



**Learning Object-Oriented
Programming and Design with TDD**

Adding a Die and a DieHandle: A Case of Double Dispatch

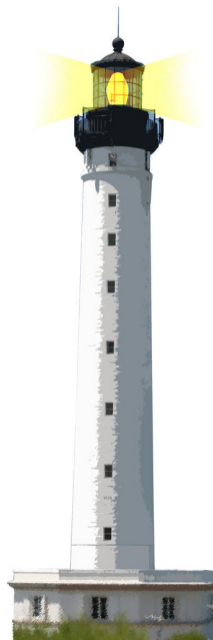
Stéphane Ducasse

<http://stephane.ducasse.free.fr>



<http://www.pharo.org>

W8S01



What You Will Learn

- How conditionals can be turned into extensible design using messages
- Basis for more complex situation such as the Visitor Design Pattern



Remember Die and DieHandle

We create a die handle and add some die to it

```
| handle |  
handle := DieHandle new  
  addDie: (Die withFaces: 6);  
  addDie: (Die withFaces: 10);  
  yourself.  
handle roll
```

Remember DieHandle

We add dieHandle together as in role playing games

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

- We could add dices to a dice handle
- We could add dice handle to another dice handle

New Requirements

We want to add two dices together

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

Now we want to be able to add a dice to an dice handle

(Die withFaces: 6) + 2 D20

2 D20 + (Die withFaces: 6)

aNewRequirement asTest

```
DieTest >> testAddTwoDice
```

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

```
DieTest >> testAddingADieAndHandle
```

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

Propose a solution!



Our approach

- When we add two elements (die or dieHandle) together.

We always do the same:

- we tell **the argument** that we want to add the receiver
- we are explicit about the receiver state since we know it
 - when the receiver is a die we say to the argument that we want to add a die
 - when the receiver is a die handle we say to the argument that we want to add a die handle

Let us do it now!



First adding two dice

```
Die >> + aDie
```

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

Limits

```
Die >> + aDie
```

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

But aDie can be

- a dice
- a die handle



A first step

Adding two dice is usefull, let us keep it and rename it:

```
Die >> sumWithDie: aDie
```

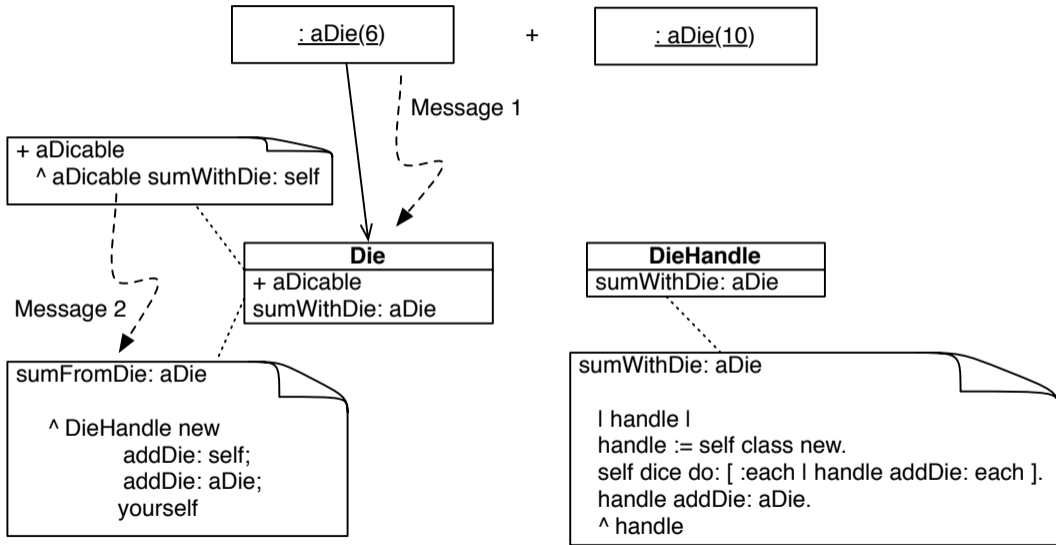
```
^ DieHandle new  
  addDie: self;  
  addDie: aDie; yourself
```

Now we just say to the argument that we want to add a die

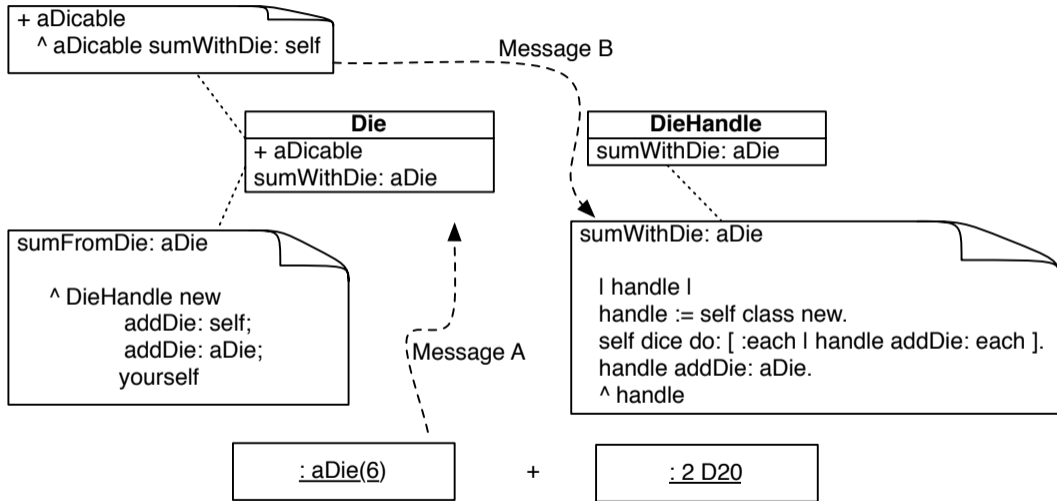
```
Die >> + aDicable  
  ^ aDicable sumWithDie: self
```



