CHAPTER 1

CHAPTER 2

Crafting a Simple Embedded DSL with Pharo

In this chapter we will develop a simple domain specific language (DSL) for rolling dice. Players of games such as Dungeon and Dragons are familiar with the DSL that we will implement. An example of such DSL is `2 D20 + 1 D6` which means that we should roll two 20-faces dices and one 6 faces die. This little exercise shows how we can (1) simply reuse traditional operator such as `+`, (2) develop an embedded DSL and (3) use class extensions (the fact that we can define a method in another package than the one of the class of the method).

2.1 Getting started

Using the code browser, define a package named `Dice` or any name your like.

Creating a test

It is always empowering to verify that the code we write is always working as we defining it. For this purpose we will create a unit test. Remember unit testing was promoted by K. Beck first in the ancestor of Pharo. Nowadays this is a common practice but this is always useful to remember our roots!

So we define the class `DieTest` as a subclass of `TestCase`.

```
TestCase subclass: #DieTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Dice'
```

The first test we define is that creating an instance of the class `Die` does not raise an Error. It is not really a good test but enough to get started with.

```
DieTest >> testInitializeIsOk
    self shouldnt: [ Die new ] raise: Error
```

What we can test is that the default number of faces of a die is 6.

```
DieTest >> testInitializeIsOk
    self assert: Die new faces equals: 6
```

If you execute the test, the system will prompt you to create a class `Die`. Do it.

Define the class Die

The class `Die` inherits from `Object` and it has an instance variable, `faces` to represent the number of faces one instance will have.

```
Your code here
```

In the `initialization` protocol, define the method `initialize` so that it simply sets the default number of faces to 6.

```
Die >> initialize
  Your code here
```

Do not hesitate to add a class comment.

Now define a method to return the number of faces an instance of `Die` has.

```
Die >> faces
  ^ faces
```

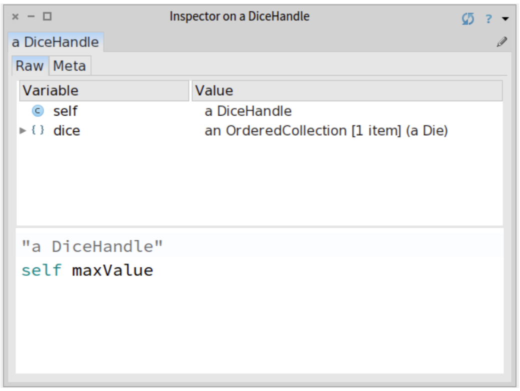
Now your tests should all pass (and turn green).

2.2 Rolling a die

To roll a die you should use the method from Number `atRandom` which draws randomly a number between one and the receiver. For example `10 atRandom` draws number between 1 to 10. Therefore we define the method `roll`:

```
Die >> roll
    Your code here
```

Now we can create an instance `Die new` and send it the message `roll` and get a result. Do `Die new inspect` and then type in the bottom pane `self roll`. You should get an inspector like the one shown in Figure 2.1. With it you can interact with a die by writing expression in the bottom pane.



Variable	Value
self	a DiceHandle
dice	an OrderedCollection [1 item] (a Die)

```
"a DiceHandle"
self maxValue
```

Figure 2.1 Inspecting and interacting with a die.

2.3 Creating another test

But better, let us define a test that verifies that rolling a new created dice with a default 6 faces only returns value comprised between 1 and 6. This is what the following test method is actually specifying.

```
DieTest >> testRolling
  | d |
  d := Die new.
  10 timesRepeat: [ self assert: (d roll between: 1 and: 6) ]
```

Note Often it is better to define the test even before the code it tests. Why? Because you can think about the API of your objects and a scenario that illustrate their correct behavior. It helps you to program your solution.

2.4 Instance creation interface

We would like to get a simpler way to create `Die` instances. For example we want to create a 20-faces die as follows: `Die faces: 20`. Let us define a test for it.

```
DieTest >> testCreationIsOk
  self assert: (Die faces: 20) faces equals: 20
```

You should define the *class* method `faces:` as follows. It creates an instance then send the message `faces:` to it and returns the instance.

```
Die class >> faces: aNumber

  | instance |
  instance := self new.
  instance faces: aNumber.
  ^ instance
```

If you execute it will not work since we did not create yet the method `faces:`. This is now the time to define it.

```
Die >> faces: aNumber
  faces := aNumber
```

Now your tests should run. So even if the class `Die` could implement more behavior, we are ready to implement a dice handle. In a first reading you can skip the following discussion.

