CHAPTER 1

Revisiting the Die DSL: a Case for Double Dispatch

In Chapter **??**, using the Die DSL we could only sum die handles together as in `2 D20 + 1 D4`. In this new chapter we extend the Die DSL implementation to support the sum of a die with another one or with a die handle (and vice versa).

One of the challenges is that the message + should be able to manage different types of receivers and arguments. The message will have either a die or a die handle as receiver and arguments, so we should manage the following possibilities: die + die handle, die + die, die handle + die handle, and die handle + die. While this extension at first may look trivial, we will take it as a way to explore double dispatch.

Double dispatch is a technic that avoids hardcoding type checks and also is able to define incrementally the behavior handling all the possible cases. Indeed double dispatch does not use any explicit conditionals and is the basis of more advanced Design Patterns such as the Visitor.

Double dispatch is based on the *Don't ask, tell* object-oriented principle applied twice. In the case of the + message, there is a first dispatch to select the adequate method. Then a second dispatch happens when in this method a new message is sent to the *argument* of the + message telling this argument the way the current receiver should be summed. This description is clearly too abstract so we will go over a full example to explain it.

1.1 A little reminder

In a previous chapter you implemented a small DSL to add dice and manage die handles. With this DSL, you could create dice and add them to a die handle. Later on you could sum two different die handles and obtain a new one following the ”Dungeons and Dragons” ruling book.

The following tests show these two behaviors: First the dice handle creation and second the sum of die handles.

```
DieHandleTest >> testCreationAdding
 | handle |
 handle := DieHandle new
  addDie: (Die withFaces: 6);
  addDie: (Die withFaces: 10);
  yourself.
 self assert: handle diceNumber equals: 2
```

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

The implementation of + was simple since we could only sum die handles together. The method + creates a new handle, adds the dice of the receiver and of the argument to the newly created handle and returns it.

```
DieHandle >> + aDieHandle
 "Returns a new handle that represents the addition of the receiver
    and the argument."
 | handle |
 handle := self class new.
 self dice do: [ :each | handle addDie: each ].
 aDieHandle dice do: [ :each | handle addDie: each ].
 ^ handle
```

1.2 [Optional] Alternate way

We could also implement + using by asking the argument die handle to add its own dice as follows:

```
DieHandle >> + aDieHandle
 "Returns a new handle that represents the addition of the receiver
    and the argument."
 | handle |
 handle := self class new.
 self dice do: [ :each | handle addDie: each ].
 aDieHandle addDiceTo: handle.
 ^ handle
```

Implement the corresponding method `addDiceTo:` and verify that your tests still pass.

1.3 New requirements

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

We want to add two dice 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
```

1.4 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 dice together:

```
DieTest >> testAddTwoDice
 | hd |
 hd := (Die withFaces: 6) + (Die withFaces: 6).
 self assert: hd dice size equals: 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 withFaces: 6)
  +
  (DieHandle new
   addDie: (Die withFaces: 10);
   yourself).
 self assert: hd dice size equals: 2
```

```
DieHandleTest >> testAddingAnHandleWithADie
 | handle res |
 handle := DieHandle new
  addDie: (Die withFaces: 6);
  addDie: (Die withFaces: 10);
  yourself.
 res := handle + (Die withFaces: 20).
```

```
 self assert: res diceNumber equals: 3
```

The two previous tests are not really robust so we will introduce a little behavior to make sure that we can have much better tests.

1.5 Introducing faces on DieHandle

The previous test `testAddingADieAndHandle` is not really good because it can pass just if we add two objects in the die handle and this is not really satisfactory. We will introduce `numberOfFaces`. This method should satisfy the following test:

```
DieTest >> testNumberOfFaces
 | hd |
 hd := (DieHandle new
   addDie: (Die withFaces: 10);
   addDie: (Die withFaces: 6);
   yourself).
 self assert: hd faces equals: 16
```

Define the method `faces` on `DieHandle`. It is following nearly the same logic as the method `roll`.

```
DieHandle >> faces
 "return the number of faces of the receiver"
 ...
```

Now we are ready to implement such requirements.

1.6 The first implementation

The first solution is to explicitly 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: aDieOrADieHandle.
     handle  ]
```

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

The problem of this solution is that it does not scale. As soon as we will have other kinds of arguments we will have to check more and more cases. 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.

