2. Smalltalk Basics

Birds-eye view

Less is More — simple syntax and semantics uniformly applied can lead to an expressive and flexible system, not an impoverished one.

Roadmap

- Everything is an Object
- Syntax on a Postcard
- Three Kinds of Messages
- Methods, variables and blocks
- Test-Driven Development
- Managing Projects

Selected material courtesy Stéphane Ducasse

Roadmap

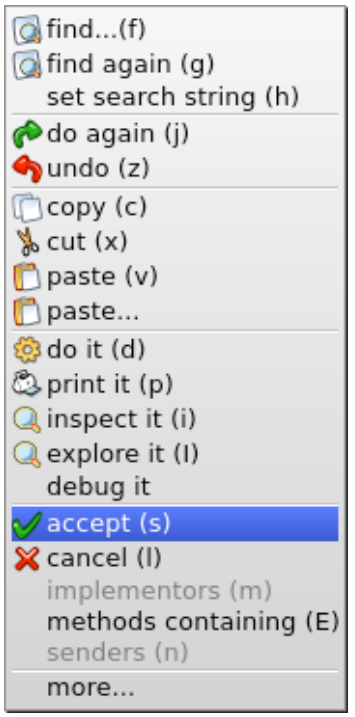
- **Everything is an Object**
- Syntax on a Postcard
- Three Kinds of Messages
- Methods, variables and blocks
- Test-Driven Development
- Managing Projects

Objects in Smalltalk

- *Everything* is an object
  - Things only happen by message passing
  - Variables are dynamically bound
- State is private to objects
  - “protected” for subclasses
- Methods are public
  - “private” methods by convention only
- (Nearly) every object is a reference
  - Unused objects are garbage-collected
- Single inheritance

Accept, DoIt, PrintIt and InspectIt

- *Accept*
  - Compile a method or a class definition
- *DoIt*
  - Evaluate an expression
- *PrintIt*
  - Evaluate an expression and print the result (`#printOn:`)
- *InspectIt*
  - Evaluate an expression and inspect the result (`#inspect`)

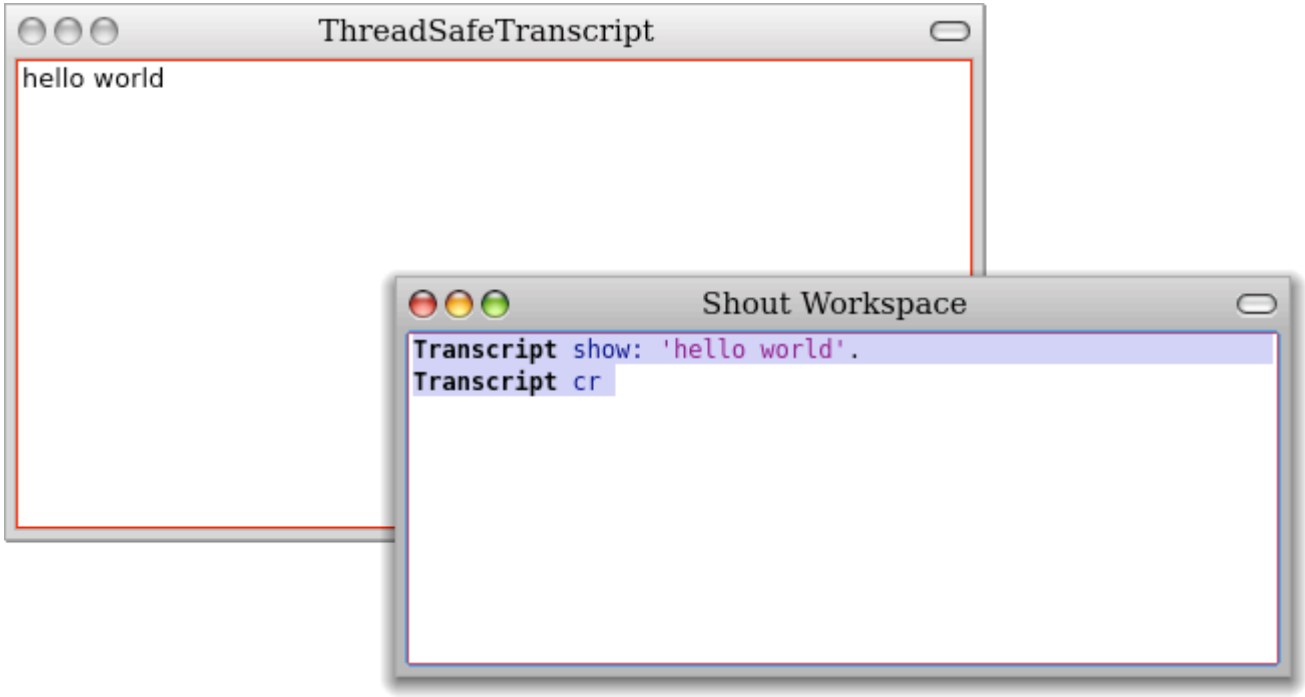


Hello World

- At anytime, *in any tool*, we can dynamically ask the system to evaluate an expression.
  - To evaluate an expression, select it and with the middle mouse button apply doIt.

```
Transcript show: 'hello world'
```

“Hello World”



- Transcript is a kind of “standard output”
  - a TextCollector instance associated with the launcher.

Everything is an Object

- ***Smalltalk is a consistent, uniform world, written in itself***
  - You can learn how it is implemented, you can extend it or even modify it.
  - All the code is available and readable.
    - *The workspace is an object.*
    - *The window is an instance of SystemWindow.*
    - *The text editor is an instance of ParagraphEditor.*
    - *The scrollbars are objects too.*
    - *`'hello word'` is an instance of String.*
    - *`#show:` is a Symbol*
    - *The mouse is an object.*
    - *The parser is an instance of Parser.*
    - *The compiler is an instance of Compiler.*
    - *The process scheduler is also an object.*

Roadmap

- Everything is an Object
- **Syntax on a Postcard**
- Three Kinds of Messages
- Methods, variables and blocks
- Test-Driven Development
- Managing Projects

Smalltalk Syntax on a Postcard

```
exampleWithNumber: x
"A method that illustrates every part of Smalltalk method syntax
except primitives. It has unary, binary, and key word messages,
declares arguments and temporaries (but not block temporaries),
accesses a global variable (but not and instance variable),
uses literals (array, character, symbol, string, integer, float),
uses the pseudo variable true false, nil, self, and super,
and has sequence, assignment, return and cascade. It has both zero
argument and one argument blocks. It doesn't do anything useful, though"
    |y|
    true & false not & (nil isNil) ifFalse: [self halt].
    y := self size + super size.
    #($a #a 'a' 1 1.0)
        do: [:each | Transcript
            show: (each class name);
            show: (each printString);
            show: ' '].
    ^ x < y
```

Language Constructs

^	return
"…"	comment
#	symbol or array
'…'	string
[ ]	block or byte array (VisualWorks)
.	statement separator
;	message cascade
|…|	local or block variable
:=	assignment (also _ or ←)
$_	character
:	end of selector name
_e_ _r_	number exponent or radix
!	file element separator (used in change sets)
<primitive: ...>	for VM primitive calls

Examples

comment:	`"a comment"`
character:	`$c $h $a $r $a $c $t $e $r $s $# $@`
string:	`'a nice string'`
symbol:	`#mac #+`
array:	`#(1 2 3 (1 3) $a 4)`
dynamic array (Pharo):	`{ 1 + 2 . 3 / 4 }`
Integer:	`1, 2r101`
real:	`1.5, 6.03e-34,4, 2.4e7`
fraction:	`1/33`
boolean:	`true, false`
pseudo variables	`self, super`
point:	`10@120`

Note that `@` is not an element of the syntax, but just a message sent to a number. This is the same for `/`, `bitShift`, `ifTrue:`, `do:` ...

