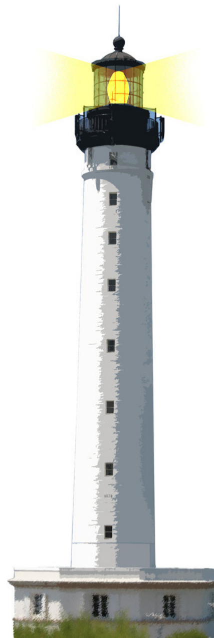
Advanced Object-Oriented Design

a Die + a DieHandle:

Practicing more

S. Ducasse

http://www.pharo.org



Goals

- 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, add some die to it, and roll it.

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

Remember DieHandle

We add dieHandles together as in role playing games

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

New Requirement One

We want to add two dices together and get a DieHandle

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

New Requirements Two

Now we want to be able to add a dice to an dice handle and the inverse

```
(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 equals: 2.
```

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

Possible solution with conditions

```
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
```

Limits of this approach

- It does not scale
- What if we have different objects that should interact with different operations e.g.,
  - different kinds of text objects: list, figures, paragraph, section, title, text, reference...
  - different operations: rendering text, HTML, LaTeX

Hints

- Sending a message is making a choice
  - the system selects the correct method for a given receiver and executes it
- To select a method based on the receiver AND the argument, we have to send a message to the argument

Sketch of the solution

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

For example as in

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

Introducing sumWithDie:

