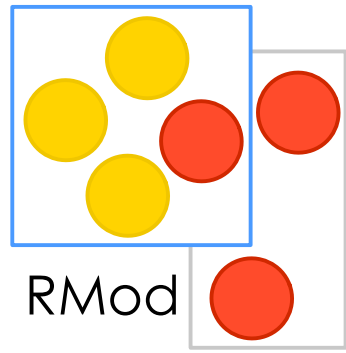


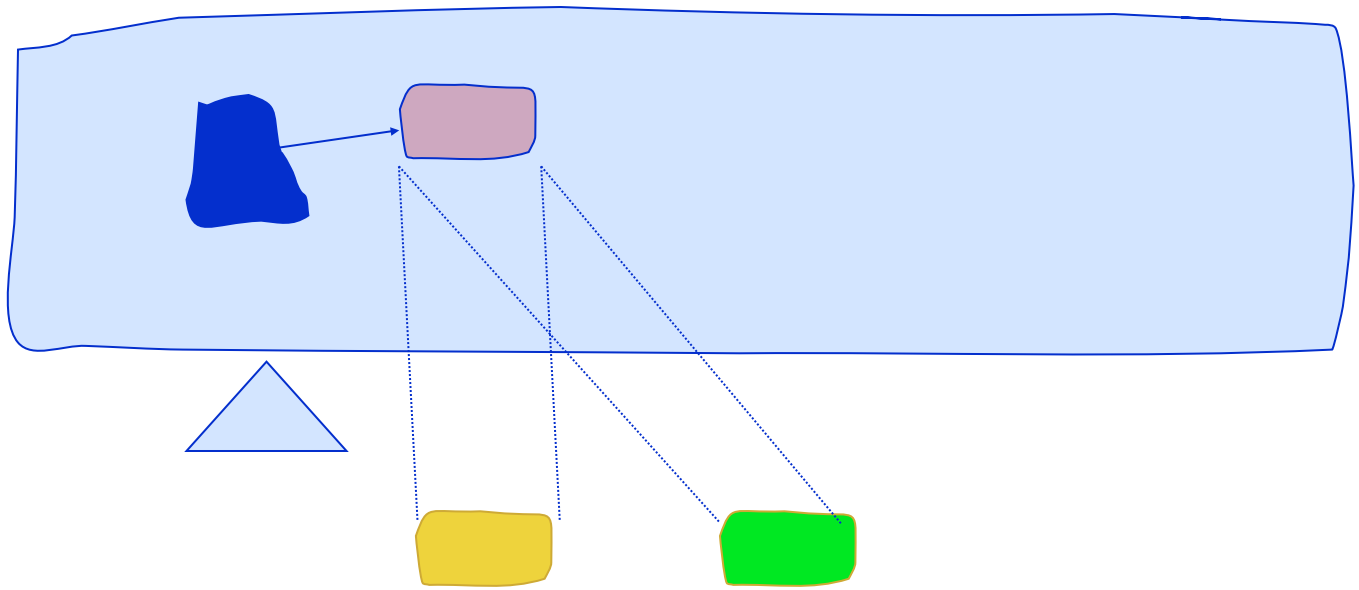
Elements of Design - Plans for Reuse

Stéphane Ducasse
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr/

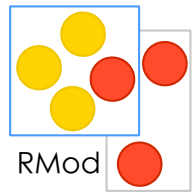
Plans for Reuse

Dynamic binding + methods
= reuse in subclasses
= specialisation in subclasses

Methods are Unit of Reuse

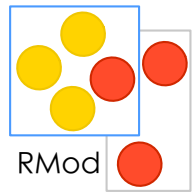


A first problem




```
Node>>computeRatioForDisplay
    | averageRatio defaultNodeSize |
    averageRatio := 55.
    defaultNodeSize := mainCoordinate /maximiseViewRatio.
    self window add:
            (UINode new with:
                    (bandWidth * averageRatio / defaultWindowSize)
    …
```

Forced to Duplicate!