Roadmap

- Everything is an Object
- Syntax on a Postcard
- **Three Kinds of Messages**
- Methods, variables and blocks
- Test-Driven Development
- Managing Projects

Messages instead of keywords

- In most languages, basic operators and control constructs are defined as language constructs and keywords
- In Smalltalk, there are only *messages* sent to objects
  - `bitShift:` (>>) is just *a message sent to a number*

```
10 bitShift: 2
```

  - `ifTrue:` (if-then-else) is just *a message sent to a boolean*

```
(x>1) ifTrue: [ Transcript show: 'bigger' ]
```

  - `do:`, `to:do:` (loops) are just *messages to collections or numbers*

```
#(a b c d) do: [:each | Transcript show: each ; cr]
1 to: 10 do: [:i | Transcript show: i printString; cr]
```

- Minimal parsing
- Language is *extensible*

Smalltalk Syntax

Every expression is a message send

- *Unary messages*

```
Transcript cr
5 factorial
```

- *Binary messages*

```
3 + 4
```

- *Keyword messages*

```
Transcript show: 'hello world'
2 raisedTo: 32
3 raisedTo: 10 modulo: 5
```

Precedence

(…) > Unary > Binary > Keyword

1. Evaluate *left-to-right*
2. Unary messages have *highest precedence*
3. Next are binary messages
4. Keyword messages have *lowest precedence*

```
2 raisedTo: 1 + 3 factorial
```

```
128
```

5. Use *parentheses* to change precedence

```
1 + 2 * 3
1 + (2 * 3)
```

```
9       (!)
7
```

Binary Messages

- Syntax:
  - aReceiver *aSelector* anArgument
  - Where *aSelector* is made up of 1 or 2 characters from:

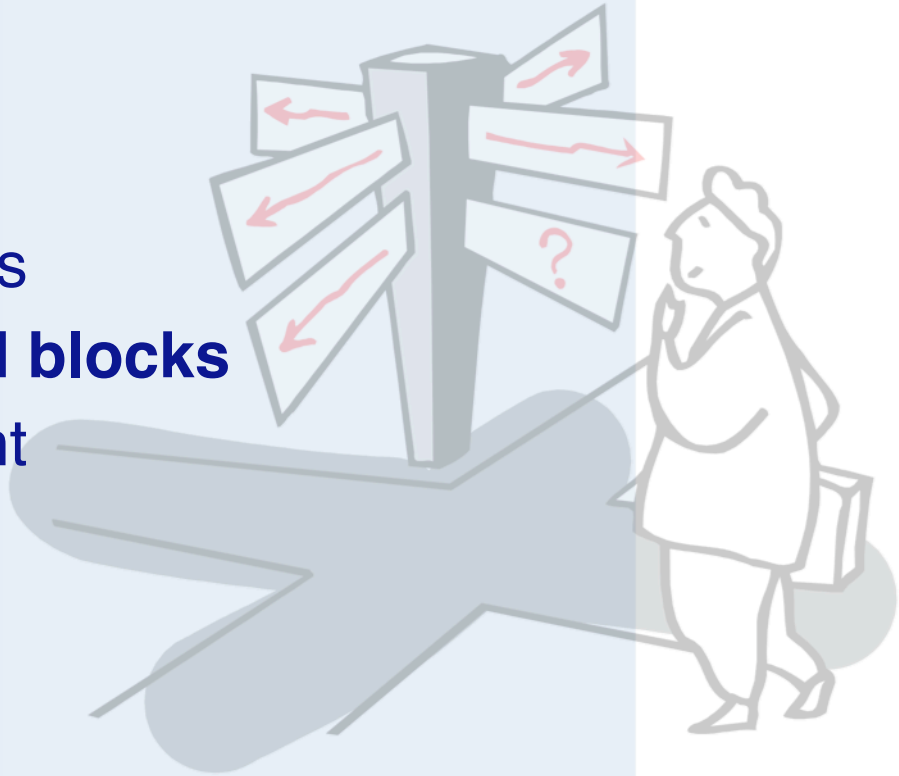
    `+ - / \ * ~ < > = @ % | & ! ? ,`
  - Except: second character may not be `$`

- Examples:

```
2 * 3 - 5
5 >= 7
6 = 7
'hello', ' ', 'world'
(3@4) + (1@2)
2<<5
64>>5
```

Roadmap

- Everything is an Object
- Syntax on a Postcard
- Three Kinds of Messages
- **Methods, variables and blocks**
- Test-Driven Development
- Managing Projects



More syntax

- Comments are enclosed in *double quotes*

```
"This is a comment."
```

- Use *periods* to separate expressions

```
Transcript cr.
Transcript show: 'hello world'.
Transcript cr      "NB: don't need one here"
```

- Use *semi-colons* to send a cascade of messages to the same object

```
Transcript cr; show: 'hello world'; cr
```

Variables

- Declare local variables with | … |

```
| x y |
```

- Use := to assign a value to a variable

```
x := 1
```

- Old fashioned assignment operator: ← (must type "_")

Method Return

- Use a *caret* to return a value from a method or a block

```
max: aNumber
    ^ self < aNumber
        ifTrue: [aNumber]
        ifFalse: [self]
```

```
1 max: 2
```

2

- Methods *always* return a value
  - By default, methods return `self`

Block closures

- Use *square brackets* to delay evaluation of expressions

```
^ 1 < 2 ifTrue: ['smaller'] ifFalse: ['bigger']
```

```
'smaller'
```

Variables

- *Local variables* within methods (or blocks) are delimited by |*var*|
- *Block parameters* are delimited by :*var*|

```
OrderedCollection>>collect: aBlock
    "Evaluate aBlock with each of my elements as the argument."
    | newCollection |
    newCollection := self species new: self size.
    firstIndex to: lastIndex do:
        [ :index |
        newCollection addLast: (aBlock value: (array at: index))].
    ^ newCollection
```

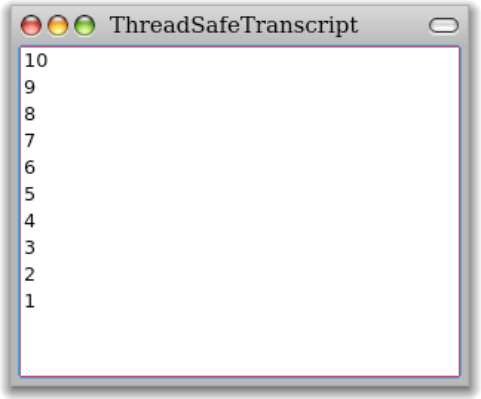
```
[:n | |x y| x := n+1. y := n-1. x * y] value: 10
```

99

Control Structures

> *Every control structure is realized by message sends*

```
|n|
n := 10.
[n>0] whileTrue:
   [  Transcript show: n; cr.
      n := n-1   ]
```




```
1 to: 10 do: [:n| Transcript show: n; cr ]
```

```
(1 to: 10) do: [:n| Transcript show: n; cr ]
```

Creating objects

- *Class methods*

```
OrderedCollection new
Array with: 1 with: 2
```

- *Factory methods*

```
1@2     "a Point"
1/2     "a Fraction"
```

Creating classes

- Send a message to a class (!)

```
Number subclass: #Complex
   instanceVariableNames: 'real imaginary'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'ComplexNumbers'
```

