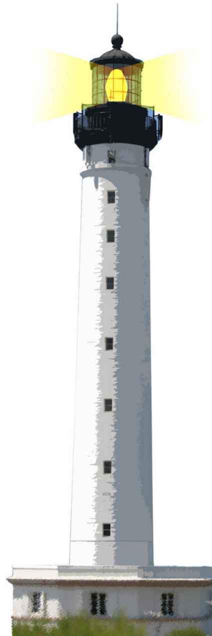
Turning Procedures to Objects

S. Ducasse



http://www.pharo.org

Goals/Objective

Super basic to say it but

- **Objects are really powerful**
- Basic for behavior reification
- An example: Behavior»printHierarchy vs. ClassHierarchyPrinter
  - printHierarchy is a method
  - ClassHierarchyPrinter is a little class

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

Pros

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

Limits

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
- End up with **too many** arguments

```
printSubclassesOn: aStream level: level filtering: aCol cut: above showVariable: bool
```

- We may not want or **cannot** add state to the domain object
  - here we cannot add state to Behavior just for printing
- We cannot design a fluid API to **configure** the output

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

DatePrinter vs. Date printOn:

Different date formats

```
10 Janvier 2023
10 Jan 23
10 / 01 / 2023
10−01−2023
01 / 10 / 2023
```

- Should `printOn:` handles all this?
- `mmddyyyy`, `ddmmyyyy` support limited scenario
- Printing a date can be the responsibility of a specific object

ZTimestampFormat

The class ZTimestampFormat is a nice example of reifying the complex

- I am ZTimestampFormat, an implementation of a textual representation for a timestamp, date or time that can be used for formatting or parsing.

ZTimestampFormat

Provides a template to shape the output

```
(ZTimestampFormat fromString: 'SAT, FEB 03 2001 (16:05:06)')
  format: ZTimestamp now.
> 'FRI, OCT 28 2022 (06:43:11)'
```

ZTimestampFormat

Another behavior: parsing date according to a template

```
(ZTimestampFormat fromString: '02/03/2001 (16:05:06)')
  parse: '10/05/2022 (12:01:01)'.
> 2022−10−05T12:01:01Z
```

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 **nicely tested** and packaged in autonomous manner

Further thought

- Turning a method into an object is the key point of Command Design pattern
- This is also the case in Visitor
- An object is a **powerful** entity

A course by

S. Ducasse, 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/