Smalltalk

Smalltalk was designed by Alan Kay in 1972 when he was working at Xerox PARC (Palo Alto) as part of his Dynabook project.

It was influenced by some of Papert's work at MIT on Logo, an interactive highly graphical computer learning environment intended to teach programming to children.

Dynabook was a far sighted early version of a handheld computer.

The first was made of cardboard and filled with lead shot, to see what size and weight people would put up with.

Smalltalk

Smalltalk was designed an an interpretive language, in which all names were looked up at runtime from and environment of declared names.

In more moder implementations code is compiled giving a speedup of 450 times!

There are several Smalltalk implementations including a GNU version and one from the Disney Corporation (where Alan Kay now works).

This latter one is embedded in a system called Squeek and the version of Smalltalk is called Self.

It consists of a complete program development system that is responsive and small enough (approx 1-2 Mbytes) to fit in a Palm Top or Handheld computer.

Smalltalk

Smalltalk is a purely object-oriented language which cleanly supports the notions of classes, methods, messages and inheritance.

All smalltalk code consists of chains of messages sent to objects.

Its syntax is almost trivial.

```
BODY -> | name ... name | E . E ... E

E    -> name                        x  Transcript   :=
      | # E                         #green
      | number                      123
      | string                      'hello'
      | ( E )
      | ^ E                         return with value of E
      | [ E . E ... E ]             [Transcript show: 'hello']
      | [ :name ... :name | E . E ... E ]   [: x | x + 1]
      | E name
      | E name E ... name E
      | E name E ... name E ; ... ; name E ...
```

Smalltalk Fragments

```
|count|
count := 0.
[count < 3]
  whileTrue:
    [ (count odd) ifTrue: [Transcript show: 'ODD@]
                  ifFalse: [Transcript show: 'EVEN'].
       count := count+1.
       Transcript cr.
    ]
```

```
|canon|
canon := [ :singers :song |
           singers do:
             [:voice | Transcript
                         show: voice asString, ',', song;
                         cr]]

canon value: #(kermit jasper fred) value: 'croak'.
```

Smalltalk Class

For programming convenience Smalltalk classes a grouped into Categories and their methods are also grouped into categories.

```
Morphic-Window                -- Class categories
Graphics-3D
...
Interface-Pluggable
Sesame-Street
   CookieMonster              -- a class
     initialization           -- method categories
     access
     queries
       isAsleep               -- methods
       isAwake
       isFull
           self isEmpty       -- definition of isFull
             ifFalse: [^ self tummy size >= self hunger]
             ifTrue: [^false]
     actions
     private
   Monster                    -- another class
Tools-Outlines
...
```

Declaring a new Class

A class is created by sending a message to the class that is going to be its parent. For example

```
!Object subclass: #Monster
  instanceVariableNames: 'colour tummy'
  classVariableNames: ' '
  poolDictionaries: ' '
  category: 'Sesame-Street'!
```

This create the class `Monster` which is a subclass of `Object`. It has two instance variables (`colour` and `tummy`) and no class variables, and `Monster` is one of the classes in the category `Sesame-Street`. Ignore `poolDictionaries`.

Classes are created dynamically (during program execution).

Definition of CookieMonster

We no create a class `CookieMonster` as a subclass of `Monster`. It is also a member of `Sesame-Street`.

```
!Monster subclass: #CookieMonster
  instanceVariableNames: 'state hunger'
  classVariableNames: ' '
  poolDictionaries: ' '
  category: 'Sesame-Street'!
```

Defining Methods

Methods can be defined by sending a `methodsFor` message to a class. For example:

```
!Monster methodsFor: 'actions'!
eat: someItem
  self tummy add: someItem
!!

!Monster methodsFor: 'queries'!
isEmpty
 ^ self tummy isNil
!!
```

Defining Methods

```
!Monster methodsFor: 'access'!
colour
 ^ colour
!
colour: aSymbol
 colour := aSymbol
!
tummy
 ^ tummy
!
tummy: aCollection
 tummy := aCollection
!!
!Monster methodsFor: 'initialization'!
initialize
  self colour: #green.
  self tummy: Bag new
!!
```

Defining Class Methods

Class methods and variables in Smalltalk are like static methods and variables in Java.

There is on class method to define for `Monster`.

```
!Monster class  methodsFor: 'creation'!
new
  ^ super new initialize
!!
```

Cookie Monster Methods

Cookie Monsters inherit from Monster, but add more specific behaviour of their own.

```
!CookieMonster methodsFor: 'private'!
askForCookie
  ^ FillInTheBlank request: 'Give me a cookie'
!
complainAbout: anItem
  Transcript show: 'No want ', anItem printString.
  Transcript cr.
  self colour #red
!
isCookie: anItem
  ^ ((anItem = 'cookie') | (anItem = #cookie))
!!
```

Cookie Monster Methods

```
!CookieMonster methodsFor: 'actions'!
eat: aCookie
  super eat: aCookie.
  self colour: #green
!
nag
[self isAwake]
whileTrue:
[| item |
 item := self askForCookie.
 (self isCookie: item)
   ifTrue: [self eat: item]
   ifFalse: [self complainAbout: item].
   (self isFull) ifTrus: [self sleep]
!
sleep
  self state: #asleep.
  self hunger: 0
!
wakeUp
  self tummy: Bag new.
  self state: #awake.
  self hunger: (Random new next * 13).
  self nag.
!!
```

Cookie Monster Methods

```
!CookieMonster methodsFor: 'queries'!
isAsleep
  ^ state = #asleep
!
isAwake
  ^ self isAsleep not
!
isFull
  self isEmpty
    ifFalse: [^ self tummy size >= self hunger]
    ifTrue: [^false]
!!
!CookieMonster methodsFor: 'access'!
hunger
  ^ hunger
!
hunger: anIntegerNumberOfCookies
  hunger := anIntegerNumberOfCookies
!
state
  ^ state
!!
```