CS410/510 Advanced Programming

Meta-Matters in Squeak

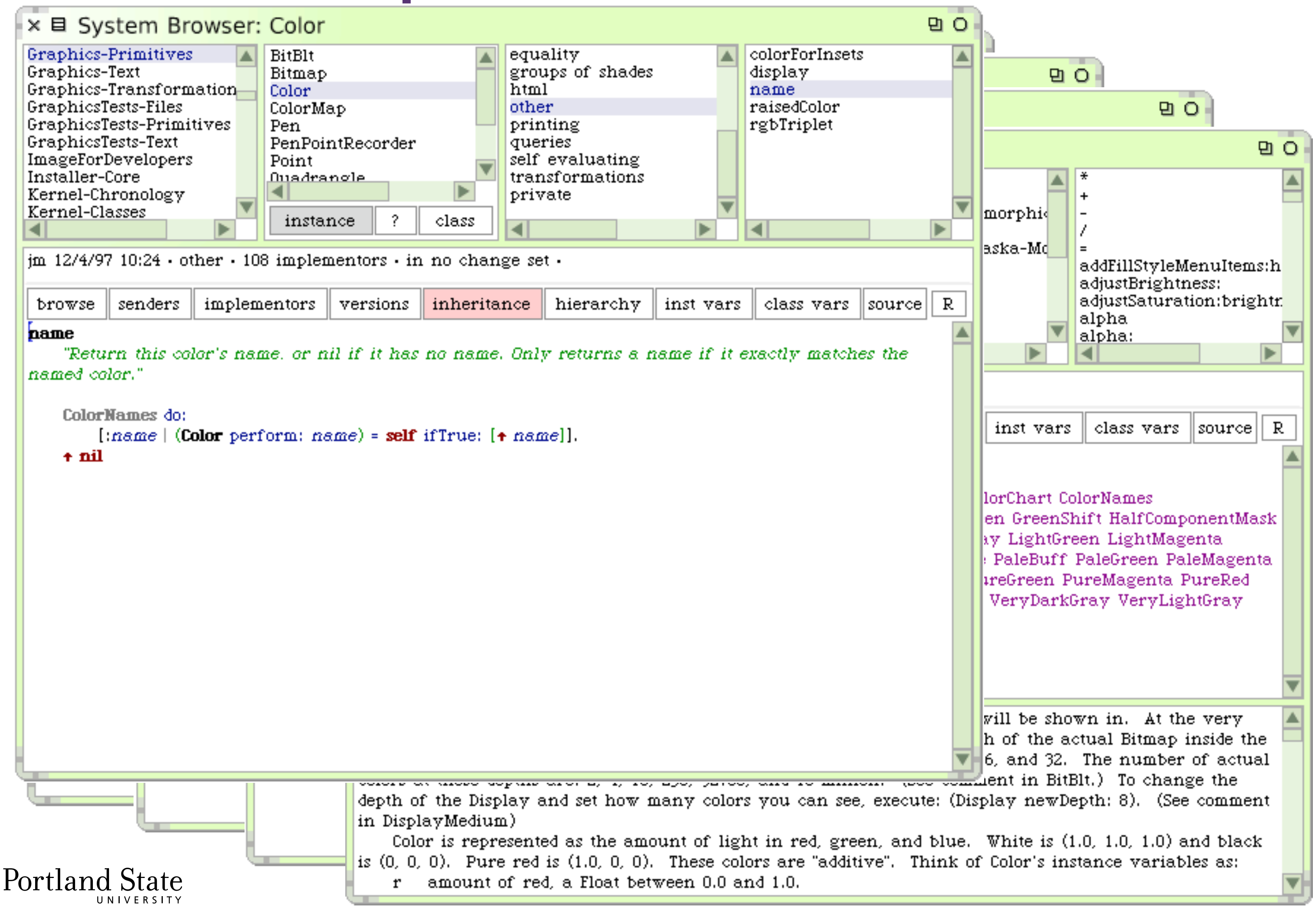
Andew P. Black

Portland State
UNIVERSITY

What's Meta?

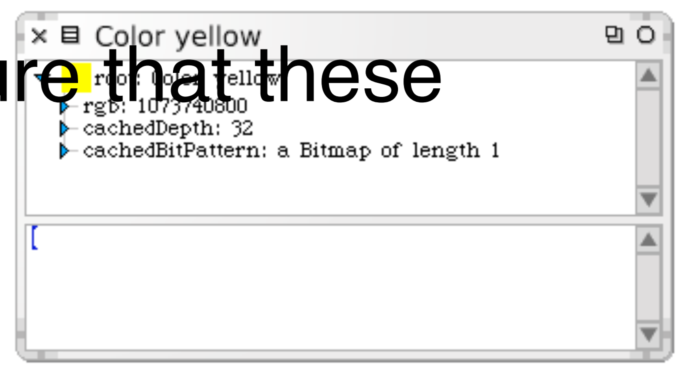
- Metaprogramming is the act of writing a program that writes or manipulates another program… or itself
- Why not? After all programs are just data!

