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

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, 0.0). Pure red is (1.0, 0.0, 0.0). These colors are "additive". Think of Color's instance variables as:

- 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

```
Named newColor 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."

<table>
<thead>
<tr>
<th>str</th>
<th>cap</th>
<th>sym</th>
<th>accessor</th>
<th>cSym</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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:

```pascal
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:

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

• Or, they can be deployed on every Bean

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

• re-defining `new` is itself a form of metaprogramming
Another Example

• We know that we can write this:
  \[(1 \text{ to} 10) \text{ select: } [ \, x \mid x \text{ even} ]\]

• How about this?
  \[(1 \text{ to} 10) \text{ select even}\]

• 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

```ruby
Try a new Equality Operation

== 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 \cdot $A \to: <$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

Figure 2: A graph describing a filesystem sat and its corresponding metagraph

The abstract relationships between different types of nodes within the browser are defined using another type of graph, a metagraph. In a metagraph, each type of node of an object graph is represented by a metanode. As a whole, the metagraph describes the structure of the object graph navigated by the browser.

Edges between metanodes define which kinds of nodes may be connected to which other kinds of nodes. "Kind of node" is a notion that is formalized by object node classes; typically, there is a metanode for each node class, as is the case in our file browser.

The metagraph for the file browser is shown in Figure 2. There are two metanodes for the two types of nodes in the object graph: A directory may contain other directories and files; the two edges in the metagraph represent these two relationships.

OmniBrowser opts to store metagraphs in a central place. New metagraphs should be returned from class methods of the class `OBMetagraph`. Thus we construct the file browser metagraph in the method `#fileBrowser`.
Metagraph as data

fileBrowser

<table>
<thead>
<tr>
<th>dir file</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir := OBMetaNode named: 'Directory'.</td>
</tr>
<tr>
<td>file := OBMetaNode named: 'File'.</td>
</tr>
<tr>
<td>dir</td>
</tr>
<tr>
<td>childAt: #directories put: dir;</td>
</tr>
<tr>
<td>childAt: #files put: file.</td>
</tr>
<tr>
<td>^ dir</td>
</tr>
</tbody>
</table>

Diagram:

```
Directory

#directories

File
```

---

Portland State University