Some Conventions

- Method selector is a *symbol*, e.g., `#add:`
- Method scope conventions using >>
  - *Instance Method* defined in the class `Node`

```
Node>>accept: aPacket
```

  - *Class Method* defined in the class `Node class`
    (i.e., in the class of the the class Node)

```
Node class>>withName: aSymbol
```

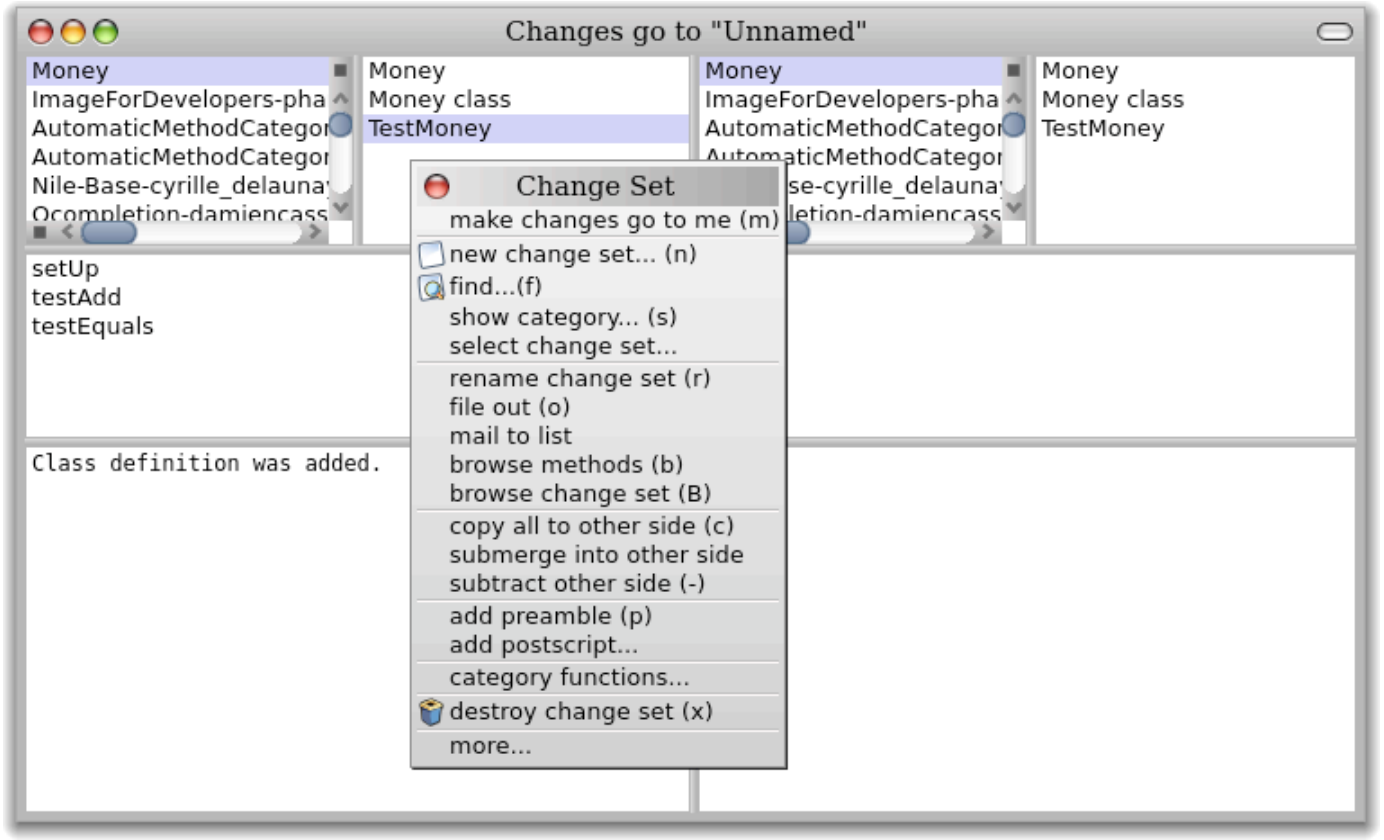
- `aSomething` is an instance of the class `Something`

Roadmap

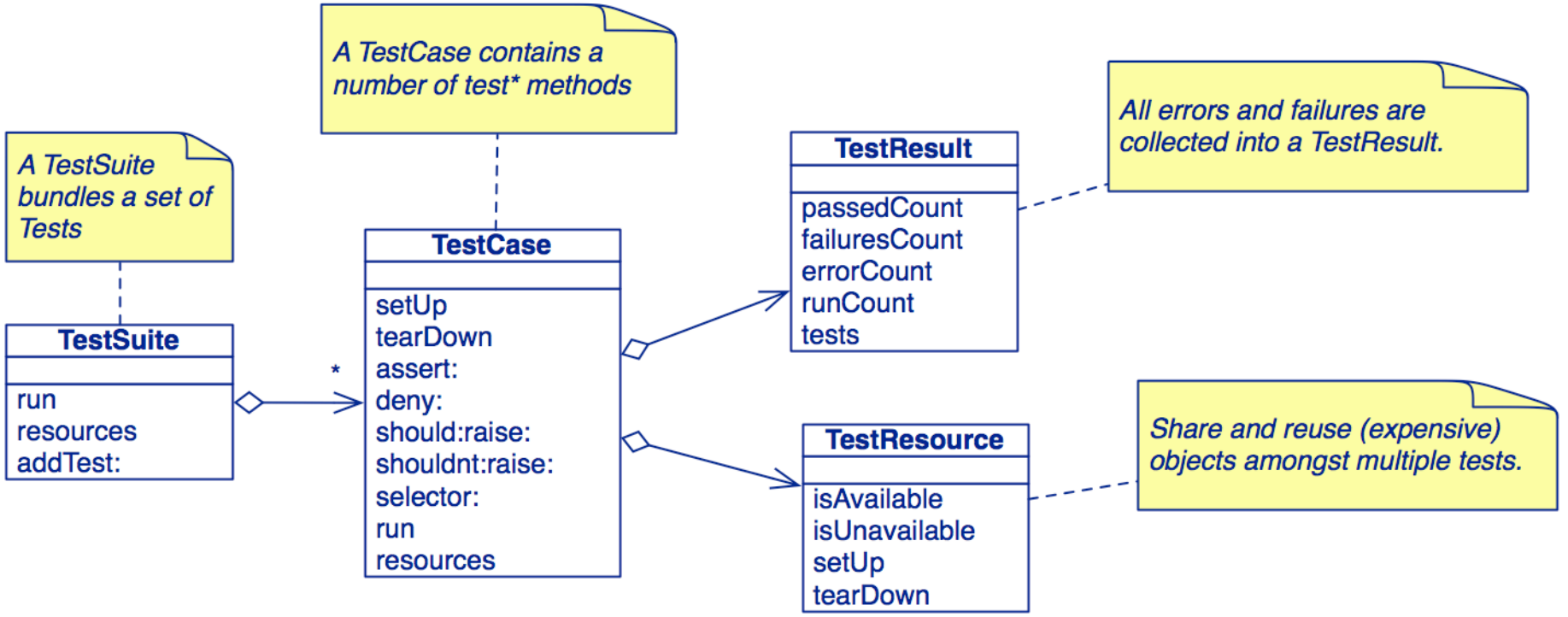
- Everything is an Object
- Syntax on a Postcard
- Three Kinds of Messages
- Methods, variables and blocks
- **Test-Driven Development**
- Managing Projects

