CHAPTER 2

Revisiting the Die DSL: a Case for Double Dispatch

In this chapter we extend the Die DSL to support the sum of die with other die or with die handles. While this extension at first may look trivial, we will take it as a way to explore double dispatch. Indeed the solution we propose, called double dispatch, does not use any explicit conditionals and is the basis of more advanced Design Patterns such as the Visitor. The solution is based on the *Don't ask, tell* object-oriented principle.

2.1 A little reminder

In a previous chapter you implemented a small DSL to add dice and manage dieHandle. With this DSL, you could create dice and add them to a die handle. Later on you could sum two different die handle and obtain a new one following the "Dungeons and Dragons" ruling book. The following tests show these two behavior:

```
DieHandleTest >> testCreationAdding
  | handle |
  handle := DiceHandle new
    addDice: (Dice faces: 6);
    addDice: (Dice faces: 10);
    yourself.
  self assert: handle diceNumber = 2
```

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

2.2 New requirements

The first requirement we have is that we want to be able to add two dices together and of course we should obtain a die handle as illustrated by the following test.

We want to add two dices together:

```
(Die withFaces: 6) + (Die withFaces: 6)
```

The second requirement is that we want to be able to mix and add a die to a die handle or vice versa as illustrated below:

```
2 D20 + (Die withFaces: 6)
```

```
 (Die withFaces: 6) + 2 D20
```

2.3 Turning requirements as tests

Since we are test-infested, we turn such expected behavior into automatically testable expected behavior: we write them as tests.

We want to add two dices together:

```
DieTest >> testAddTwoDice
  | hd |
  hd := (Die withFaces: 6) + (Die withFaces: 6).
  self assert: hd dice size = 2.
```

The second requirement is that we want to be able to mix and add a die to a die handle or vice versa as illustrated by the two following tests:

```
DieTest >> testAddingADieAndHandle
  | hd |
  hd := (Die faces: 6)
    +
    (DieHandle new
      addDie: 6;
      yourself).
  self assert: hd dice size equals: 2
```

```
DieHandleTest >> testAddingAnHandleWithADie
  | handle res |
  handle := DieHandle new
    addDie: (Die faces: 6);
    addDie: (Die faces: 10);
    yourself.
  res := handle + (Die faces: 20).
  self assert: res diceNumber equals: 3
```

Now we are ready to implement such requirements.

2.4 A first implementation

A first solution is to explicit type check the argument to decide what to do.

```
DieHandle >> + aDieOrADieHandle

  ^ (aDieOrADieHandle class = DieHandle)
    ifTrue: [ | handle |
          handle := self class new.
          self dice do: [ :each | handle addDie: each ].
          aDieOrADieHandle dice do: [ :each | handle addDie: each ].
          handle ]
    ifFalse: [ | handle |
          handle := self class new.
          self dice do: [ :each | handle addDie: each ].
          handle addDie: aDie.
          handle  ]
```

```
Die >> + aDieOrADieHandle
  | selfAsDiceHandful |
  selfAsDieHandle := DieHandle new addDie: self.
  ^ selfAsDiceHandle + aDieOrADieHandle
```

The problem of this solution is that it does not scale. As soon as we will haveother kinds of arguments we will have to check more and more. You may think that this is just a spurious argument. But when you have a model that has around 35 different kinds of nodes as in Pillar the document processing system used to produce this book, this kind of testing logic becomes a nightmare to maintain and extend.

We can do better. The logic of the solution we have in mind is quite simple but it may be distabilizing when reading the code. In fact we just to tell the receiver of the message + that we want to add to it and argument. As a response to this message, the receiver being a die or an handle will just tell us how the receiver should sum it.

2.5 Adding two dice

Let us step back and start by supporting the addition of two dice. This is rather simple we create and return a die handle to which we add the receiver and the argument.

```
Die >> + aDie

  ^ DieHandle new
    addDie: self;
    addDie: aDie; yourself
```

Our first test should pass `testAddTwoDice`. But this solution does not support the fact that the argument can be either a die or a die handle.