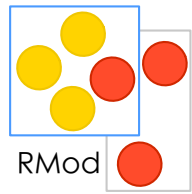



```
Node>>computeRatioForDisplay
        | averageRatio defaultNodeSize |
        averageRatio := 55.
        defaultNodeSize := mainCoordinate /maximiseViewRatio.
        self window add:
                (UINode new with:
                        (bandWidth * averageRatio / defaultWindowSize)
        …
```

- We are *forced* to *copy* the complete method!

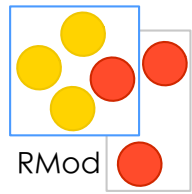
```
SpecialNode>>computeRatioForDisplay
        |averageRatio defaultNodeSize|
        averageRatio := 55.
        defaultNodeSize := mainCoordinate + minimalRatio /
        maximiseViewRatio.
        self window add:
                (UINode new with: (self bandWidth * averageRatio /
        defaultWindowSize)
```

Solution



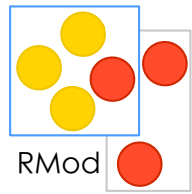
Define methods
Send messages to these methods
Let subclasses customize such methods

Self sends: Plan for Reuse




```
Node>>computeRatioForDisplay
        | averageRatio defaultNodeSize |
        averageRatio := 55.
       defaultNodeSize := self defaultNodeSize.
       self window add: (UINode new with: (bandWidth *
   averageRatio / defaultWindowSize)

Node>>defaultNodeSize
       ^ mainCoordinate / maxiViewRatio
SpecialNode>>defaultNodeSize
       ^ mainCoordinate+minimalRatio/maxiViewRatio
```

Do not Hardcode Class Names

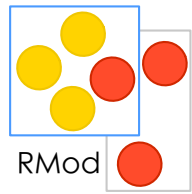



```
Node>>computeRatioForDisplay
        |averageRatio defaultNodeSize|
        averageRatio := 55.
        defaultNodeSize := mainWindowCoordinate / maximiseViewRatio.
        self window add:
                (UINode new with:
                        (bandWidth * averageRatio / defaultWindowSize).
        ...
```

- We are forced to copy the method!

```
SpecialNode>>computeRatioForDisplay
        |averageRatio defaultNodeSize|
        averageRatio := 55.
        defaultNodeSize := mainWindowCoordinate / maximiseViewRatio.
        self window add:
                (ExtendedUINode new with:
                        (bandWidth * averageRatio /
```

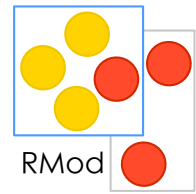
Class Factories



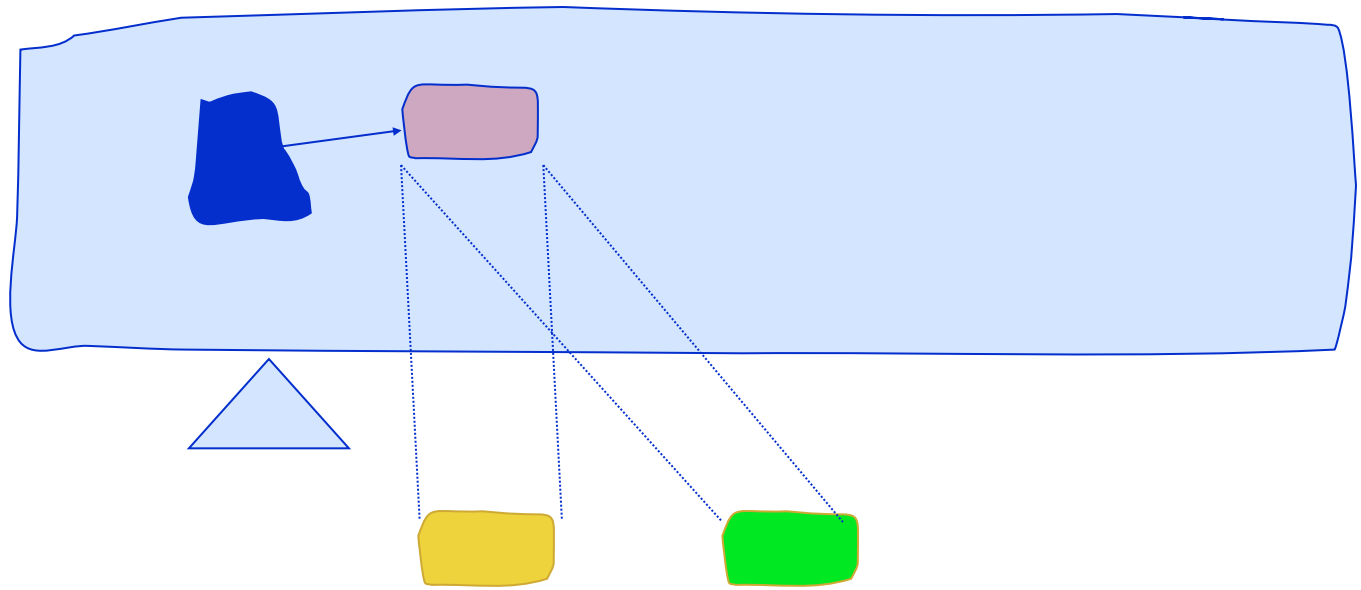

```
Node>>computeRatioForDisplay
    | averageRatio |
    averageRatio := 55.
    self window add:
            self UIClass new with:
                    (self bandWidth * averageRatio / self
        defaultWindowSize)
        ...

Node>>UIClass
    ^ UINode

SpecialNode>>UIClass
    ^ ExtendedUINode
```