Alternate instance creation definition

The *class* method definition `faces:` above is strictly equivalent to the one below.

```
Die class >> faces: aNumber
  ^ self new faces: aNumber; yourself
```

Let us explain it a bit. `self` represents the class `Die` itself. Sending it the message `new`, we create an instance and send it the `faces:` message. And we return the expression. So why do we need the message `yourself`. The message `yourself` is needed to make sure that whatever value the instance message `faces:` returns, the instance creation method we are defining returns the new created instance. You can try to redefine the instance method `faces:` as follows:

```
Die >> faces: aNumber
  faces := aNumber.
  ^ 33
```

Without the use of `yourself`, `Die faces: 20` will return `33`. With `yourself` it will return the instance.

The trick is that `yourself` is a simple method defined on `Object` class: The message `yourself` returns the receiver of a message. The use of `;` sends the message to the receiver of the previous message (here `faces:`). The message `yourself` is then sent to the object resulting from the execution of the expression `self new` (which returns a new instance of the class `Die`), as a consequence it returns the new instance.

2.5 First specification of a dice handle

Let us define a new class `DiceHandle` that represents a dice handle. Here is the API that we would like to offer for now. We create a new instance of the handle then add some dice to it.

```
DiceHandle new
  addDie: (Die faces: 6);
  addDie: (Die faces: 10);
  yourself
```

Of course we will define tests first for this new class. We define the class DiceHandleTest.

```
TestCase subclass: #DiceHandleTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Dice'
```

Testing a Dice Handle

We define a new test method as follows. We create a new handle and add one die of 6 faces and one die of 10 faces. We verify that the handle is composed of two dice.

```
DiceHandleTest >> testCreationAdding
  | handle |
  handle := DiceHandle new
      addDie: (Die faces: 6);
      addDie: (Die faces: 10);
      yourself.
  self assert: handle diceNumber = 2.
```

In fact we can do it better and we add a new test method that verifies that we can even add two dice having the same number of faces.

```
DiceHandleTest >> testAddingTwiceTheSameDice
  | handle |
  handle := DiceHandle new.
  handle addDie: (Die faces: 6).
  self assert: handle diceNumber = 1.
  handle addDie: (Die faces: 6).
  self assert: handle diceNumber = 2.
```

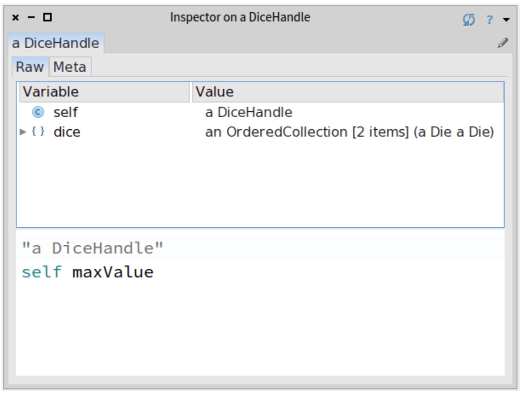
Now that we specified what we want, we should implement the expected class and messages. Easy!

2.6 Defining the DiceHandle class

The class DiceHandle inherits from Object and it defines one instance variable to hold the dice it contains.

```
Your code here
```

We simply initialize it so that its instance variable dice contains an instance of OrderedCollection.



```
Variable    Value
self        a DiceHandle
dice        an OrderedCollection [2 items] (a Die a Die)
```

```
"a DiceHandle"
self maxValue
```

Figure 2.2 Inspecting a DiceHandle.

```
DiceHandle >> initialize
  Your code here
```

Then define a simple method `addDie:` to add a die to the list of dice of the handle. You can use the message `add:` sent to a collection.

```
DiceHandle >> addDie: aDie
  Your code here
```

Now you can execute the code snippet and inspect it. You should get an inspector as shown in Figure 2.2

```
DiceHandle new
  addDie: (Die faces: 6);
  addDie: (Die faces: 10);
  yourself
```

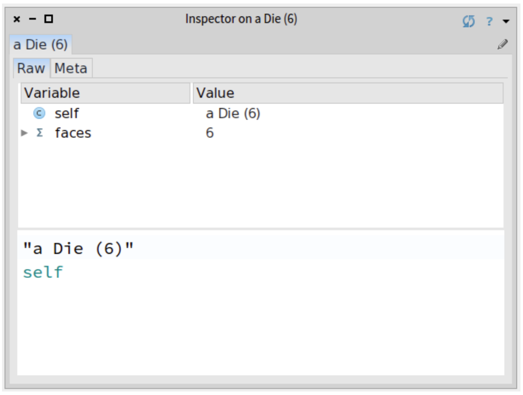
Finally we should add the method `diceNumber` to the `DiceHandle` class to be able to get the number of dice of the handle. We just return the size of the dice collection.

```
DiceHandle >> diceNumber
  ^ dice size
```

Now your tests should run and this is good moment to save and publish your code.