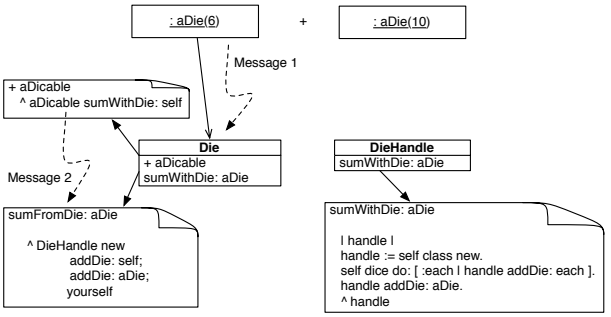

Figure 2.1 Summing two dice and be prepared for more.

2.6 Adding a die and a die or a handle

Now we want to handle the fact that we can add a die or a die handle to the receiver as illustrated by the test `testAddingADieAndHandle`.

```
DieTest >> testAddingADieAndHandle
  | hd |
  hd := (Die faces: 6)
    +
    (DieHandle new
      addDie: 6;
      yourself).
  self assert: hd dice size equals: 2
```

The previous method `+` is definitively what we want to do when we have two dice. So let us rename it as `sumWithDie:` so that we can invoke it later.

```
Die >> sumWithDie: aDie

  ^ DieHandle new
    addDie: self;
    addDie: aDie; yourself
```

Now what we can do is to implement `+` as follows. Notice that we named the argument `aDicable` because we want to convey that the argument can be either a die or a die handle.

```
Die >> + aDicable
  ^ aDicable sumWithDie: self
```

We just tell the argument (which can be a die or a die handle) that we want to add to it an die. And in our two tests `testAddTwoDice` and `testAddingADieAndHandle` we know that the receiver is a die because the method is defined in the class of `Die`. Easy, no. At this point the test `testAddTwoDice` should pass because we are adding two dice as shown in Figure 2.1.

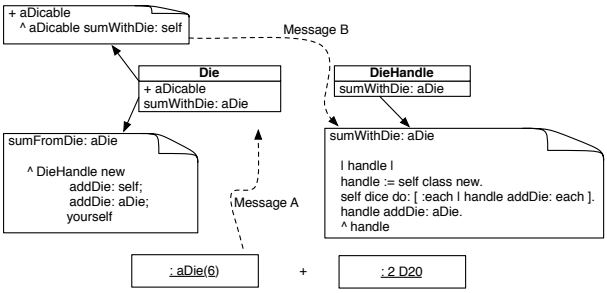

Figure 2.2 Summing a die and a dicable.

Now we still have to find a solution for the case where the argument to the message `+` is a die handle. It is easy, isn't? We know how to add a die to a die handle: we simply creates a new die handle, add all the die of the previous die handle to the new one and add the argument too.

So we just have to define the method `sumWithDie:` to the class `DieHandle` implementing this logic.

```
DieHandle >> sumWithDie: aDie
  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  handle addDie: aDie.
  ^ handle
```

Now we are able to sum a die and a die handle as shown in Figure 2.2. The test `testAddingADieAndHandle` should now pass.

Now you may ask why this is working. We defined two methods `sumWithDie:` and when hhe method `+` on class `Die` will send the message `sumWithDie:` to either a die or a die handle, the message dispatch will select the correct method for us as shown in Figure 2.3.

2.7 Now a DieHandle as receiver

Our solution does not handle the case where the receiver is a die handle and this is what we will address now. Now we are ready to apply the same idea when the receiver is a die handle. We will just say to the argument that we want to add a die handle this time.

We know what is to add two die handles, it is the code we already defined in the past chapter. We rename the `+` method as `sumWithHandle:`. Basically we create a new handle, then we add the dice of the receiver and the argument to it and we return the new handle.