1.7 Sketching double dispatch

We can do better. The logic of the solution we have in mind is quite simple but it may be destabilizing at first. Let us sketch it.

- When we execute a method we know its receiver and the kind of receiver we have: it can be a die or a die handle. The method dispatch will select the correct method at runtime. Imagine that we have two `+` methods for each class `Die` and `DieHandle`. When a given method `+` will be executed, we will know the exact kind of the receiver. For example, when the method `+` defined on the class `Die` will be executed, we will know that the receiver is a die (instance of this class). Similarly when the method `+` defined on the class `DieHandle` will be executed, we will know that the message receiver is a die handle. This is the power of method dispatch: it selects the right method based on the message receiver.
- Then the idea is to tell the argument that we want to sum it with that given receiver. It means that each `+` method on a different class has just to send a different message based on the fact that the receiver was a die or a die handle to its argument and let the method dispatch to act once again. After this second dispatch, the correct method will be selected.

But let us makes this really concrete.

1.8 Adding two dice

Let us step back and start by supporting the sum 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.

1.9 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 withFaces: 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
 ... Your code ...
```

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
 ... Your code ...
```

We tell the argument `aDicable` (which can be a die or a die handle) that we want to add a die to it (we know that `self` in this method is a `Die` because this is the method of this class that is executed). When rewritting the `+` method, we switched `self` and `aDicable` to send the new message `sumWithDie:` to the argument (`aDicable`). This switch kicks a new method dispatch and we finally have a double dispatch (one of `+` and one for `sumWithDie:`).

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`. At this point the test `testAddTwoDice` should pass because we are adding two dice as shown in Figure 1-1.

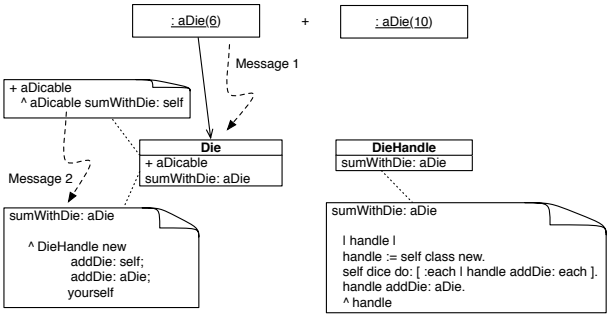

Figure 1-1 Summing two dice and be prepared for more.

1.10 When the argument is a die handle

Now we still have to find a solution for the case where the argument to the message `+` is a die handle. In fact, the argument will receive the message `sumWithDie:`. Therefore if we define a method with that name in the class `DieHandle` it will be executed when the argument of message `+` is a die handle.

We know how to sum a die with a die handle: we simply create 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
  ... Your code ...
```

Now we are able to sum a die with a die handle as shown in Figure 1-2. The test `testAddingADieAndHandle` should now pass.

1.11 Stepping back

You may ask why this is working. We defined two methods `sumWithDie:` one on class `Die` and one on the class `DieHandle` and when the 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 `sumWithDie:` for us as shown in Figure 1-3.

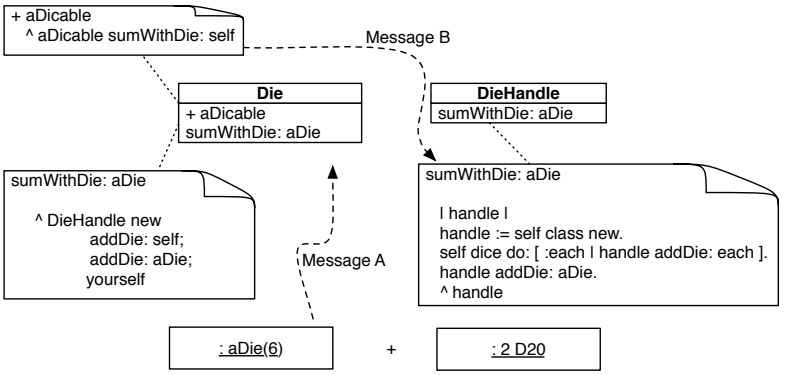

Figure 1-2 Summing a die and a dicable.

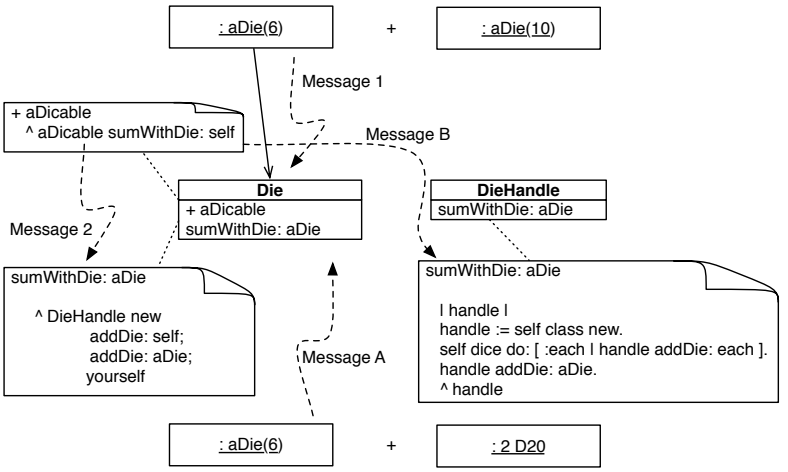

Figure 1-3 Summing a die and a dicable

1.12 Now a DieHandle as receiver

Our solution does not handle the case where the receiver is a die handle. This is what we will address now. Now we are ready to apply the same pattern than before but for the case where the receiver is a die handle. We will just say to the argument of the message + that we want to sum it with a *die handle* this time.

We know how to sum two die handles, it is the code we already defined in the previous chapter. We rename the + method as `sumWithHandle:` to be able to invoke it while redefining the method +. Basically this method creates a new handle, then adds the dice of the receiver and the argument to it and returns the new handle.