2.7 Improving programmer experience

Now when you open an inspector you cannot see well the dice that compose the dice handle. Click on the `dice` instance variable and you will only get a list of `a Dice` without further information. What we would like to get is



Variable	Value
self	a Die (6)
faces	6

```
"a Die (6)"
self
```

Figure 2.3 Die details.

something like a `Die (6)` or a `Die (10)` so that in a glance we know the faces a die has.

```
DiceHandle new
  addDie: (Die faces: 6);
  addDie: (Die faces: 10);
  yourself
```

This is the message `printOn:` that is responsible to provide a textual representation of the message receiver. By default, it just prints the name of the class prefixed with `'a'` or `'an'`. So we will enhance the `printOn:` method of the `Die` class to provide more information. Here we simply add the number of faces surrounded by parenthesis. The `printOn:` message is sent with a stream as argument. This is in such stream that we should add information. We use the message `nextPutAll:` to add a number of characters to the stream. We concatenate the characters to compose `()` using the message `,` comma defined on collections (and that concatenate collections and strings).

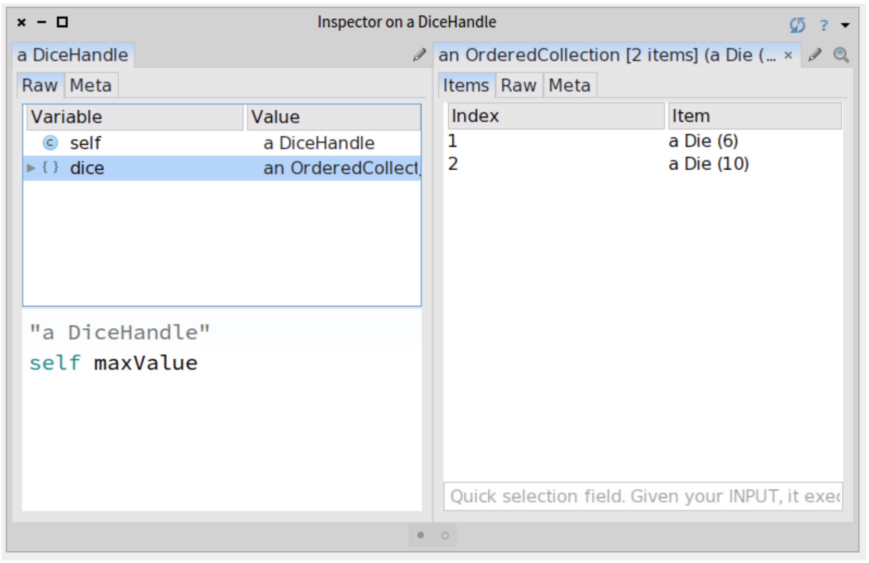
```
Die >> printOn: aStream

  super printOn: aStream.
  aStream nextPutAll: ' (', faces printString, ')'
```

Now in your inspector you can see effectively the number of faces a dice handle has as shown by Figure 2.3 and it is now easier to check the dice contained inside a handle (See Figure 2.4).

2.8 Rolling a dice handle

Now we can define the rolling of a dice handle by simply summing result of rolling each of its dice. Implement the `roll` method of the `DiceHandle` class.


Inspector on a DiceHandle
a DiceHandle
Raw
Meta
an OrderedCollection [2 items] (a Die (...
Items
Raw
Meta
Quick selection field. Given your INPUT, it exe

Variable	Value
self	a DiceHandle
dice	an OrderedCollect

```
"a DiceHandle"
self maxValue
```

Index	Item
1	a Die (6)
2	a Die (10)

Figure 2.4 Dice Handle with more information.

This method must collect the results of rolling each dice of the handle and sum them.

> **Note** You may want to have a look at the method `sum` in the class `Collection` or use a simple loop.

```
DiceHandle >> roll
  Your code here
```

Now we can send the message `roll` to a dice handle.

```
handle := DiceHandle new
    addDie: (Die faces: 6);
    addDie: (Die faces: 10);
    yourself.
handle roll
```

Define a test to cover such behavior. Rolling an handle of n dice should be between n and the sum of the face number of each die.

```
DiceHandleTest >> testRoll
  Your code here
```

2.9 Role playing syntax

Now we are ready to offer a syntax following practice of role playing game, i.e., using `2 D20` to create a handle of two 20 faces dice. For this purpose we will define class extensions: we will define methods in the class `Integer` but these methods will be only available when the package Dice will be loaded.

But first let us specify what we would like to obtain by writing a new test in the class `DiceHandleTest`. Remember to always take any opportunity to write tests. When we execute `2 D20` we should get a new handle composed of two dice and can verify that. This is what the method `testSimpleHandle` is doing.

```
DiceHandleTest >> testSimpleHandle
  self assert: 2 D20 diceNumber = 2.
```

Verify that the test is not working! It is much more satisfactory to get a test running when it was not working before. Now define the method `D20` with a protocol named `'*Dice`' (if you named your package 'Dice'). The `*` (star) prefixing the protocol name indicates that the protocol belongs to another package. You can define this method either from the debugger resulting from the execution of the failed test or in the code browser.