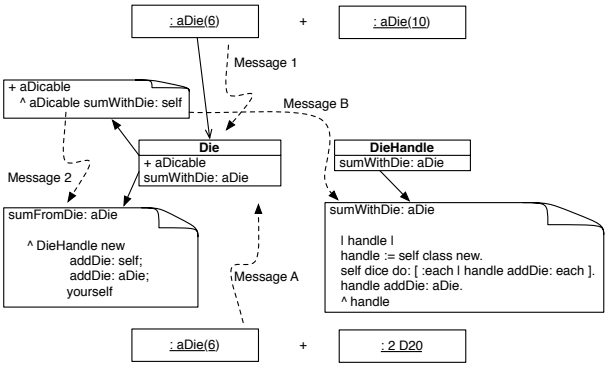

Figure 2.3 Summing a die and a dicable

```
DieHandle >> sumWithHandle: aDieHandle
  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  aDieHandle dice do: [ :each | handle addDie: each ].
  ^ handle
```

Now we can define a more powerful version of + by simply sending the message `sumWithHandle:` to the **argument**. Remember that sending back a new message to the argument is the key aspect. Why? Because we kick in a new message lookup and dispatch.

```
DieHandle >> + aDicable
  ^ aDicable sumWithHandle: self
```

Up until here we did not change much and all the tests adding two dice handle should continue to run.

To get the final behavior just have to define a new method `withWithHandle:` on the `Die` class. The logic is similar to the one adding one die to one die handle.

```
Die >> sumWithHandle: aDieHandle
  | handle |
  handle := DieHandle new.
  aDieHandle dice do: [ :each | handle addDie: each ].
  handle addDie: self
  ^ handle
```

Figure 2.4 shows the full set up. We suggest to follow the execution of messages for the different cases to understand that just sending a new message to the argument and relying on method dispatch produces modular conditional execution. Now the following test should pass and we are done.

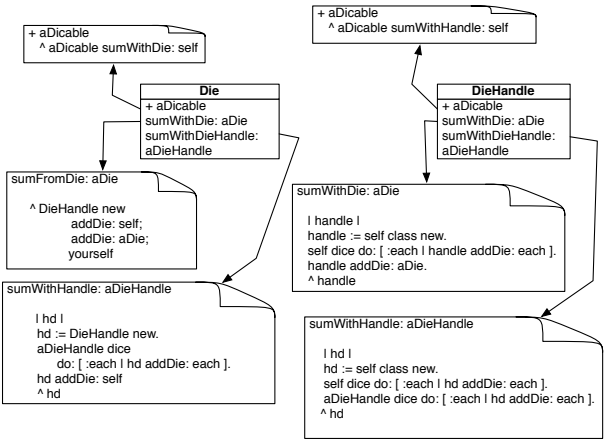

Figure 2.4 Handling all the cases: summing a die/die handle with a die/die handle .

```
DieHandleTest >> testAddingAnHandleWithADie
  | handle res |
  handle := DieHandle new
    addDie: (Die faces: 6);
    addDie: (Die faces: 10);
    yourself.
  res := handle + (Die faces: 20).
  self assert: res diceNumber equals: 3
```

2.8 Conclusion

In this chapter we presented double dispatch. The idea is to use method dispatch several times. While the resulting design is simple, it is not trivial to understand deeply and it requires time to digest it. At its core it relies on the fact that sending a message to an object selects the correct method. So sending another message to the message argument will select a method based on the argument too.

When we step badk we see that we applied twice the *Don't ask, tell* principle: First the message + plus selects the corresponding methods in either `Die` or `DieHandle` classes. Then a more specific message is sent to the argument and the dispatch kicks in again selecting the correct method either `sumWithDie:` or `sumWithHandle:`. Therefore we have effectively selected a method according to the receiver and the argument of a messages.

Double dispatch is the basis for the Visitor Design pattern that is effective when dealing with complex data structure such as documents, compilers. In

such context it is not rare to have more than 30 or 40 different nodes that should be manipulated together to produce specific behavior.