Example: named colors

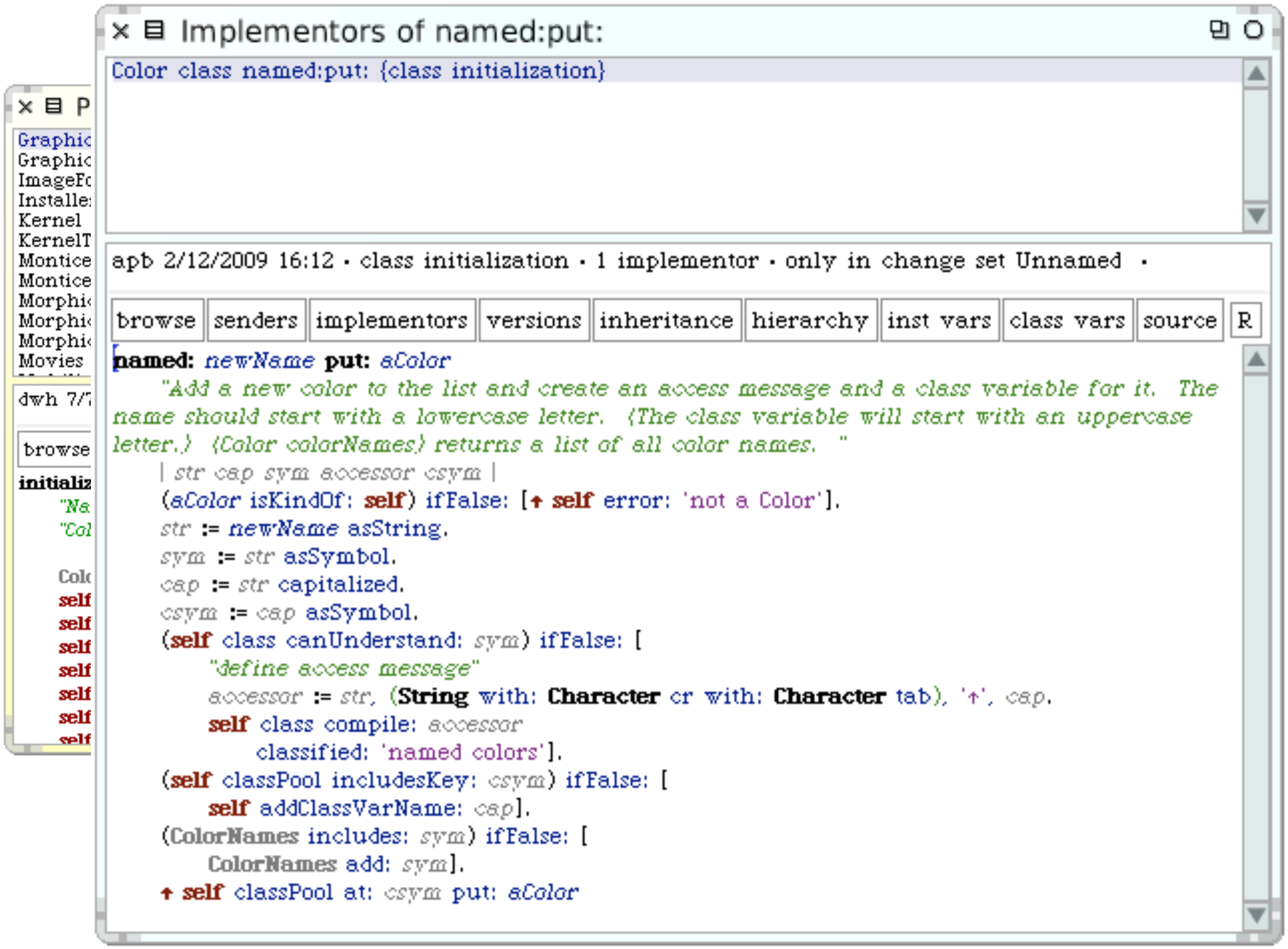


Named Colors (cont)

- 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



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*.