Change sets

> Make sure your changes are logged to a new change set



SUnit



Money

- We will implement the `Money` example in Smalltalk
  - First, we develop a test case for a single currency

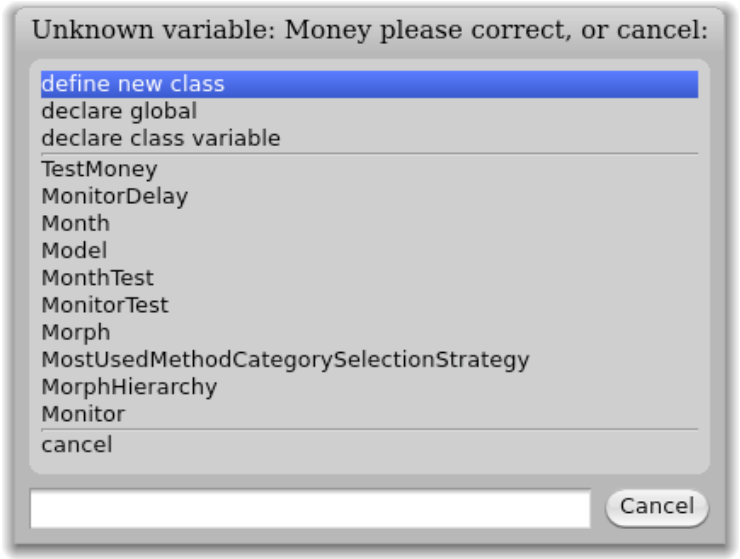
```
TestCase subclass: #MoneyTest
  instanceVariableNames: 'chf2 chf8 chf10'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Money'
```

NB: *This is just a message sent to the `TestCase` class object (!)*

SetUp

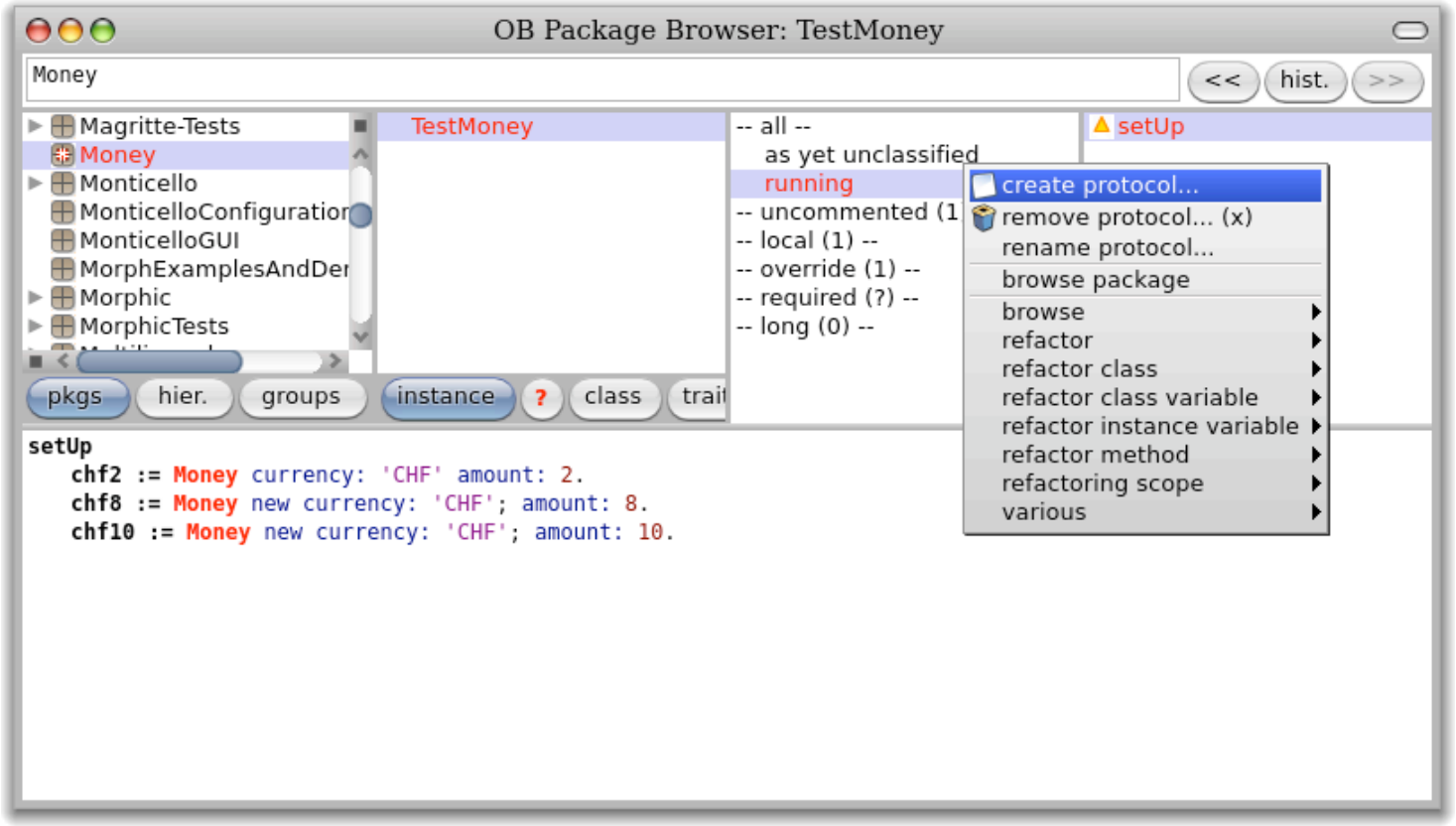
- We will need setters for the private Money state

```
MoneyTest>>setUp
   chf2 := Money new currency: 'CHF'; amount: 2.
   chf8 := Money new currency: 'CHF'; amount: 8.
   chf10 := Money new currency: 'CHF'; amount: 10.
```



Protocols

Classify methods into *protocols* that reveal their intent



MoneyTest>>testEquals

- Some obvious tests

```
MoneyTest>>testEquals
   self assert: chf2 = chf2.
   self assert: chf2 = (Money new currency: 'CHF'; amount: 2).
   self assert: chf2 != chf8.
```

Money

- We define Money as a subclass of Object, with getters and setters

```
Object subclass: #Money
  instanceVariableNames: 'currency amount'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Money'
```

