CHAPTER 1

Crafting a Simple Embedded DSL with Pharo

In this tutorial we develop a simple Domain Specific Language (DSL) for rolling dice. Players of games such as Dungeon and Dragons are familiar with the DSL we implement. An example of such DSL is `2 D20 + 1 D6`. This example means that we should roll two 20-faces dice and one time a 6 faces die. This tutorial shows how we can (1) simply reuse traditional operator such as `+`, (2) develop an embedded DSL and (3) use class extensions (aka open classes).

1.1 Getting Started

Using the code browser (click on the background to open the `World menu` and select `System Browser`), define a package named `Dice` (contextual menu, `Add package`).

Defining the Class Die

In this package, define a class `Die` with an instance variable `faces` (contextual menu, `Add class`).

```
Your code here
```

Add an `initialize` protocol, and define a method `initialize` that simply sets the default number of faces to 6. Save this method (`ctrl s`)

```
Your code here
```

Creating a Test

It is always empowering to verify that the code we write is always working as we defined it. For this purpose we create a unit test. So we define the class `DieTest` as a subclass of `TestCase`.

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

We add a new test `testInitializeIsOk` to make sure we can create a new dice.

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

When this method is saved, you can click on the gray icon in front of the method name in the methods pane to execute the test. The icon should become green.

1.2 Rolling a Die

To roll a die we use the method `atRandom` from `Number`. This method randomly chooses a number between one and the receiver. For example `10 atRandom` draws a number between 1 and 10. Now, define the `roll` method of the class `Die` class so that it returns a random number between 1 and the number of faces.

```
Your code here
```

Now we can create a new instance of `Die`, send it the message `roll`, and get a result. Open a workspace (`World menu`, `Workspace`). Write `Die new inspect` and execute the expression (`ctrl d`). You should get an inspector like the one shown in Figure 1.1. With it you can interact with the die by writing expressions in the bottom pane. In this pane, type `self roll` and print the result (`ctrl p`). You should get a random number.

Creating Another Test

We define a test that verifies that rolling a new created dice (with 6 faces by default) only returns values comprised between 1 and 6. This is what the following test method is actually specifying.

```
DieTest >> testRolling
    | d |
    d := Die new.
    1000 timesRepeat: [ self assert: (d roll between: 1 and: 6) ]
```

Execute the test to make sure it runs fine.

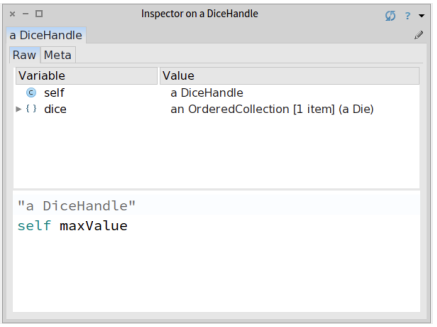

Figure 1.1 Inspecting and interacting with a Die.

1.3 Instance Creation Interface

Now, we would like to get a simpler way to create `Die` instances. For example we want to create a 20-faces dice as follows: `Die faces: 20`. Let us define a test for it. When you save this method, you might get a warning message saying that `faces:` is an unknown method name (aka., selector). Just confirm the name to validate.

```
DieTest >> testCreationIsOk
    self shouldnt: [ Die faces: 20 ] raise: Error
```

Execute the test and you should get an error message because the class `Die` does not have a method `faces:`. The message `faces:` is sent to the class `Die` and not to an instance of this class. Such methods are called class methods or class-side methods; this is equivalent to static methods in Java. To add such a method to a class from the code browser, you have to tick the `class` checkbox.

Another way to create this method is to do it right from the debugger that pops up when a method does not exist. Just click the `Create` button in the debugger, select the `Die` class, and the `instance creation` protocol. The method first creates an instance, then sends it the message `faces:`, and returns the instance. You can implement the method right in the debugger.

```
Die class >> your code here
```

To implement this method, you will have to send the message `new` to the class: `self` is the current class when implementing a class method.

> **Note** If your implementation of the method uses a temporary variable, you may want to simplify it by using the cascade operator `;` and the `yourself` message. Look at the implementation of `Object >> yourself` and its senders (contextual menu, `Senders of`) to get examples on how to do this.

This method can not yet be executed because you first have to implement the instance-side method `faces:` (the setter for the variable) that configure a new die with a number of faces passed as an argument. In Java, you would not have to do that because constructors can access instance variables. In Pharo, a class method only knows about the class variables not the instance variables. You can keep on using the debugger to do that or you can go back to the code browser.

```
Your code here
```

Now, all your tests should pass and this is good moment to save your code (`World menu`, `Save`).

So even if the class `Die` could implement more behavior, we are ready to implement a dice handle.