Portland State UNIVERSITY

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
```

Portland State University

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`, …)?

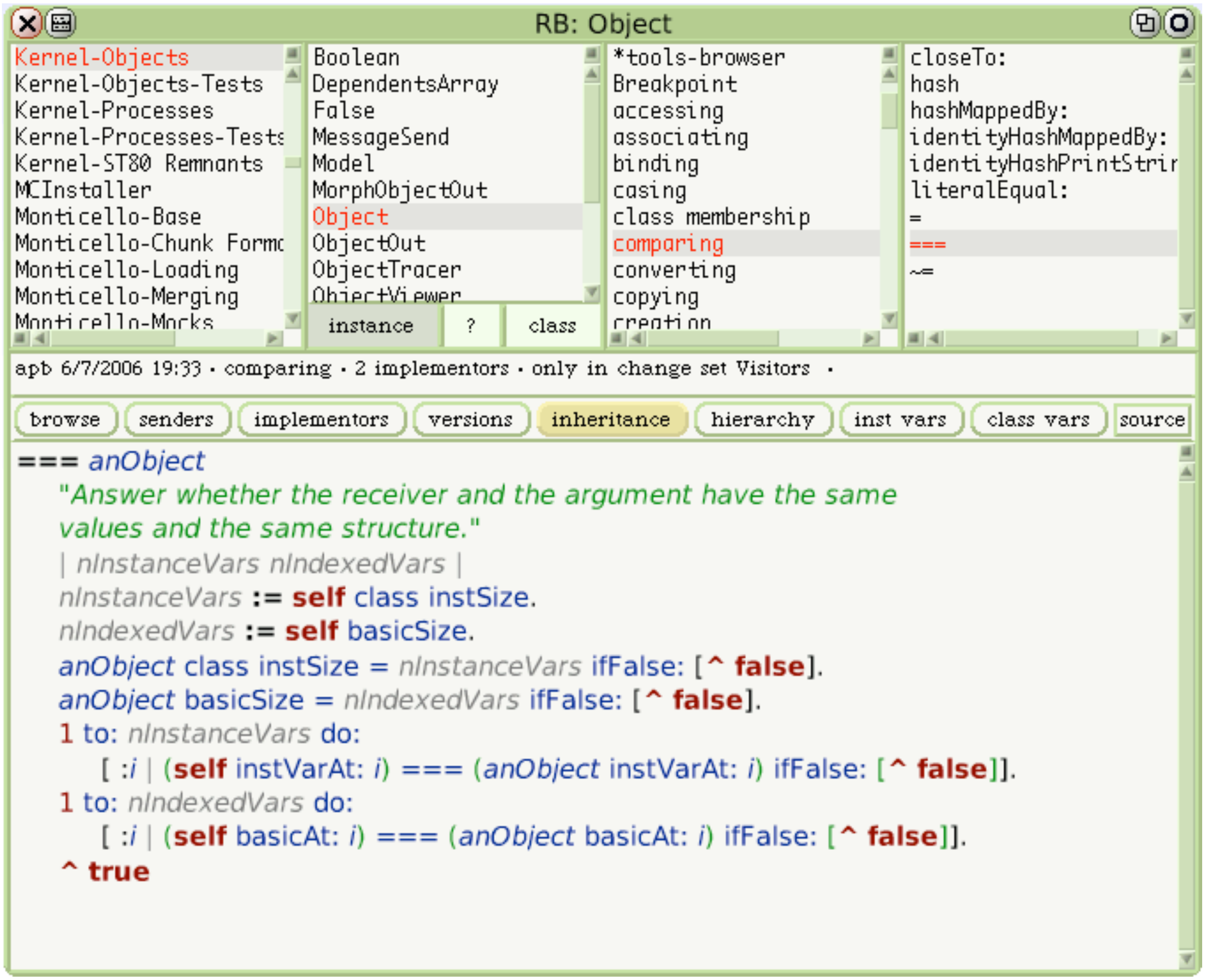
Summary of Solution

- (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



How does === work out?

Portland State University

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

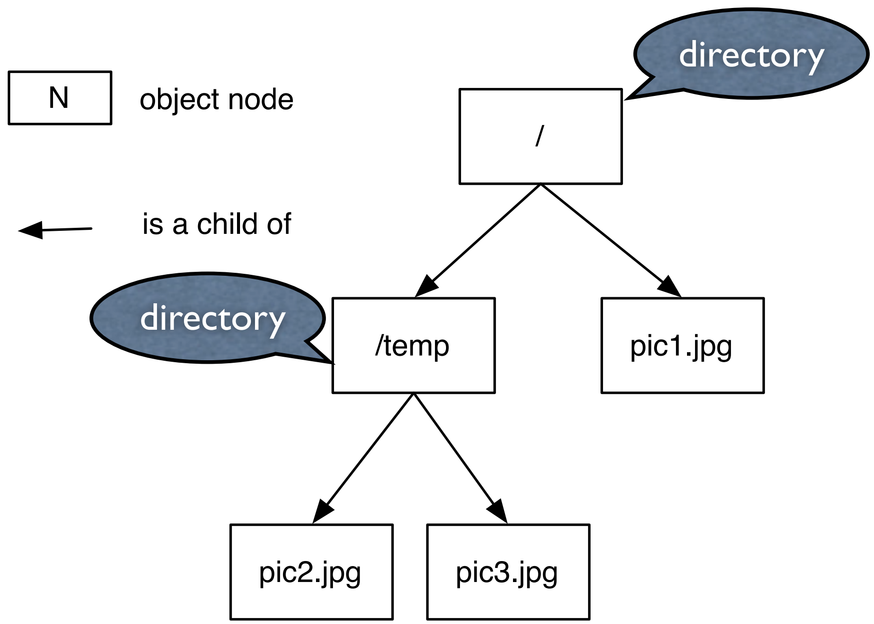
The OmniBrowser

- 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