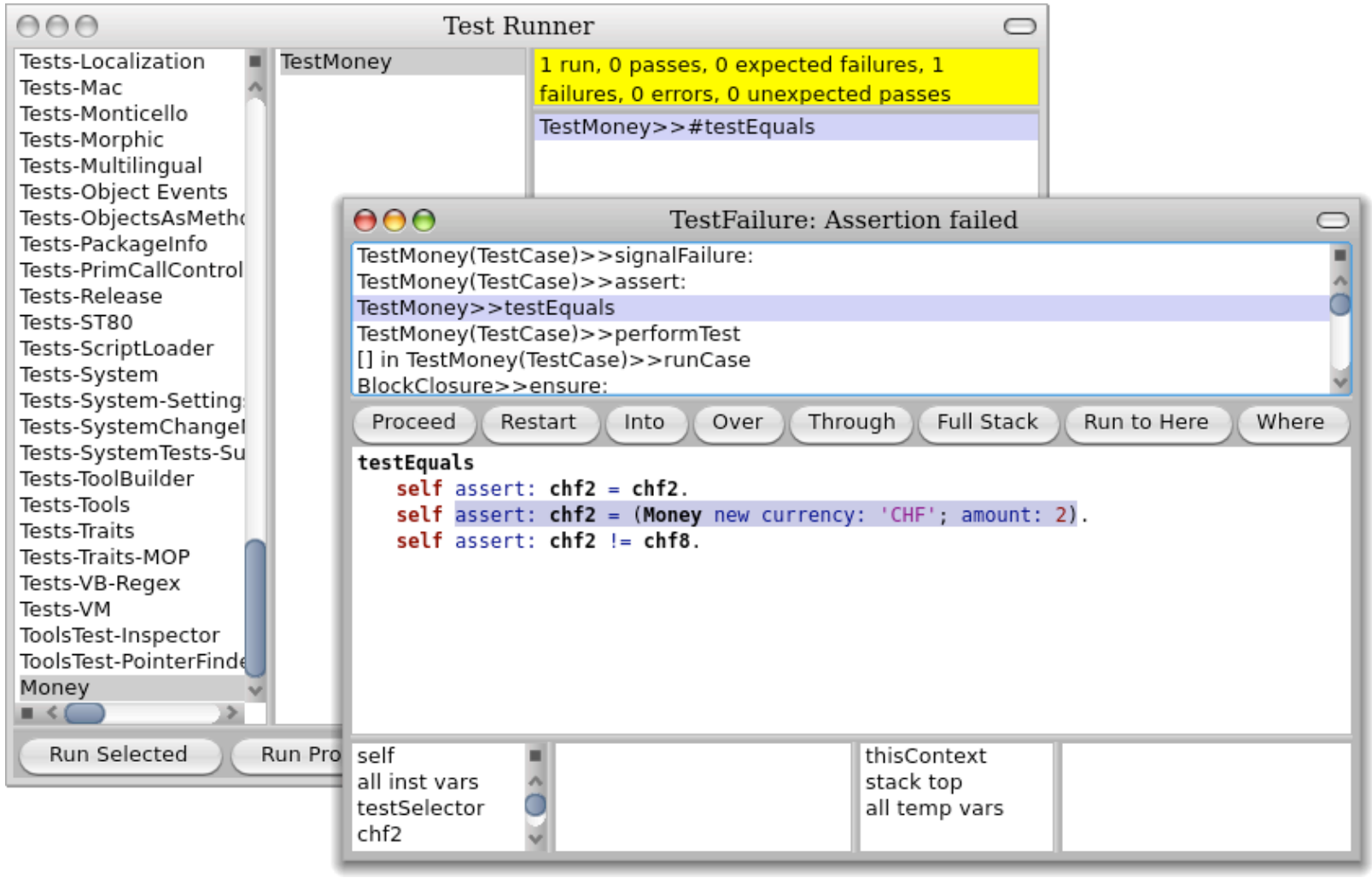
```
Money>>currency: aString
  currency := aString.
```

```
Money>>currency
  ^ currency
```

```
Money>>amount: aNumber
  amount:= aNumber.
```

```
Money>>amount
  ^ amount
```

Failing tests



Comparisons

```
Money>>= aMoney
   ^ self currency = aMoney currency
   and: [ self amount = aMoney amount ]
```

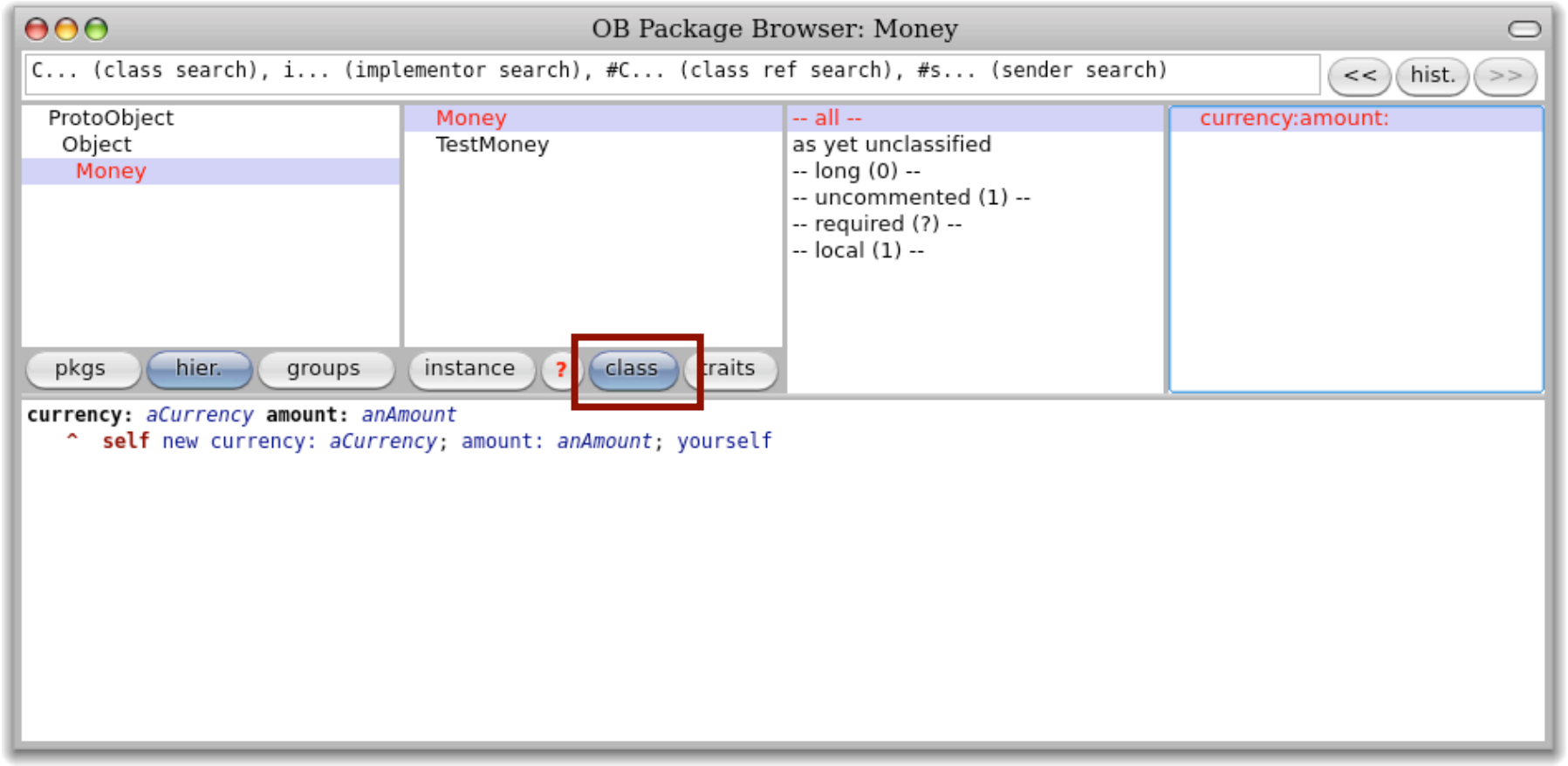
```
Money>>!= aMoney
   ^ (self = aMoney) not
```

Constructors

```
MoneyTest>>testEquals
   self assert: chf2 = chf2.
   self assert: chf2 =
      (Money currency: 'CHF' amount: 2).
   self assert: chf2 != chf8.
```

We need a constructor on the *class side* of `Money`

Class methods



NB: Which "self" is referred to in the method body?