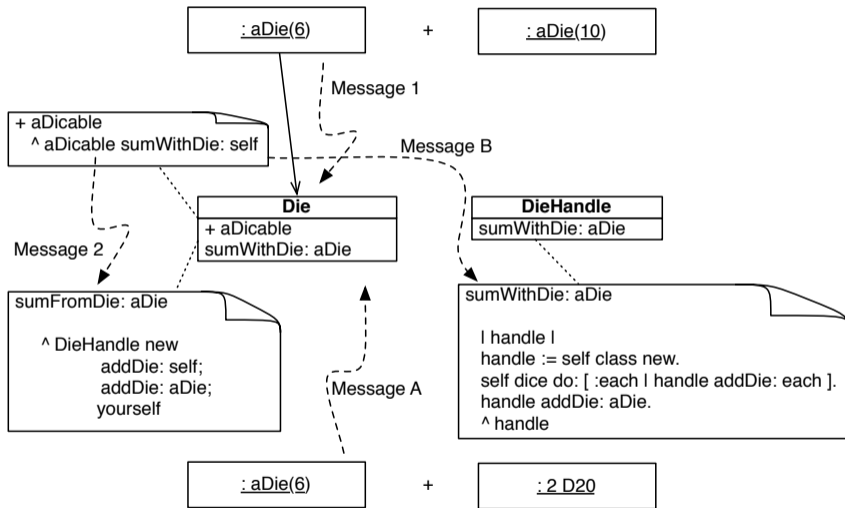
Adding Two Dice and Ready for More



Handling DieHandle as Argument



Sending a Message is Making a Choice



DieHandle as a receiver

We apply the same principle

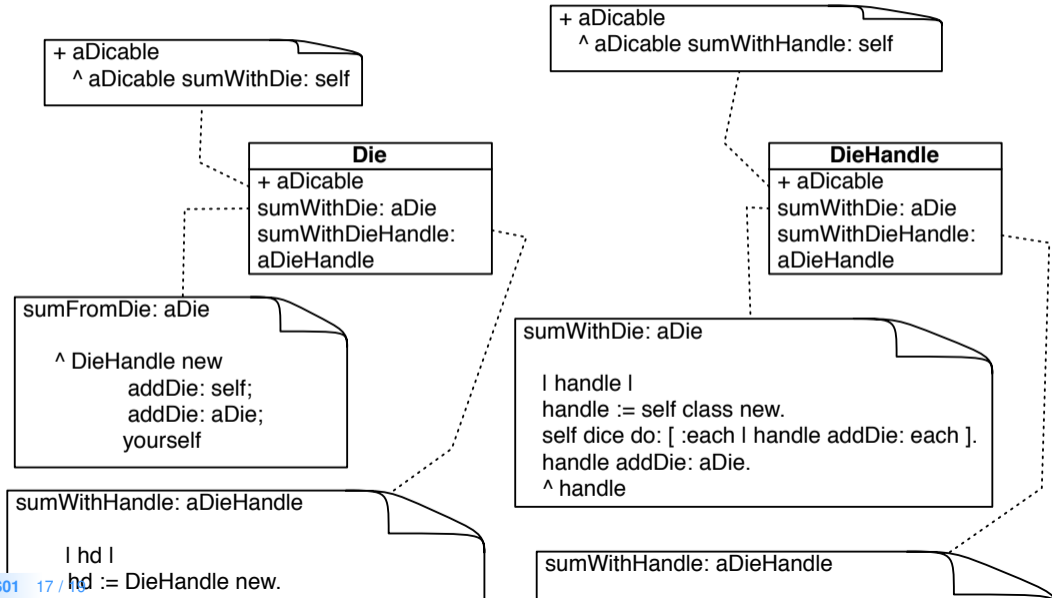
```
DieHandle >> + aDicable  
^ aDicable sumWithHandle: self
```

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

Now the argument can be a die

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


Double Dispatch between Die and DieHandle



Conclusion

- Basis for advanced design such as the Visitor Design Pattern
- Powerful
- Modular (compiler with 70 nodes scales without problems)
- Just sending an extra message to an argument and using late binding once again



A course by Stéphane Ducasse
<http://stephane.ducasse.free.fr>

Reusing some parts of the Pharo Mocc by

Damien Cassou, Stéphane Ducasse, Luc Fabresse
<http://mocc.pharo.org>



Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France
<https://creativecommons.org/licenses/by-nc-nd/3.0/fr/>