The method `D20` simply creates a new dice handle, adds the correct number of dice to this handle, and returns the handle.

```
Integer >> D20
  Your code here
```

About class extensions

We asked you to place the method `D20` in a protocol starting with a star and having the name of the package (`'*Dice'`) because we want this method to be saved (and packaged) together with the code of the classes we already created (`Die`, `DiceHandle`,...) Indeed in Pharo we can define methods in classes that are not defined in our package. Pharoers call this action a class extension: we add method to a class that is not ours. For example `D20` is defined on the class `Integer`. Now such methods only make sense when the package Dice is loaded. This is why we want to save and load such methods with the package we created and using the `'*Dice'` notation is way for the system to know that it should save the methods with the package and not with the package of the class `Integer`.

Now your tests should pass and this is probably a good moment to save your work either by publishing your package and to save your image.

We can do the same for the default dice with different faces number: 4, 6, 10, and 20. But we should avoid duplicating logic and code. So first we will introduce a new method `D:` and based on it we will define all the others.

Make sure that all the new methods are placed in the protocol `'*Dice'`. To verify you can press the button Browse of the Monticello package browser and you should see the methods defined in the class `Integer`.

```
Integer >> D: anInteger
  Your code here
```

```
Integer >> D4

  ^ self D: 4
```

```
Integer >> D6

  ^ self D: 6
```

```
Integer >> D10

  ^ self D: 10
```

```
Integer >> D20

  ^ self D: 20
```

We have now a compact form to create dice and we are ready for the last part: the addition of handles.

2.10 Handle's addition

Now what is missing is that possibility to add several handles as follows: `2 D20 + 3 D10`. Of course let's write a test first to be clear on what we mean.

```
DiceHandleTest >> testSumming
  | handle |
  handle := 2 D20 + 3 D10.
  self assert: handle diceNumber = 5.
```

We will define a method `+` on the HandleDice class. In other languages this is often not possible or is based on operator overloading. In Pharo `+` is just a message as any other, therefore we can define it on the classes we want.

Now we should ask ourself what is the semantics of adding two handles. Should we modify the receiver of the expression or create a new one. We preferred a more functional style and choose to create a third one.

The method `+` creates a new handle then add to it the dice of the receiver and the one of the handle passed as argument to the message. Finally we return it.

```
DiceHandle >> + aDiceHandle
  Your code here
```

Now we can execute the method `(2 D20 + 1 D6) roll` nicely and start playing role playing games, of course.

2.11 Solutions

Here are the possiblle solutions of the implementation we asked.

Define class Die

```
Object subclass: #Die
  instanceVariableNames: 'faces'
  classVariableNames: ''
  package: 'Dice'
```

```
Die >> initialize
  super initialize.
  faces := 6
```

Rolling a die

```
Die >> roll
    ^ faces atRandom
```

Define class DieHandle

```
Object subclass: #DiceHandle
  instanceVariableNames: 'dice'
  classVariableNames: ''
  package: 'Dice'
```

```
DiceHandle >> initialize
  super initialize.
  dice := OrderedCollection new.
```

Die addition

```
DiceHandle >> addDie: aDie
  dice add: aDie
```

2.12 Rolling a dice handle

```
DiceHandle >> roll

  | res |
  res := 0.
  dice do: [ :each | res := res + each roll ].
  ^ res
```

2.13 Role playing syntax

```
Integer >> D20
  | handle |
  handle := DiceHandle new.
  self timesRepeat: [ handle addDie: (Die faces: 20)].
  ^ handle
```

```
Integer >> D: anInteger

  | handle |
  handle := DiceHandle new.
  self timesRepeat: [ handle addDie: (Die faces: anInteger)].
  ^ handle
```

2.14 Adding DiceHandles

```
DiceHandle >> + aDiceHandle

  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  aDiceHandle dice do: [ :each | handle addDie: each ].
  ^ handle
```

This definition only works if the method `dice` defined below has been defined

```
DiceHandle >> dice
  ^ dice
```

Indeed the first expression `self dice do:` could be rewritten as `dice do:` because dice is an instance variable of the class `DiceHandle`. Now the expression `aDiceHandle dice do:` cannot. Why? Because in Pharo you cannot access the state of another object directly. Here `2 D20` is one handle and `3 D10` another one. The first one cannot access the dice of the second one directly (while it can accessed its own). Therefore there is a need to define a message that provide access to the dice.

2.15 Conclusion

This chapter illustrates how to create a small DSL based on the definition of some domain classes (here `Dice` and `DiceHandle`) and the extension of core class such `Integer`. It shows that in Pharo we can use usual operators to express natural models.