Addition

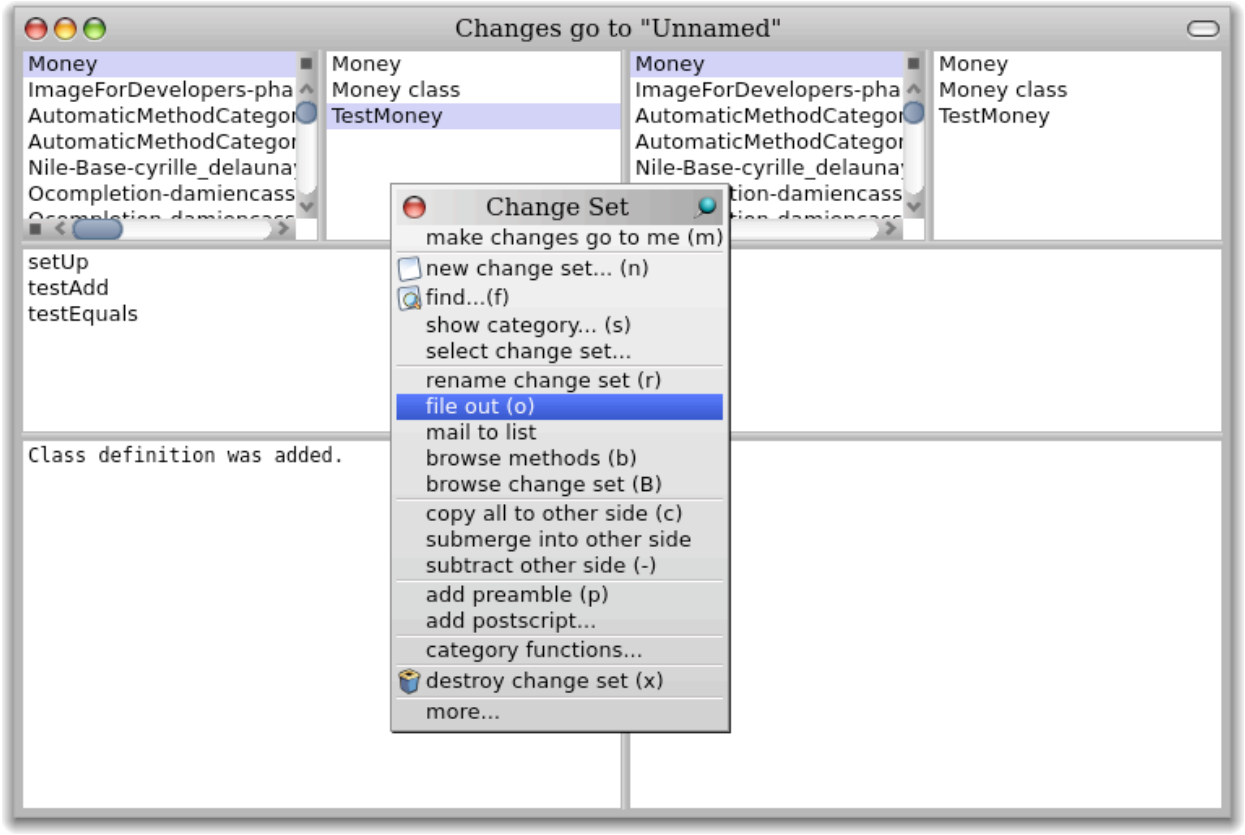
```
MoneyTest>>testAdd
   self assert: chf2 + chf8 = chf10
```

```
Money>>+ aMoney
   ^ Money currency: self currency
             amount: self amount + aMoney amount
```

And so on …

Filing out your changes

- You can “file out” all your changes so they can be loaded into another image



Change Sets

```
'From Pharo1.0beta of 16 May 2008 [Latest update: #10418] on 8 September 2009
at 1:58:51 pm'!
Object subclass: #Money
    instanceVariableNames: 'currency amount'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Money'!
TestCase subclass: #TestMoney
    instanceVariableNames: 'chf2 chf8 chf10'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Money'!

!Money methodsFor: 'as yet unclassified' stamp: 'on 7/2/2007 13:21'!
!!= aMoney
    ^ (self = aMoney) not! !

!Money methodsFor: 'as yet unclassified' stamp: 'on 7/2/2007 13:26'!
+ aMoney
    self assert: [ self currency = aMoney currency ].
    ^ Money currency: self currency amount: self amount + aMoney amount! !

!Money methodsFor: 'as yet unclassified' stamp: 'on 7/2/2007 13:20'!
= aMoney
    ^ self amount = aMoney amount and: [ self currency = aMoney currency ]! !
...
```

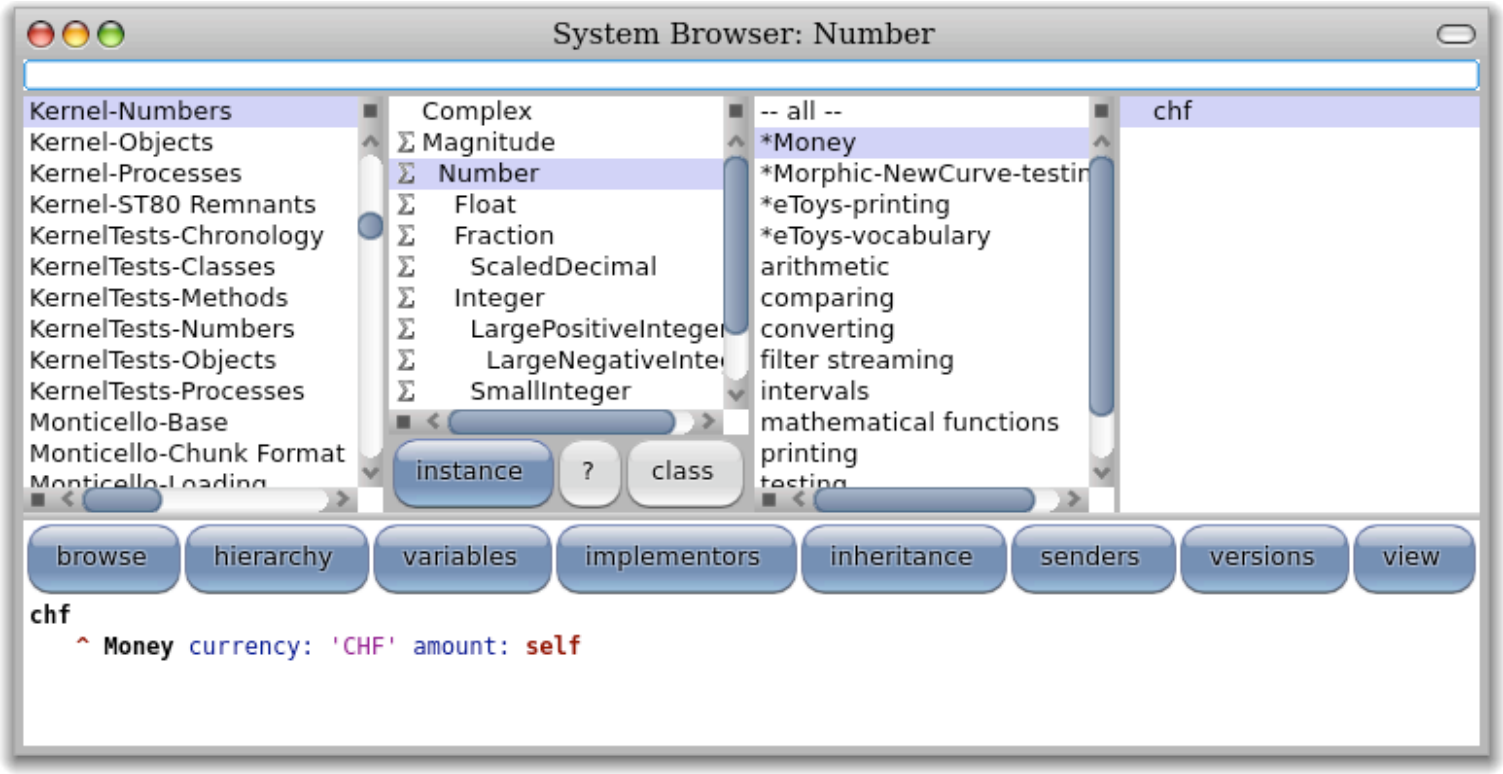
Extensions

- Creating new instances of `Money` is still cumbersome
  - We would like to be able to write instead code like this:

```
TestMoney>>setUp
  chf2 := 2 chf.
  chf8 := 8 chf.
  chf10 := 10 chf.
```

