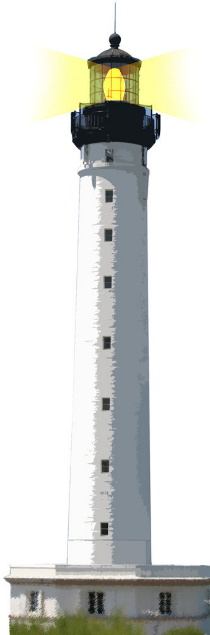
Advanced Object-Oriented Design

# Hooks and Template

One of cornerstone of OOP

S. Ducasse



http://www.pharo.org

## Goal/Outline

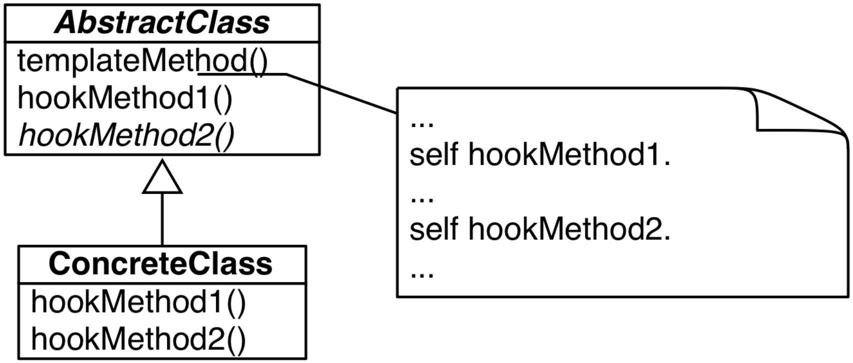
- Hook and Template
- printString/printOn: case
- copy case

## Remember...

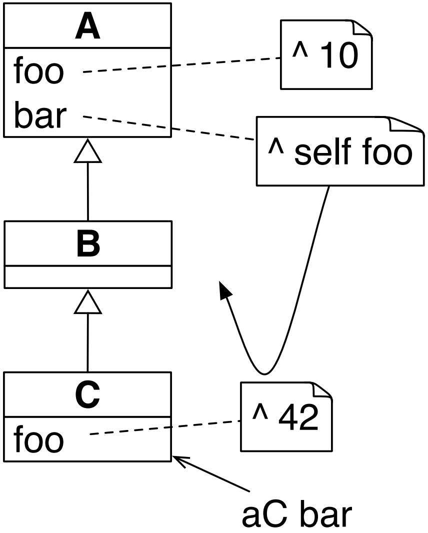
- Sending a message is making a choice
- A class defines one possible choice
- Self-sends are plans for reuse (e.g., defines a hook):
  - i.e., a place where subclasses can inject variations

## The template method

- A template method specifies a skeleton with hooks
- Hooks are places to be customized by subclasses
- Hooks may or may not have a default behavior



## Principle



## Studying the printString template method

Example of printString

```
>>> (Delay forSeconds: 10) printString
'a Delay(10000 msecs)'
```

# printString template method

```
Object >> printString
  "Answer a String whose characters are a description of the receiver."
  ^ self printStringLimitedTo: 50000
```

```
Object >> printStringLimitedTo: limit
  | limitedString |
  limitedString := String
                        streamContents: [ :s | self printOn: s ]
                        limitedTo: limit.
  limitedString size < limit ifTrue: [ ^ limitedString ].
  ^ limitedString , '...etc...'
```

## printOn: A default hook

```
>>> Node new printString
a Node
```

```
>>> Apple new printString
an Apple
```

Default behavior:

```
Object >> printOn: aStream
  "Append to the argument, aStream, a sequence of characters that identifies the
      receiver."
  | title |
  title := self class name.
  aStream
    nextPutAll: (title first isVowel ifTrue: [ 'an ' ] ifFalse: [ 'a ' ]);
    nextPutAll: title
```

## Hook refinement

```
>>> (Delay forSeconds: 1) printString
a Delay(1000 msecs)
```

Reusing and extending default behavior:

```
Delay >> printOn: aStream
  super printOn: aStream.
  aStream
    nextPutAll: '(';
    print: millisecondDelayDuration;
    nextPutAll: ' msecs)'
```

## Hook redefinition

```
>>> true not printString
false
```

Redefinition in False:

```
False >> printOn: aStream
  aStream nextPutAll: 'false'
```

## Hook redefinition

```
>>> 1 to: 100
(1 to: 100)
>>> 1 to: 100 by: 3
(1 to: 100 by: 3)
```

Redefinition in Interval:

```
Interval >> printOn: aStream
  aStream
    nextPut: $(;
    print: start;
    nextPutAll: ' to: ';
    print: stop.
  step ~= 1
    ifTrue: [ aStream nextPutAll: ' by: '; print: step ].
  aStream nextPut: $)
```

## Another template method: Object copy

Copying objects is complex:

- graph of connected objects
- cycles
- each class may want a different copy strategy

A simple solution for simple cases: `copy/postCopy`

## Object » copy

```
Object >> copy
 "Answer another instance just like the receiver.
 Subclasses typically override postCopy.
 Copy is a template method in the sense of Design Patterns.
 So do not override it. Override postCopy instead. P
 ay attention that normally you should call postCopy of your superclass too."

  ^ self shallowCopy postCopy
```

```
Object >> shallowCopy
 "Answer a copy of the receiver which shares the receiver's instance variables.
     Subclasses that need to specialize the copy should specialize the postCopy hook
     method."
 <primitive: 148>
 ...
```

## Default hook

```
Object >> postCopy
  "I'm a hook method in the sense of Design Patterns Template/Hook Method.
  I'm called by copy.
  self is a shallow copy, subclasses should copy fields as necessary to
  complete the full copy"

  ^ self
```

## postCopy: Refinement

```
Collection subclass: #Bag
  instanceVariableNames: 'contents'
  classVariableNames: ''
  package: 'Collections-Unordered'
```

```
Bag >> postCopy
  super postCopy.
  contents := contents copy
```

- contents is a Dictionary
- postCopy recursively invoke copy on the dictionary

## Dictionary » postCopy: Deeper copy

```
Dictionary >> postCopy
  "Must copy the associations, or later store will affect both the original and the copy"
  array := array
          collect: [ :association |
                  association ifNotNil: [ association copy ] ]
```

## Conclusion

- Hooks and Template is a very common design pattern
- Template method sets the context
- Hooks specify variations
- A self-send message defines a hook
- Sending a message to another object opens space for dispatch (see Strategy Design lecture)

A course by

S. Ducasse, G. Polito, and Pablo Tesone