Hook and Template

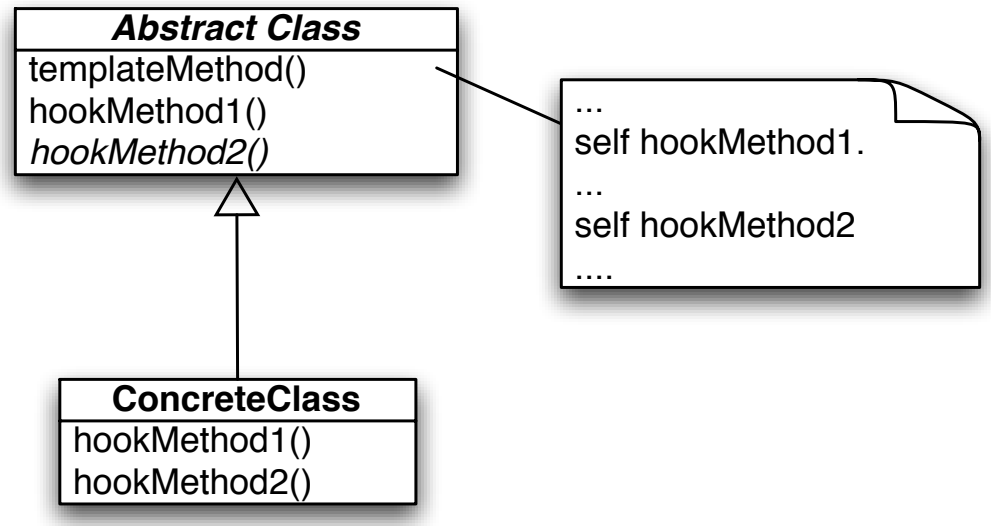
Hook and Template Methods



- Hooks: place for reuse
- Template: context for reuse

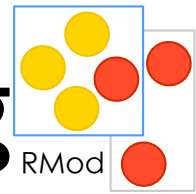


Hook and Template Methods



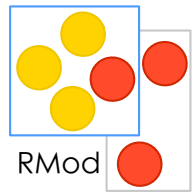
- *Templates:* Context reused by subclasses
- *Hook methods:* holes that can be specialized
- Hook methods do not have to be abstract, they may define default behavior or no behavior at all.