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

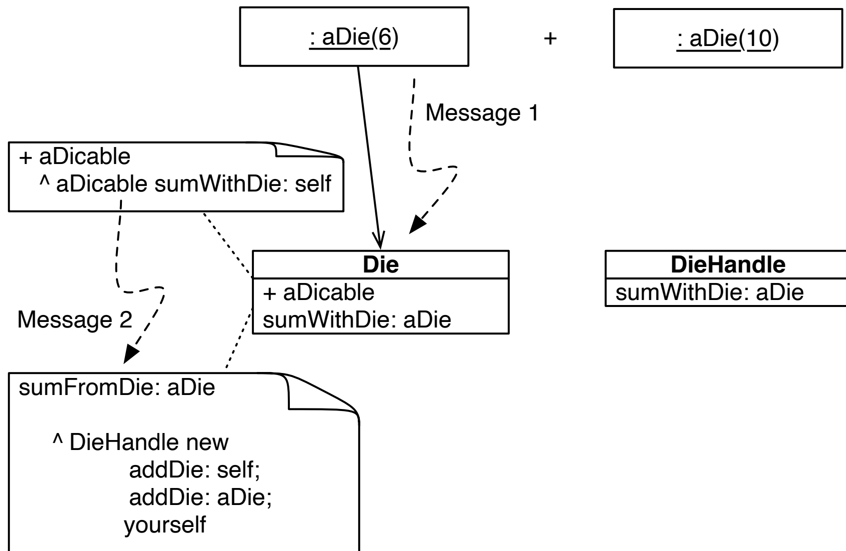
```
Die >> sumWithDie: aDie

  ^ 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

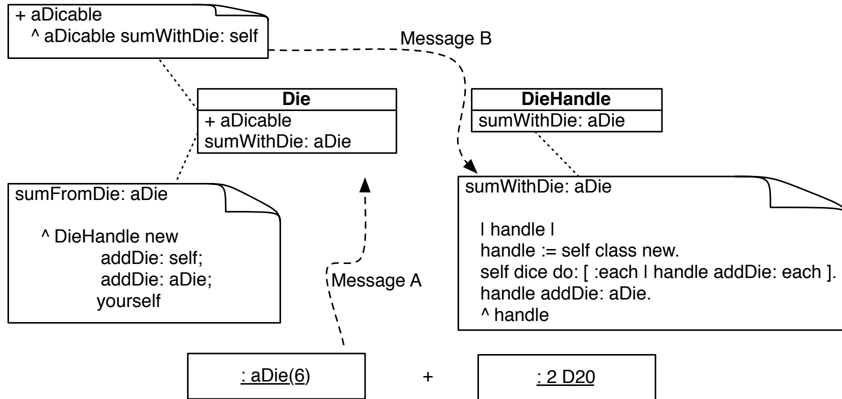


Handling DieHandle as Argument

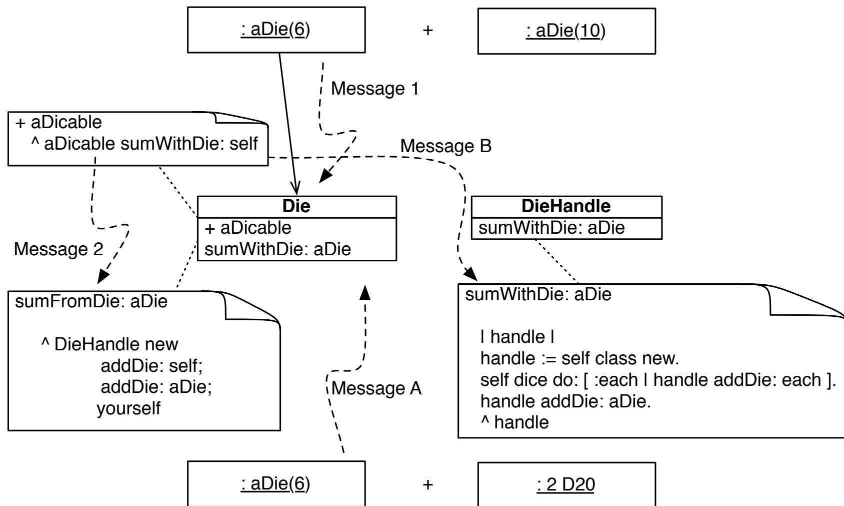
```
(Die withFaces: 6) + 2 D20
```

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

Handling DieHandle as Argument



Sending a Message is Making a Choice



Sending a Message is Making a Choice

- We get two messages/choices
  - one message for +
  - one message for sumWithDie:

DieHandle as a receiver

We apply the same principle

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

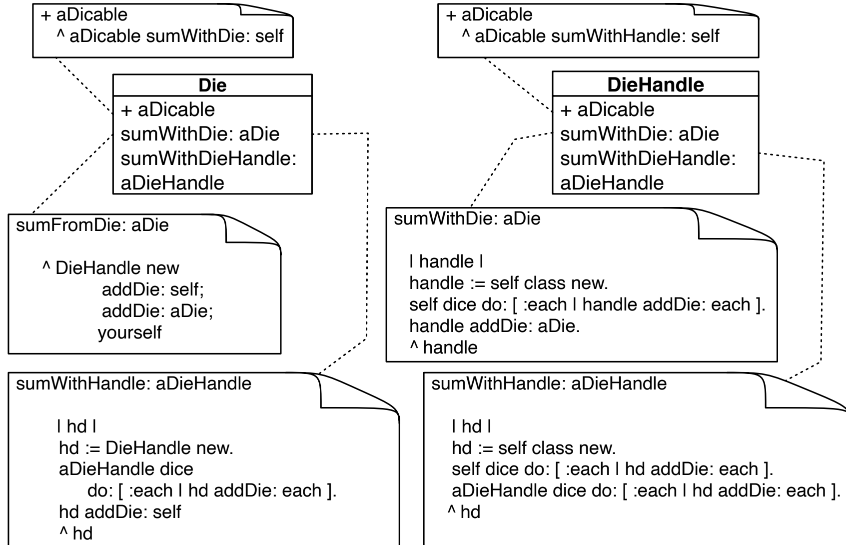
```
DieHandle >> sumWithHandle: aDieHandle
  | handle |
  handle := self class new.
  self dice do: [ :each | handle addDie: each ].
  aDieHandle dice do: [ :each | handle addDie: each ].
  ^ handle
```

Now the argument can be a die

Since the argument can be a die, we define sumeWithHandle also on 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



Stepping back

- We applied two times a simple principle.
  - Sending a message is making a choice/selecting the right method
- So sending a message to the argument is a way to select again between a couple of methods.

Conclusion

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

A course by

S. Ducasse, L. Fabresse, G. Polito, and Pablo Tesone

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/

l'Université numérique
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR, DE LA RECHERCHE ET DE L'INNOVATION
FRANCE RELANCE
FONDATION UNIT