

Meta-Matters in Squeak

Andrew P. Black

What's Meta?

- Metaprogramming is the act of writing a program that writes or manipulates another program... or itself
- Why not? After all programs are just data!

Example: named colors

The screenshot shows the Squeak System Browser. On the left, the class hierarchy for Color is visible, including ColorNames, Color, and ColorMap. On the right, the ColorNames class is shown with a class variable 'ColorNames' containing a list of color names: red, green, blue, yellow, cyan, magenta, black, white, gray, brown, purple, pink, orange, olive, gold, silver, steelblue, darkslategray, firebrick, darkred, darkgreen, darkblue, darkcyan, darkmagenta, darkgray, darkolivegreen, darkslateblue, darkred, darkgreen, darkblue, darkcyan, darkmagenta, darkgray, darkolivegreen, darkslateblue. The Color class is shown with a class method 'newColor:put:'.

Named Colors (cont)

- Each named color, e.g., yellow
 - should have a class method, so that we can write *Color yellow*
 - should be in the collection *ColorNames*, so that the *name* method works
 - should have a corresponding class variable, e.g., *Yellow*, whose value is the right rgb triple
- How can we make sure that these invariants hold?
- Metaprogramming!

The screenshot shows the Squeak System Browser. On the left, the class hierarchy for Color is visible, including ColorNames, Color, and ColorMap. On the right, the ColorNames class is shown with a class variable 'ColorNames' containing a list of color names: red, green, blue, yellow, cyan, magenta, black, white, gray, brown, purple, pink, orange, olive, gold, silver, steelblue, darkslategray, firebrick, darkred, darkgreen, darkblue, darkcyan, darkmagenta, darkgray, darkolivegreen, darkslateblue. The Color class is shown with a class method 'newColor:put:'.

Constructing the Color Names

The screenshot shows the Squeak System Browser. On the left, the class hierarchy for Color is visible, including ColorNames, Color, and ColorMap. On the right, the Color class is shown with a class method 'named:put:'. The method implementation is as follows:

```

named:put:
    "Add a new color to the list and create an access message and a class variable for it. The name should start with a lowercase letter. (The class variable will start with an uppercase letter.) (Color colorNames) returns a list of all color names."
    | str obj sym accessor cymal
    (color isKindOf: self) ifFalse: [self error: 'not a Color'].
    str := newName asString.
    sym := str asSymbol.
    cymal := str capitalized.
    accessor := obj asSymbol.
    "define access message"
    self class compile: accessor
    self class compile: accessor
    classVars: 'named colors'.
    (self classPool includesKey: cymal) ifFalse: [
        self addClassVarName: cymal.
        ColorNames includes: sym] ifFalse: [
            ColorNames add: sym].
    self classPool at: cymal put: cColor
    
```

Solution (continued)

2. Every concrete class *Foo* in the Expression hierarchy gets a method *accept: aVisitor* defined as follows:

```

Foo >> accept: aVisitor
    ↑ aVisitor visitFoo: self
    
```

- Note how the selector of the message tells the visitor what kind of node it is visiting
- Do this for *Foo* = Difference, Product, Quotient, Sum, etc.

I wrote these methods with a metaprogram:

```

Expression allSubclassesDo: [ :each | each compile: 'accept: aVisitor
    ↑ aVisitor visit', (each name), ': self' classified: 'visiting' ]
    
```

Alternative Solution

- Instead of writing a separate program to write our program, we could make the program write itself:

- Put the following single method at the root of the hierarchy:

```
Expression >> accept: aVisitor  
↑ aVisitor perform: ('visit', (self class name), ':') asSymbol  
with: self
```

- This is a reflective program — one that writes *itself* dynamically

Example Problem

- suppose that you want to do some action before and after every method on an object

e.g.,

```
OrderBean >> orderNumber  
↑ orderNumber
```

becomes

```
OrderBean >> orderNumber  
logger logSendOf: #orderNumber.  
result := orderNumber.  
logger logAnswerOf: #orderNumber as: result.  
↑ result
```

Solution: a Wrapper Object

- Define a class BeanWrapper with the following methods:

```
doesNotUnderstand: aMessage  
"Do logging and forward message"  
↑(tracedObject respondsTo: aMessage selector)  
ifTrue: [self pvtDoAround: aMessage]  
ifFalse: [super doesNotUnderstand: aMessage]
```

```
pvtDoAround: aMessage  
| result |  
logger logSendOf: aMessage.  
[↑result := aMessage sendTo: tracedObject]  
ensure: [logger logAnswerOf: aMessage as: result]
```