Hook / Template Example: Printing



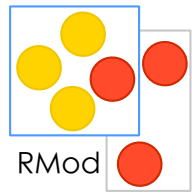

```
Object>>printString
    "Answer a String whose characters are a description of the
    receiver."

    | aStream |
    aStream := WriteStream on: (String new: 16).
    self printOn: aStream.
    ^aStream contents
```

Hook



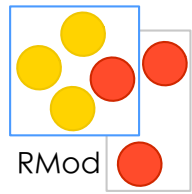

```
Object>>printOn: aStream
  "Append to the argument aStream a sequence of characters
            that describes the receiver."

  | title |
  title := self class name.
  aStream nextPutAll:
            ((title at: 1) isVowel ifTrue: ['an '] ifFalse: ['a ']).
  aStream print: self class
```

Overriding the Hook



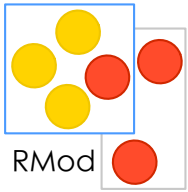

```
Array>>printOn: aStream
    "Append to the argument, aStream, the elements of the Array
    enclosed by parentheses."

    | tooMany |
    tooMany := aStream position + self maxPrint.
    aStream nextPutAll: '#('.
    self do: [:element |
                aStream position > tooMany
            ifTrue: [ aStream nextPutAll: '...(more)...)'.
                ^self ].
                 element printOn: aStream]
        separatedBy: [aStream space].
```

Overriding




```
False>>printOn: aStream
        "Print false."

        aStream nextPutAll: 'false'
```

Specialization of the Hook



The class ***Behavior*** that represents a class extends the default hook but still invokes the default one.

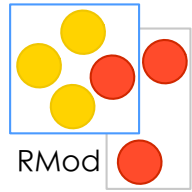
Behavior>>***printOn:*** aStream

"Append to the argument aStream a statement of which

superclass the receiver descends from."

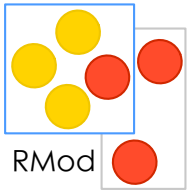
aStream nextPutAll: 'a descendent of '.
superclass ***printOn:*** aStream

Another Example: Copying



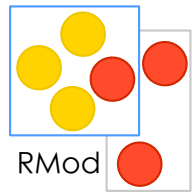
Complex (deepCopy, veryDeepCopy...)
Recursive objects
Graph of connected objects
Each object wants a different copy of itself
No up-front solution

The copy hook/template



Smallest template method ever (2 messages)
three messages: copy, shallowCopy, postCopy

Hook Example: Copying



Object>>copy

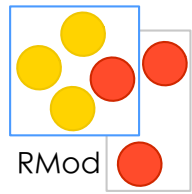
" Answer another instance just like the receiver. Subclasses normally override the postCopy message, but some objects that should not be copied override copy. "

^self shallowCopy ***postCopy***

Object>>shallowCopy

"Answer a copy of the receiver which shares the receiver's instance variables."