Extension protocols

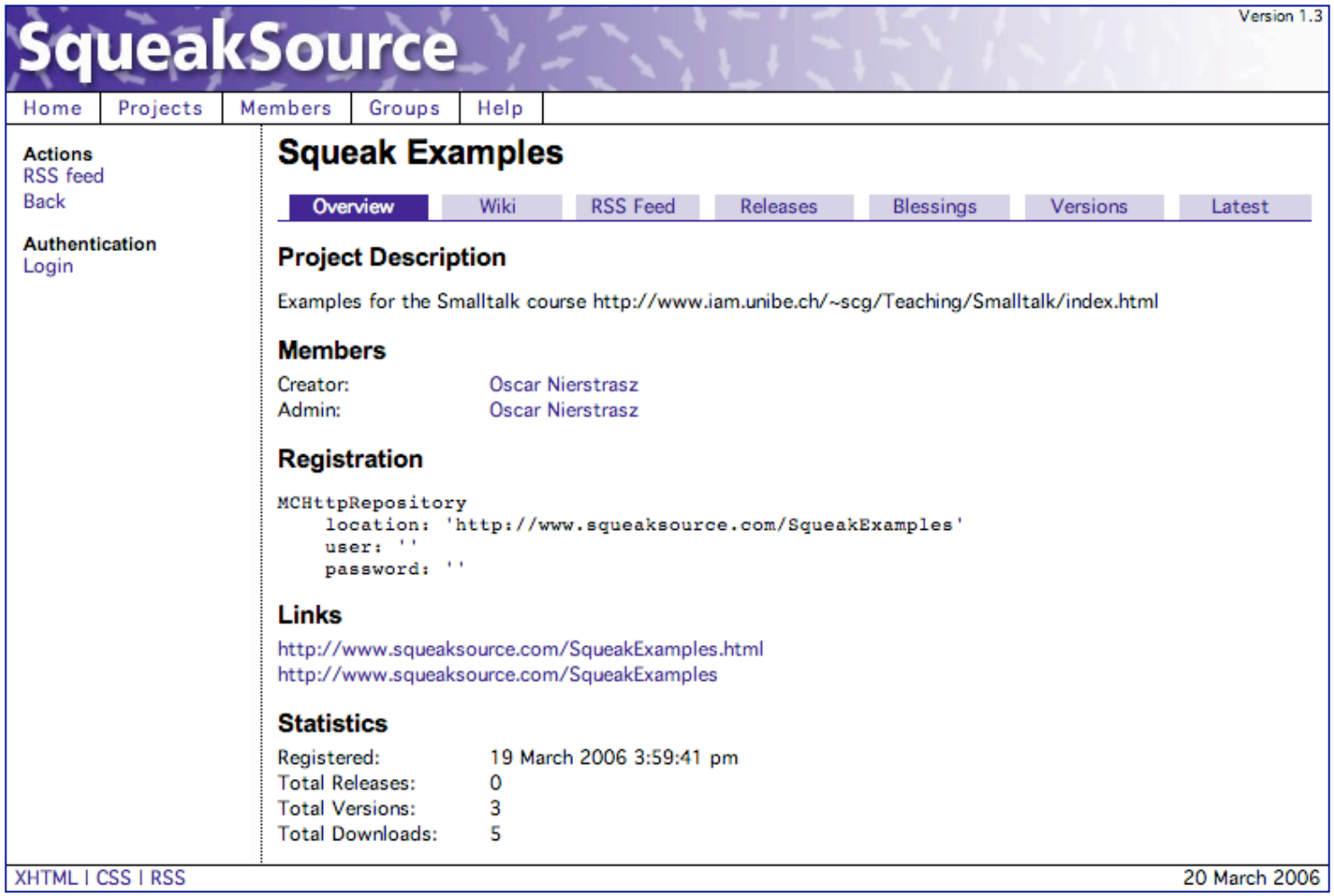
To extend an existing class for a category `MyProject`, define the protocol `*MyProject` (or `*myproject`) for that class.



Roadmap

- Everything is an Object
- Syntax on a Postcard
- Three Kinds of Messages
- Methods, variables and blocks
- Test-Driven Development
- **Managing Projects**

SqueakSource.com

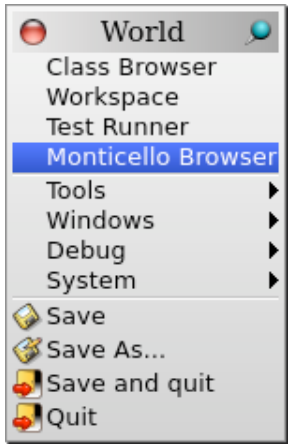


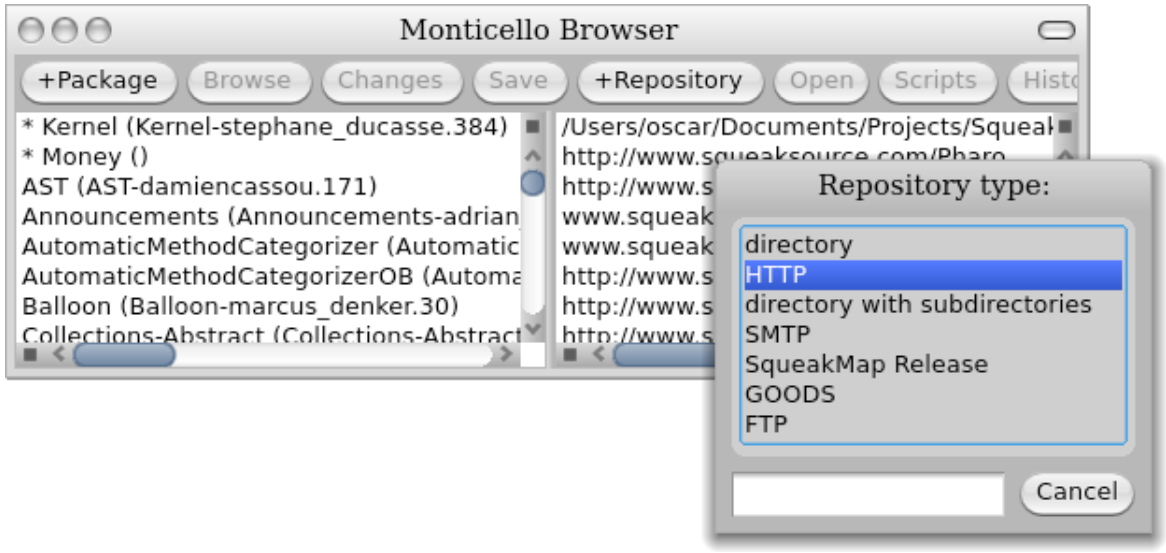
Categories, Projects and Packages

- A system category `MyProject` (and possibly `MyProject-*`) contains the classes of your application
- A Monticello package `MyProject` contains the categories `MyProject` and `MyProject-*`
  - *NB: If you have sub-categories, keep the top one empty!*
- A SqueakSource project `MyProject` stores everything in the Monticello package `MyProject`

