

CS410/510 Advanced Programming

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 System Browser interface for the `Color` class. The left pane shows the class hierarchy with `Color` selected. The middle pane shows the class's superclass, `other`, and its subclasses, including `colorForInsets`, `display`, `name`, `raisedColor`, and `rgbTriplet`. The right pane shows the source code for the `name` method.

System Browser: Color

Graphics-Primitives
Graphics-Text
Graphics-Transformation
GraphicsTests-Files
GraphicsTests-Primitives
GraphicsTests-Text
ImageForDevelopers
Installer-Core
Kernel-Chronology
Kernel-Classes

BitBlt
Bitmap
Color
ColorMap
Pen
PenPointRecorder
Point
Quadrangle

equality
groups of shades
html
other
printing
queries
self evaluating
transformations
private

colorForInsets
display
name
raisedColor
rgbTriplet

jm 12/4/97 10:24 · other · 108 implementors · in no change set ·

browse senders implementors versions inheritance hierarchy inst vars class vars source R

name
"Return this color's name, or nil if it has no name. Only returns a name if it exactly matches the named color."

ColorNames do:
[:name | (Color perform: name) = self ifTrue: [+ name]].
+ nil

inst vars class vars source R

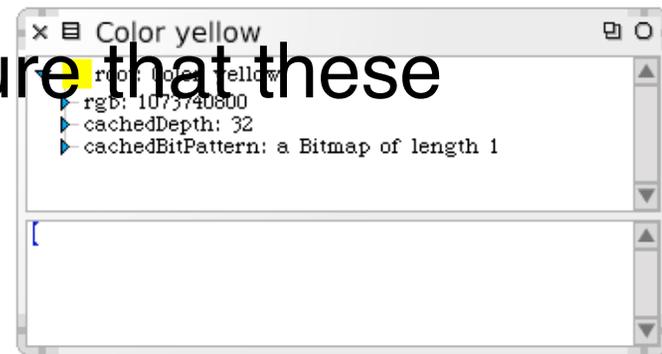
ColorChart ColorNames
en GreenShift HalfComponentMask
ay LightGreen LightMagenta
PaleBuff PaleGreen PaleMagenta
reGreen PureMagenta PureRed
VeryDarkGray VeryLightGray

will be shown in. At the very
h of the actual Bitmap inside the
6, and 32. The number of actual
depth of the Display and set how many colors you can see, execute: (Display newDepth: 8). (See comment in DisplayMedium)

Color is represented as the amount of light in red, green, and blue. White is (1.0, 1.0, 1.0) and black is (0, 0, 0). Pure red is (1.0, 0, 0). These colors are "additive". Think of Color's instance variables as:
r amount of red, a Float between 0.0 and 1.0.

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!



Constructing the Color Names

The screenshot shows a Smalltalk IDE window titled "Implementors of named:put:". The main pane displays the implementation of the `named:put:` message for the `Color` class. The implementation includes a comment, a block of code that capitalizes the name, creates a symbol, and defines an accessor message. A sidebar on the left shows a list of objects, including `Graphic`, `ImageF`, `Kernel`, `KernelT`, `Montice`, `Morphi`, `Morphi`, `Morphi`, `Movies`, `dwh 7/`, and `initializ`.

```
Color class named:put: {class initialization}

apb 2/12/2009 16:12 · class initialization · 1 implementor · only in change set Unnamed ·

browse senders implementors versions inheritance hierarchy inst vars class vars source R

named: newName put: aColor
    "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 cap sym accessor csym |
    (aColor isKindOf: self) ifFalse: [+ self error: 'not a Color'].
    str := newName asString.
    sym := str asSymbol.
    cap := str capitalized.
    csym := cap asSymbol.
    (self class canUnderstand: sym) ifFalse: [
        "define access message"
        accessor := str, (String with: Character cr with: Character tab), '+', cap.
        self class compile: accessor
            classified: 'named colors'].
    (self classPool includesKey: csym) ifFalse: [
        self addClassVarName: cap].
    (ColorNames includes: sym) ifFalse: [
        ColorNames add: sym].
    + self classPool at: csym put: aColor
```

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`, ...)?