postCopy

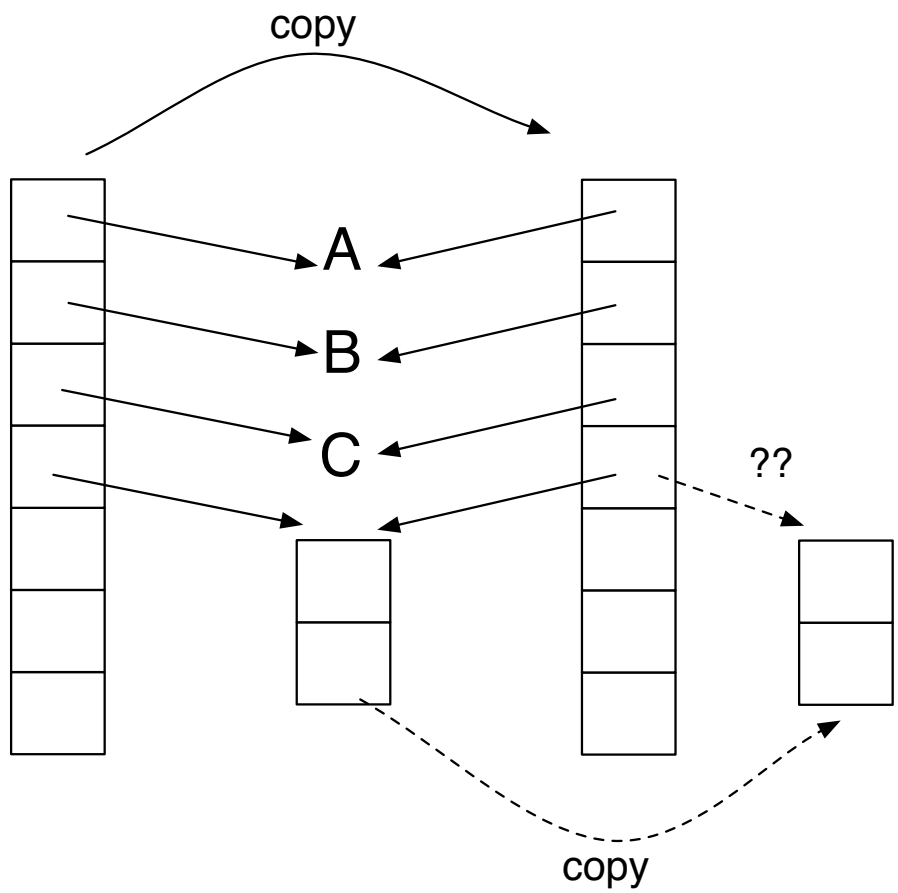


Object>>postCopy

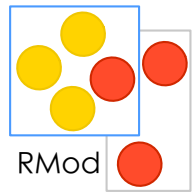
"Finish doing whatever is required, beyond a shallowCopy, to implement 'copy'. Answer the receiver. This message is only intended to be sent to the newly created instance. Subclasses may add functionality, but they should always do super postCopy first. "

^self

Sounds Trivial?



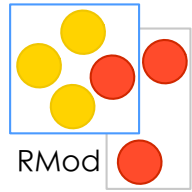
Stuff>>copy



We do not want to share the fourth element

```
Stuff>>postCopy
        super postCopy.
        fourth := fourth copy
```

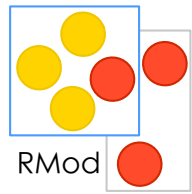
Analysis



Recursive invocation of copy

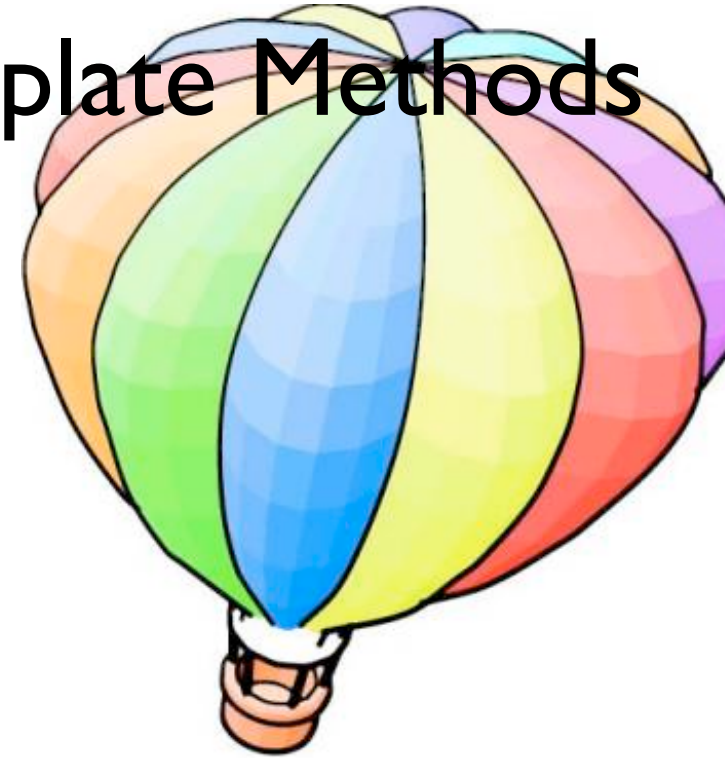
we do not know how the subelement wants to be copied

Hook Specialisation




```
Bag>>postCopy
        "Make sure to copy the contents fully."

        | new |
        super postCopy.
        new := contents class new: contents capacity.
        contents keysAndValuesDo:
            [:obj :count | new at: obj put: count].
         contents := new.
```

Guidelines for Creating Template Methods



Simple implementation.
- Implement all the code in one method.

Break into steps.
- Comment logical subparts

Make step methods.
- Extract subparts as methods

Call the step methods

Make constant methods, i.e., methods doing nothing else than returning.

Repeat steps 1-5 if necessary on the methods created