Deploying the wrappers

- Wrappers can be deployed selectively on some particular Bean objects:

```
b := OrderBean new.  
w := BeanWrapper wrap: b.
```

- Or, they can be deployed on *every* Bean

```
Bean >> new  
↑ BeanWrapper wrap: super new
```

- re-defining `new` is itself a form of metaprogramming

Another Example

- We know that we can write this:
`(1 to: 10) select: [:x | x even]`
- How about this?
`(1 to: 10) select even`
- Can we make this work? What about other unary messages (`odd`, `isPrime`, ...)?

Summary of Solution

- (1 to: 10) `select` must answer an object that "remembers" the collection and the fact that we plan to do a `select:` operation
 - This object is called a *Trampoline*
- How can we make the trampoline understand `even`, `odd`, `isPrime`, `factorial` ...
 - Reflection!

Structural Equality

- We saw how to build a recursive equality operation in Haskell that reaches down into the structure of a data type
- Can we do the same in Squeak?
 - How is equality defined in *Object*?

Try a new Equality Operation

```

RB: Object
Kernel-Objects: Boolean
Kernel-Objects-Tests: DependentsArray
Kernel-Processes: False
Kernel-Processes-Tests: MessageSend
Kernel-ST88: Remnants: Model
M: Installer: MorphObjectOut
Monticello-Base: Object
Monticello-Chunk: Form: ObjectOut
Monticello-Loading: ObjectTracer
Monticello-Merging: ObjectFilter
Monticello-Merks: ObjectFilter
...
*tools-browser: closeTo:
Breakpoint: hash
accessing: hashMappedBy:
associating: identifyHashPrint5Str:
binding: literalEqual:
casing: ==
class membership: ==
comparing: ==
converting: ==
copying: ==
creating: ==
...
== anObject
"Answer whether the receiver and the argument have the same values and the same structure."
| ninstanceVars nindexedVars |
ninstanceVars := self class instSize.
nindexedVars := self basicSize.
anObject class instSize = ninstanceVars ifFalse: [ ^ false ].
anObject basicSize = nindexedVars ifFalse: [ ^ false ].
1 to: ninstanceVars do:
[ :i | (self instVarAt: i) == (anObject instVarAt: i) ifFalse: [ ^ false ]. ]
1 to: nindexedVars do:
[ :i | (self basicAt: i) == (anObject basicAt: i) ifFalse: [ ^ false ]. ]
^ true
    
```

How does == work out?

What about zipAllWith: ?

- We would like to be able to write


```
{ $a to: $z . $A to: $Z } zipAllWith:
[ :lo :up | String with: lo with: up ]
```

 for n collections and any n argument block
- Can we do it?

Smalltalk Browsers

- There are *lots* of different browsers in the Smalltalk environment
 - system browser, hierarchy browser, protocol browser, inheritance browser, ... inspector, explorer, change set browser, file system browser
- Each one “knows” about the structure that it is browsing
 - *e.g.*, the system browser has hardwired into its code the facts that Categories contain Classes and Classes contain Protocols and Protocols contain methods

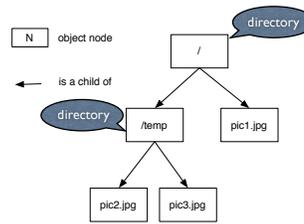
The OmniBrowser

- The OmniBrowser is a browser for everything and nothing in particular
 - it doesn’t “know” about any system structure
 - instead, it is driven by metadata that describes the thing that it is browsing
- The metadata takes the form of a graph of objects — the metagraph
- The domain that the browser navigates is also a graph of objects — the subject graph

A File System Browser

- We will build an instance of the OmniBrowser that examines the file system
- The file system is *not* a graph of objects
- That's OK: we build *OBNodes* to represent the entities that we are browsing
- We define two subclasses of *OBNode*: *OBDirectoryNode* and *OBFileNode*
- What do these *OBNodes* have to do?
 - that is defined by the metagraph

File System: Graph & Metagraph



Metagraph as data

```

fileBrowser
  dir := OBMetaNode named: Directory.
  file := OBMetaNode named: File.
  dir
    childAt: #directories put: dir;
    childAt: #files put: file.
  ^ dir
    
```