```
DieHandle >> sumWithHandle: aDieHandle
 ... Your code ...
```

Now we can define a more powerful version of + by simply sending the message `sumWithHandle:` to the **argument** (aDicable) of the message +. Again we send a message to the argument (`aDicable`) to kick in a new message lookup and dispatch for the message `sumWithHandle:`.

```
DieHandle >> + aDicable
 ... Your code ...
```

We said that this is version of + is more powerful than the one of `sumWithHandle:` because once we will implement the missing method `sumWithHandle:` on the class `Die`, the + method will be able to sum a die handle with a die or two die handles.

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

1.13 sumWithHandle: on Die class

To get the possibility to sum a die handle with a single die, we just have to define a new method `sumWithHandle:` on the `Die` class. The logic is similar to the one adding one die to one die handle

```
Die >> sumWithHandle: aDieHandle
 ... Your code ...
```

Note that we could have sent the message `aDieHandle sumWithDie: self` as body of `sumWithHandle:` definition.

Figure 1-4 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.

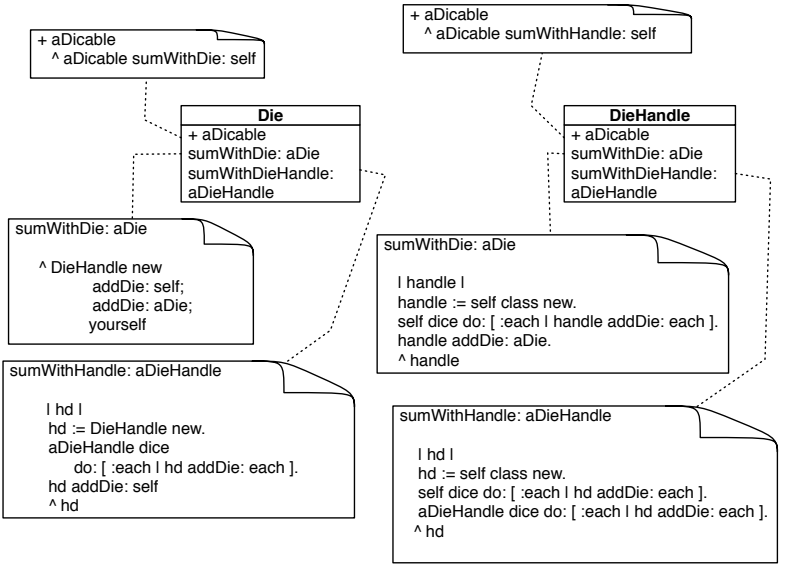

Figure 1-4 Handling all the cases: summing a die/die handle with a die/die handle .

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

1.14 Conclusion

When we step back, we see that we applied the *Don't ask, tell* principle twice: First the message `+` 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 for the messages `sumWithDie:` or `sumWithHandle:`.

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

according to the receiver and the argument of a message.

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.