1.4 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
```

Testing a Dice Handle

Of course we define a test for this new class. We define the class `DiceHandleTest` as follow.

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

We define a new test method as follows.

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

When you save the method, the code browser will offer to create the class and may warn you that `addDice` and `diceNumber` are unknown method names.

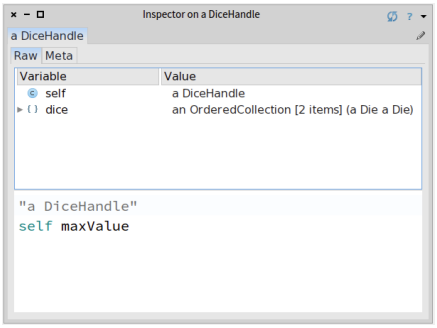

Figure 1.2 Inspecting a dice handle

With another test, we can make sure we can add two times a similar dice.

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

Defining the DiceHandle class

The `DiceHandle` class defines one instance variable named `dice` to hold a collection of dice. Add this variable to the class now.

When an instance is initialized, the instance variable `dice` must contain an empty `OrderedCollection`. Implement the corresponding `initialize` method.

```
Your code here
```

Then we define a simple method (`addDie:`) to add a dice to the list of dice of the handle. You can do that from the code browser or from a debugger after having run a failing unit test.

```
Your code here
```

Now you can execute the code snippet and inspect it (`ctrl i`):

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

You should get an inspector as shown in Figure 1.2.

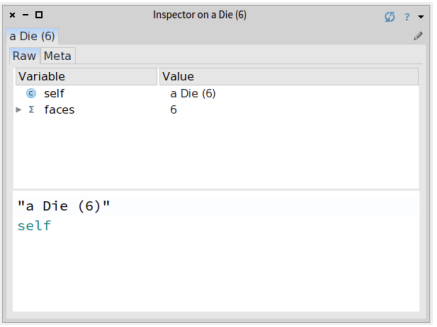

Figure 1.3 Die details.

Finally we should add the method `diceNumber` to the `DiceHandle` class to get the number of dice of the handle. The method returns the size of the `dice` collection.

```
Your code here
```

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

1.5 Improving Programmer Experience

When you open an inspector on a dice handle (such as the one in Figure 1.2) you cannot see the details of the dices that compose the dice handle. Click on the `dice` instance variable and you only get a list of `"a Die"` without further information as you can see in Figure 1.2.

Enhance the `printOn:` method of the `Die` class to provide more information: simply add the number of faces surrounded by parenthesis to make the result look like the one in Figure 1.4.

> **Note** You may want to have a look at all the implementors of `printOn:` (contextual menu of the `printOn:` method, `Implementors of`) to get examples.

```
Your code here
```

Now in your inspector you can see the number of faces a dice has as shown by Figure 1.3 and it is now easier to check the dice contained inside a handle (see Figure 1.4).

1.6 Rolling a Dice Handle

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

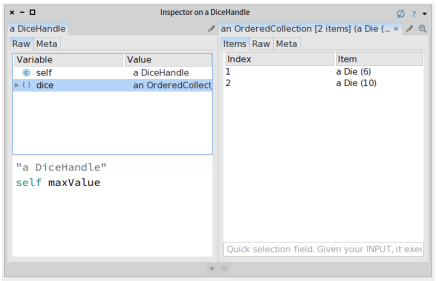

Figure 1.4 Dice Handle with more information.

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`.

```
Your code here
```

We can now send the message `roll` to a dice handle.

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

1.7 Role Playing Syntax

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 use class extensions: we define methods in the class `Integer`. We want these methods to only be available when the package `Dice` is loaded.

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 equals: 2.
```

Verify that the test is not passing! It is much more satisfactory to get a test running when it was not working before. Now define the method `D20` with a protocol name `*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.

```
Your code here
```

Your test should pass and this is probably a good moment to save your work.

We could do the same for different face numbers: 4, 6 and 10. But we should avoid duplicating logic and code. We first introduce a new method `D:` in the `Integer` class and, based on it, we define all the others.

```
Your code here
```

The `D20` method should now look like:

```
Integer >> D20
    ^ self D: 20
```

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

Adding Handles

Now we can simply support the addition of handles. Let us write a test first.

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

We define a `+` method on the `DiceHandle` 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 the `+` method on the classes we want.

What is the semantics of adding two handles? Should we modify the receiver of the expression or create a new handle? In this tutorial we choose the functional style and decide that a new handle should be created. As a result, the method `+` first creates a new handle then adds to it to the dice of the receiver and the ones of the handle passed as argument. Finally, `+` must return the newly created handle. To access the dice of the dice handle passed as a parameter, you will have to add a dedicated accessor.

```
Your code here
```

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

1.8 Conclusion

This tutorial illustrates how to create a small DSL based on the definition of some domain classes (here `Die` and `DiceHandle`). We used class extension (on `Integer`) to simplify the instanciation of these classes. This tutorial shows that in Pharo we can use standard operators to express natural models.