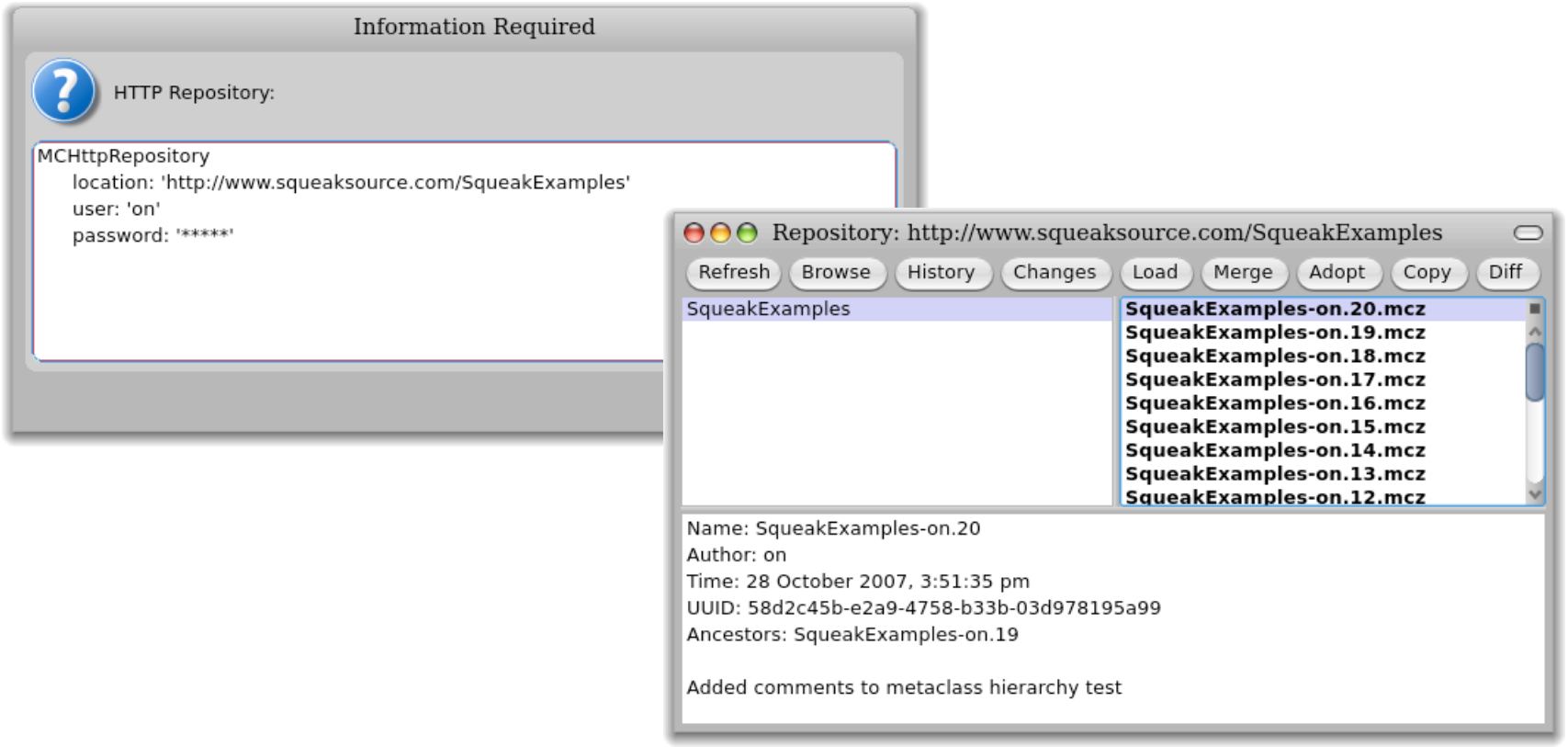
Loading a project

- To load a project from SqueakSource.com
  - Find the project in SqueakSource
  - Open the Monticello Browser
  - Add an HTTP Repository
    - *Accept the code fragment from the SqueakSource page*
  - Open the repository and load the latest version



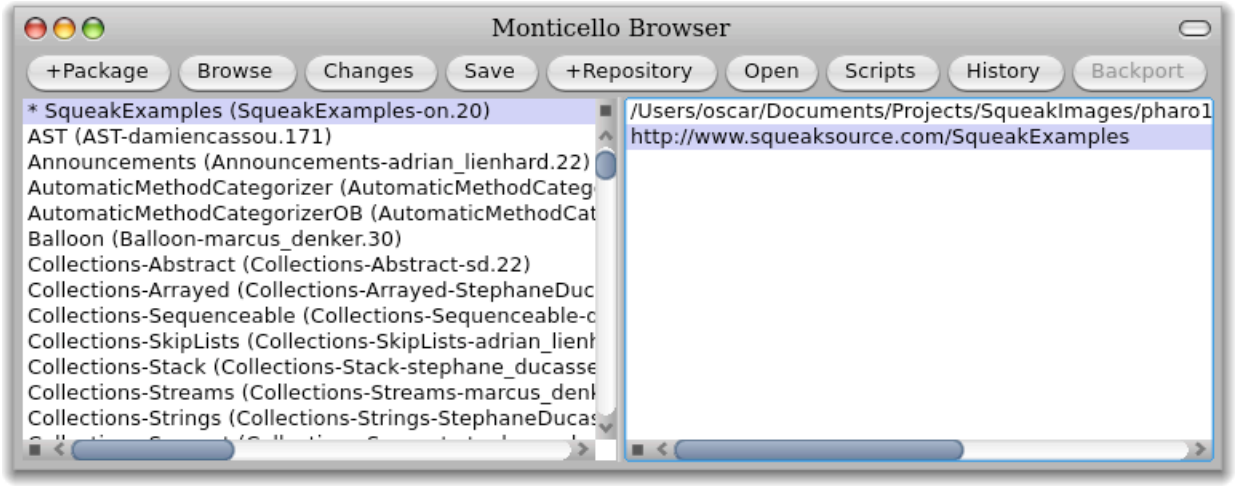


Loading a project



Creating a project

- To create a project on SqueakSource.com
  - Define your categories `MyProject` or `MyProject-*`
  - Create a SqueakSource project named `MyProject`
  - Define a Monticello package `MyProject`
  - Add an HTTP Repository for `MyProject`
    - *Accept the code fragment from the SqueakSource page*
  - Save the package



What you should know!

- *How can you indicate that a method is “private”?*
- *What is the difference between a comment and a string?*
- *Why does 1+2*3 = 9?*
- *What is a cascade?*
- *How is a block like a lambda expression?*
- *How do you create a new class?*
- *How do you inspect an object?*

Can you answer these questions?

- *Why does Smalltalk support single (and not multiple) inheritance?*
- *Is the cascade strictly necessary?*
- *Why do you need to declare local variables if there are no static types?*
- *How can you discover the class of a GUI object?*
- *How does SUnit differ from JUnit?*

License

http://creativecommons.org/licenses/by-sa/3.0/