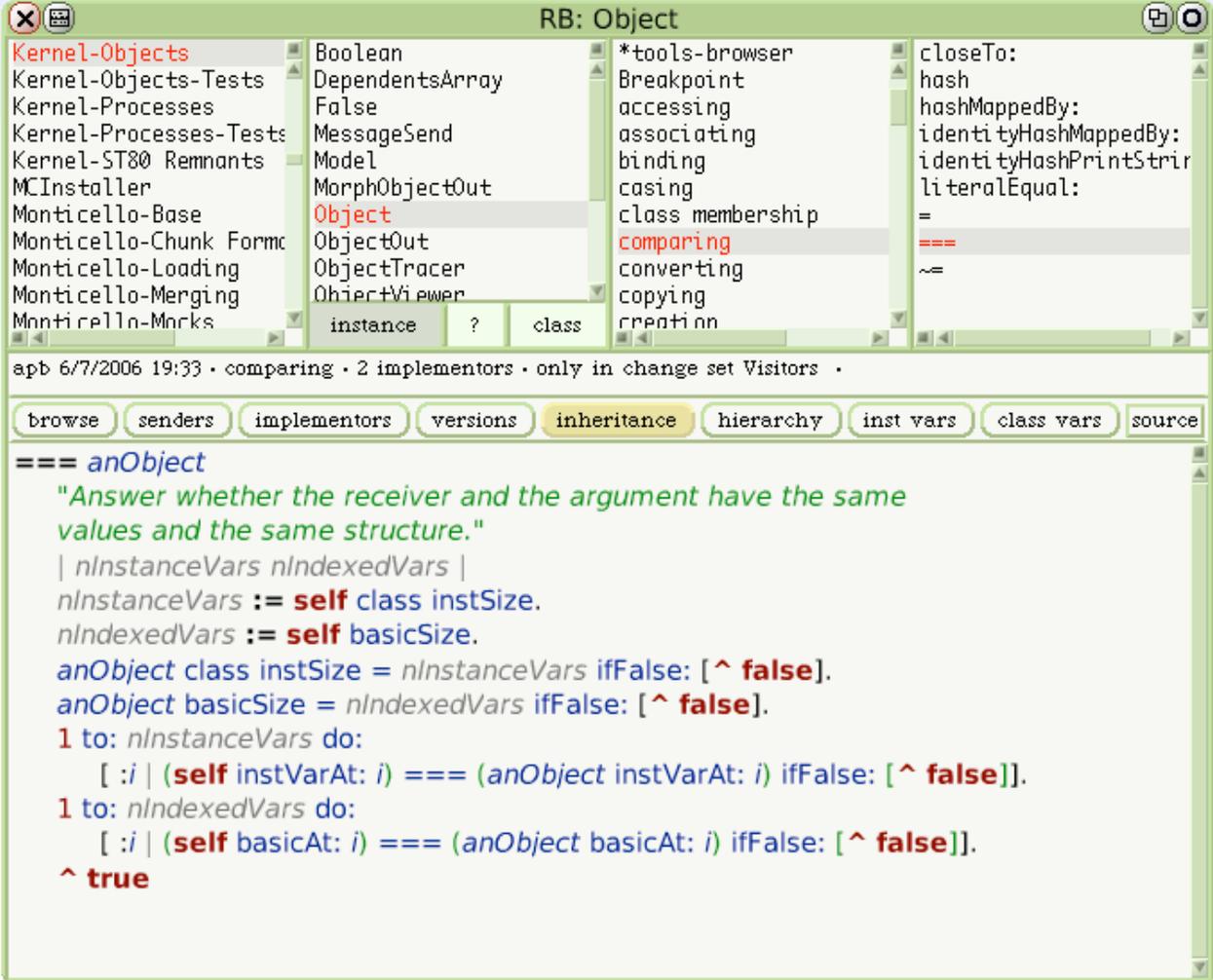
Summary of Solution

- (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



The screenshot shows the Ruby Object browser interface for the `Object` class. The left pane lists various classes, with `Object` selected. The right pane shows the `comparing` method. Below the browser, the source code for the `comparing` method is displayed, showing its implementation for comparing two objects based on their instance and indexed variables.

```
apb 6/7/2006 19:33 · comparing · 2 implementors · only in change set Visitors ·  
browse senders implementors versions inheritance hierarchy inst vars class vars source  
=== 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 iffFalse: [ ^ false].  
anObject basicSize = nIndexedVars iffFalse: [ ^ false].  
1 to: nInstanceVars do:  
  [:i | (self instVarAt: i) === (anObject instVarAt: i) iffFalse: [ ^ false]].  
1 to: nIndexedVars do:  
  [:i | (self basicAt: i) === (anObject basicAt: i) iffFalse: [ ^ 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

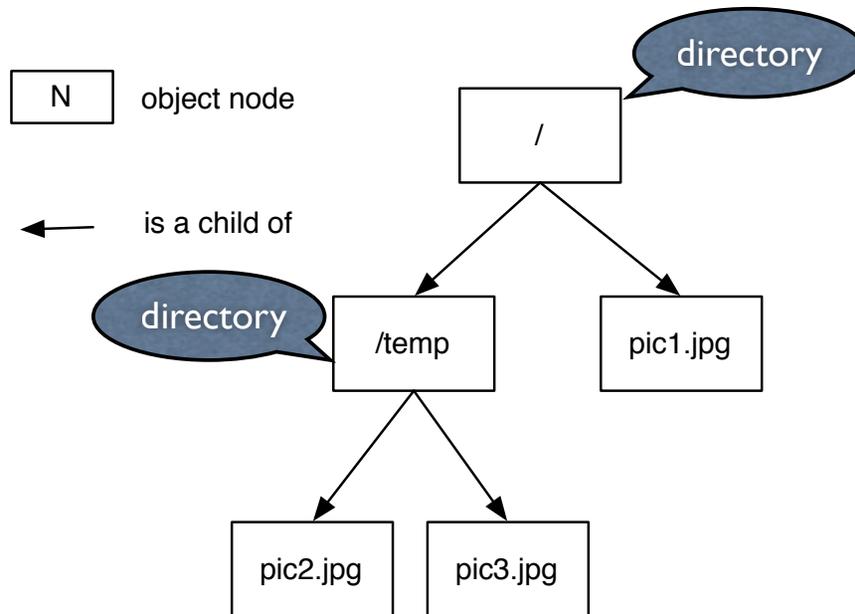
The OmniBrowser

- 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

