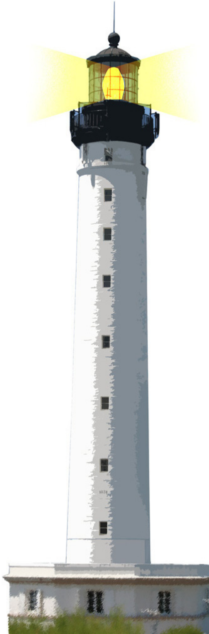
Advanced Object-Oriented Design

Hooks and Templates

An application of self-sends are plans for reuse

S. Ducasse and L. Fabresse



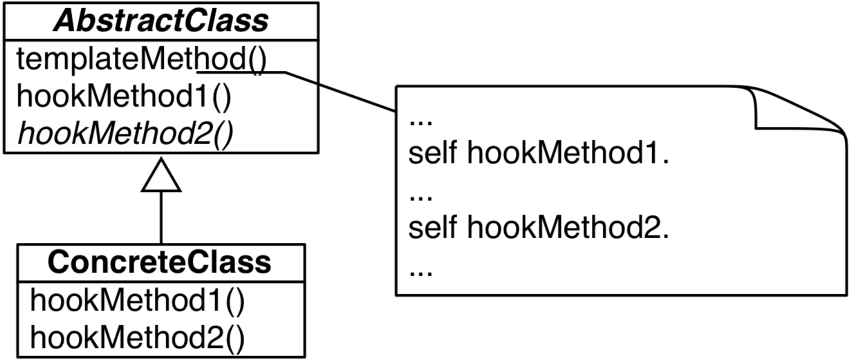
http://www.pharo.org

Remember...

- A message send leads to a choice
- A class hierarchy defines the choices
- Code can be reused and refined in subclasses
- Sending a message in a class defines a hook:
  - i.e., a place where subclasses can inject variations

The Template Method

- a template method specifies a skeleton
- the skeleton has hooks, i.e., places to be customized
  - hooks may or may not have a default behavior



AbstractClass

templateMethod()
hookMethod1()
hookMethod2()

ConcreteClass

hookMethod1()
hookMethod2()

```
...
self hookMethod1.
...
self hookMethod2.
...
```

printString: A Template Method

```
(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...'
```

A Default Hook: printOn:

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

```
Apple new
>>> 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
```

printOn: Refinement

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

Reusing and extending default behavior:

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

printOn: Redefinition

```
true not
> false
```

Redefinition in False:

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

printOn: 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
- 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. Pay 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 TemplateHook/Methods. 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 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

- Template Method is a very common design pattern
- Sending a message defines a hook
- Sending a message increases potential reuse

A course by

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