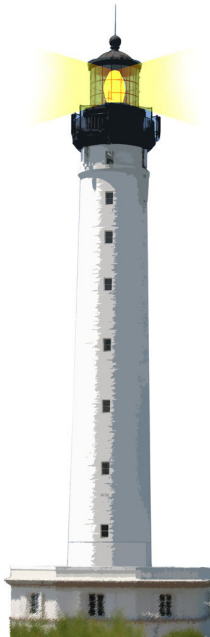
Turning Procedure to Objects

S. Ducasse



Pharo

http://www.pharo.org

Objects are powerful

- Super basic to say it but
- **Objects are powerful**
- An example: Behavior»printHierarchy vs. ClassHierarchyPrinter
  - printHierarchy is a method
  - ClassHierarchyPrinter is a little class

Functionality we want

- Printing the hierarchy of class

```
>>> Rectangle printHierarchy
'ProtoObject #()
  Object #()

    Rectangle #(#origin #corner)
      CharacterBlock #(#stringIndex #text #textLine)'
```

Coded as...

```
Behavior >> printHierarchy
  "Answer a description containing the names and instance variable names
  of all of the subclasses and superclasses of the receiver."

  | aStream index |
  index := 0.
  aStream := (String new: 16) writeStream.
  self allSuperclasses reverseDo:
    [:aClass |
    aStream crtab: index.
    index := index + 1.
    aStream nextPutAll: aClass name.
    aStream space.
    aStream print: aClass instVarNames].
  aStream cr.
  self printSubclassesOn: aStream level: index.
  ^aStream contents
```

With...

```
Behavior >> printSubclassesOn: aStream level: level
    "As part of the algorithm for printing a description of the receiver, print the
    subclass on the file stream, aStream, indenting level times."

    | subclassNames |
    aStream crtab: level.
    aStream nextPutAll: self name.
    aStream space; print: self instVarNames.
    self == Class
      ifTrue:
        [aStream crtab: level + 1; nextPutAll: '[ ... all the Metaclasses ... ]'.
        ^self].
    subclassNames := self subclasses asSortedCollection:[:c1 :c2| c1 name <= c2 name].
    "Print subclasses in alphabetical order"
    subclassNames do:
      [:subclass | subclass printSubclassesOn: aStream level: level + 1]
```

Analyis

- **Procedural** decomposition
- **Simple** (two methods)
- State is passed as arguments

Does not work if we need

- To filter subclasses (RBLintRule printHierarchy)
- To cut above a given superclass or if class is from a given package
- Do not want to see instance variables

Limits

- End up with **too many** arguments
- We cannot design a fluid API to **configure** the output
- We may not want or **cannot** add state to the domain object
  - here we do not want to add state to Behavior just for printing

Turning it into an object

We can simply do

```
ClassHierarchyPrinter new
  forClass: Rectangle;
  doNotShowState;
  doNotShowSuperclasses
```

A more complex scenario

```
ClassHierarchyPrinter new
  forClass: RBNode;
  doNotShowState;
  doNotShowSuperclasses;
  excludedClasses: (RBNode withAllSubclasses
          select: [ :each | each name beginsWith: RBPattern ]);
  limitedToClasses: (RBNode withAllSubclasses
            select: [:each | each name beginsWith: RB]).
```

Looking at ClassHierarchyPrinter

```
Object << #ClassHierarchyPrinter
  slots: { #theClass . #excludedClasses . #limitedToClasses . #stream .
  #level . #showSuperclasses . #showState };
  tag: 'ForPharo';
  package: 'Kernel-ExtraUtils'
```

API

- doNotShowState, doNotShowSuperclasses
- limitedToClasses: to offer specific scope
- excludedClasses: to remove unwanted subclasses
- cr, tab, nextPutAll: to let decorations

Stepping back

- Created **little objects** that can be configured!
- The object holds the **specific state** for its computation
- The API can extended if needed
- Functionality can be removed from Behavior
- Functionality can be **nicely tested** and packaged outside of Kernel

Further thought

- An object is a **powerful** entity
- If the functionality needs to behave differently on different objects (Classes, Traits, Metaclasses...) we can use double dispatch between the objects that the printers as it is done in the ClassDefinitionPrinter

A course by

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