

Chapter 1

Crafting a little Embedded DSL

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 we will implement. An example of such DSL is $2 D20 + 1 D6$ which means that we should roll two times a 20-faces dice and one time a 6 faces dice. This chapter will show how we can (1) simply reuse traditional operator such as $+$, (2) develop an embedded DSL and (3) show a nice usage of class extensions.

1.1 Getting started

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

Defining the class Dice

```
Object subclass: #Dice
  instanceVariableNames: 'faces'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Dice'
```

In the initialize category, define the method initialize as follows. It simply set the default number of faces to 6.

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

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 Smalltalk. Nowadays this is a common practice but this is always useful to remember our roots!

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

```
TestCase subclass: #DiceTest
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Dice'
```

```
DiceTest>>testInitializesOk
```

```
self shouldnt: [ Dice new ] raise: Error
```

1.2 Rolling a dice

To roll a dice we will use the method from `Number` `atRandom` which draws randomly a number between one and the receiver. For example `10 at random` draws number between 1 to 10. Therefore we define the method `roll` as follows.

```
Dice>>roll
```

```
^ faces atRandom
```

Now we can create an instance `Dice new` and send it the message `roll` and get a result. Do `Dice new inspect` and then type in the bottom pane `self roll`. You should get an inspector like the one shown in Figure 1.1. With it you can interact with the dice by writing expression in the bottom pane.

Creating another test

We will 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.

```
DiceTest>>testRolling
```

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

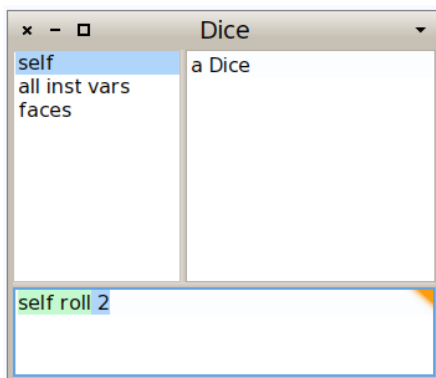


Figure 1.1: Inspecting and interacting with a Dice.

Note that 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.

1.3 Instance creation interface

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

```
DiceTest>>testCreationIsOk
```

```
self shouldnt: [ Dice faces: 20 ] raise: Error
```

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

```
Dice class>>faces: aNumber
```

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

This method is strictly equivalent to the one below. The trick is that your-self is a simple method defined on Object class. yourself returns the receiver of a message and the use of ; sends the message to the receiver of the previous message (here faces:), therefore yourself is sent to the object resulting from the execution of the expression self new (which returns a new instance of the class Dice).

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

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

```
Dice>>faces: aNumber
  faces := aNumber
```

Now your tests should run.

So even if the class `Dice` 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
  addDice: (Dice faces: 6);
  addDice: (Dice faces: 10);
  yourself
```

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

Testing Handle Dice

```
TestCase subclass: #DiceHandleTest
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Dice'
```

We define a new test method as follows.

```
DiceHandleTest>>testCreationAdding
  | handle |
  handle := DiceHandle new
    addDice: (Dice faces: 6);
```

```

    addDice: (Dice faces: 10);
    yourself.
    self assert: handle diceNumber = 2.

```

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

```

DiceHandleTest>>testAddingTwiceTheSameDice

```

```

| handle |
handle := DiceHandle new
    addDice: (Dice faces: 6);
    yourself.
self assert: handle diceNumber = 1.
handle addDice: (Dice faces: 6).
self assert: handle diceNumber = 2.

```

Defining the DiceHandle class

This class defines one instance variable to hold dices it contains.

```

Object subclass: #DiceHandle
    instanceVariableNames: 'dice'
    classVariableNames: ""
    poolDictionaries: ""
    category: 'Dice'

```

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

```

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

```

Then we define a simple method to add a dice to the list of dices of the handle.

```

DiceHandle>>addDice: aDice
    dice add: aDice

```

Now you can execute the code snippet and inspect it. You should get an inspector as shown in Figure 1.2

```

DiceHandle new
    addDice: (Dice faces: 6);
    addDice: (Dice faces: 10);
    yourself

```

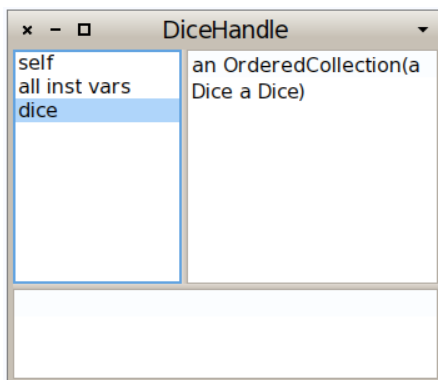


Figure 1.2: Inspecting a Dice.

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 returns the size of the dices collection.

```
DiceHandle>>diceNumber
```

```
^ dice size
```

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

1.5 Improving programmer experience

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

```
DiceHandle new
```

```
  addDice: (Dice faces: 6);
  addDice: (Dice faces: 10);
  yourself
```

So we will enhance the `printOn:` method of the `Dice` class to provide more information. Here we simply add the number of faces surrounded by parenthesis.

```
Dice>>printOn: aStream
```

```
  super printOn: aStream.
```

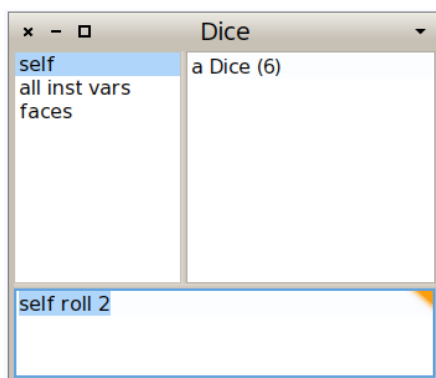


Figure 1.3: Dice details.

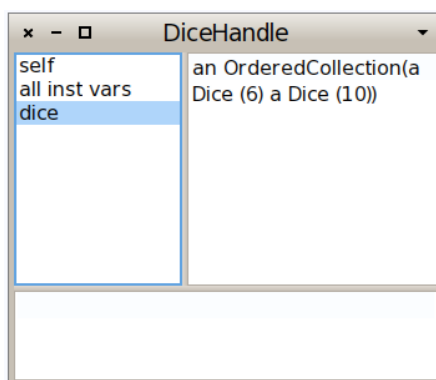


Figure 1.4: Dice Handle with more information.

```
aStream nextPutAll: ' ('; faces printString, ')
```

Now in your inspector you can see effectively the number of faces a dice handle 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 handle of dice by simply summing the dice rolls.

```
DiceHandle>>roll
```

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

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

```
handle := DiceHandle new  
  addDice: (Dice faces: 6);  
  addDice: (Dice faces: 10);  
  yourself.  
handle roll
```

1.7 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 category name that is `*Dice` (if you named your package `Dice`). This method simply creates a new dice handle, add the correct number of dice to this handle and return it.

```
Integer>>D20
```

```
| handle |  
handle := DiceHandle new.  
self timesRepeat: [ handle addDice: (Dice faces: 20)].  
^ handle
```


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

Now we could 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

```
Integer>>D: anInteger
```

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

```
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.

Handle's Addition

Now we can simply support the addition of handles. But of course let's write a test first.

```
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 the 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
```

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

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

1.8 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.