The screenshot shows the RB: OBMetagraph IDE. The left pane lists a hierarchy of classes including `OmniBrowser-Morphic`, `OmniBrowser-Nodes`, `OmniBrowser-Notificati`, `OmniBrowser-Panels`, `OmniBrowser-Tests`, `OmniBrowser-Utilities`, `OB-Standard-Actors`, `OB-Standard-Browsers`, `OB-Standard-Defini tion`, `OB-Standard-Nodes`, and `OB-Standard-Tests`. The middle pane shows a list of classes: `OBDefinition`, `OBMetaGraphBuilder`, `OBMetagraph`, `OBModalFilter`, and `OBMorphicPanellLayout`. The right pane shows a list of classes: `fileBrowser`. Below the panes is a status bar with the text: `no timeStamp · *OB-Andrew's Experiment · 1 implementor · only in change set Visitors ·`. A navigation bar contains buttons for `browse`, `senders`, `implementors`, `versions`, `inheritance`, `hierarchy`, `inst vars`, `class vars`, and `source`. The main editor displays the following code for `fileBrowser`:

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

To the right of the code is a metagraph diagram with two nodes: `Directory` and `File`. A self-loop arrow on the `Directory` node is labeled `#directories`. A downward arrow from `Directory` to `File` is labeled `#files`.