the source code on your harddisc - This is not the favorite way to save your code.
- Use the Monticello browser to save a package.

Use SmalltalkHub to save your work. Go to http://www.smalltalkhub.com/ and create a member account then register a new project. You get then an HTTP entry that refer to your project. Enter it as an HTTP repository in Monticello.

To save your work, simply publish your package. This will open a dialog where you can give a comment, version numbers and blessing. From then on, other people can load it from there, in the same way that you would use cvs or other multi-user versioning systems. Saving the image is also a way to save your working environment, but publishing it is better.

You can of course both publish your package (so that other people can load it, and that you can compare it with other versions, etc.) and save your image (so that next time that you start your image you are in the same working environment).

1.9 Conclusion

In this exercise you learned how to define classes, methods, and define tests. The flow of programming that we chose for this first exercise 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.