SUMMARY OF *SMALLTALK* MESSAGE FORMS AND INTENTIONS

Alan C. Kay
Xerox Palo Alto Research Center
August 1974

Of the three parts from which *SMALLTALK* communications are built--a message form, its intent, and the actual method used to carry out the intent--only the first two are shown in this note. The methods have been completely suppressed irregardless of how simple they might have been to say.[1]

SMALLTALK is based on a simple yet comprehensive model of the universe: there are only *objects* which contain local memory (or state); the objects communicate by sending and receiving *messages* (which are also objects): the objects are defined by a *class description* which is a collection of properties common to each member of the class and is specified by giving recognizers and replies to only those messages to which an object in the class is willing to respond.

MINOR DETAILS

a. *To load your disk pack:* If the switch on the left and the light on the right don't say "load", then push the switch to "load" and wait for the light. (A previous user will leave a note on the Keyboard if this is not OK.) Pull the door open and remove the disk if there. Slide your disk in, close the door and push switch to "run". In about 20 seconds, the "run" light should come on and your pack is loaded.

b. *To start SMALLTALKING:* Look for a little switch on the Keyboard...either behind near the cable or on the right underside. Push it. The screen should go blank and then ask for your name. Type it and push "doit" <line feed>; a prompt (□) will be displayed and you are ready to go.

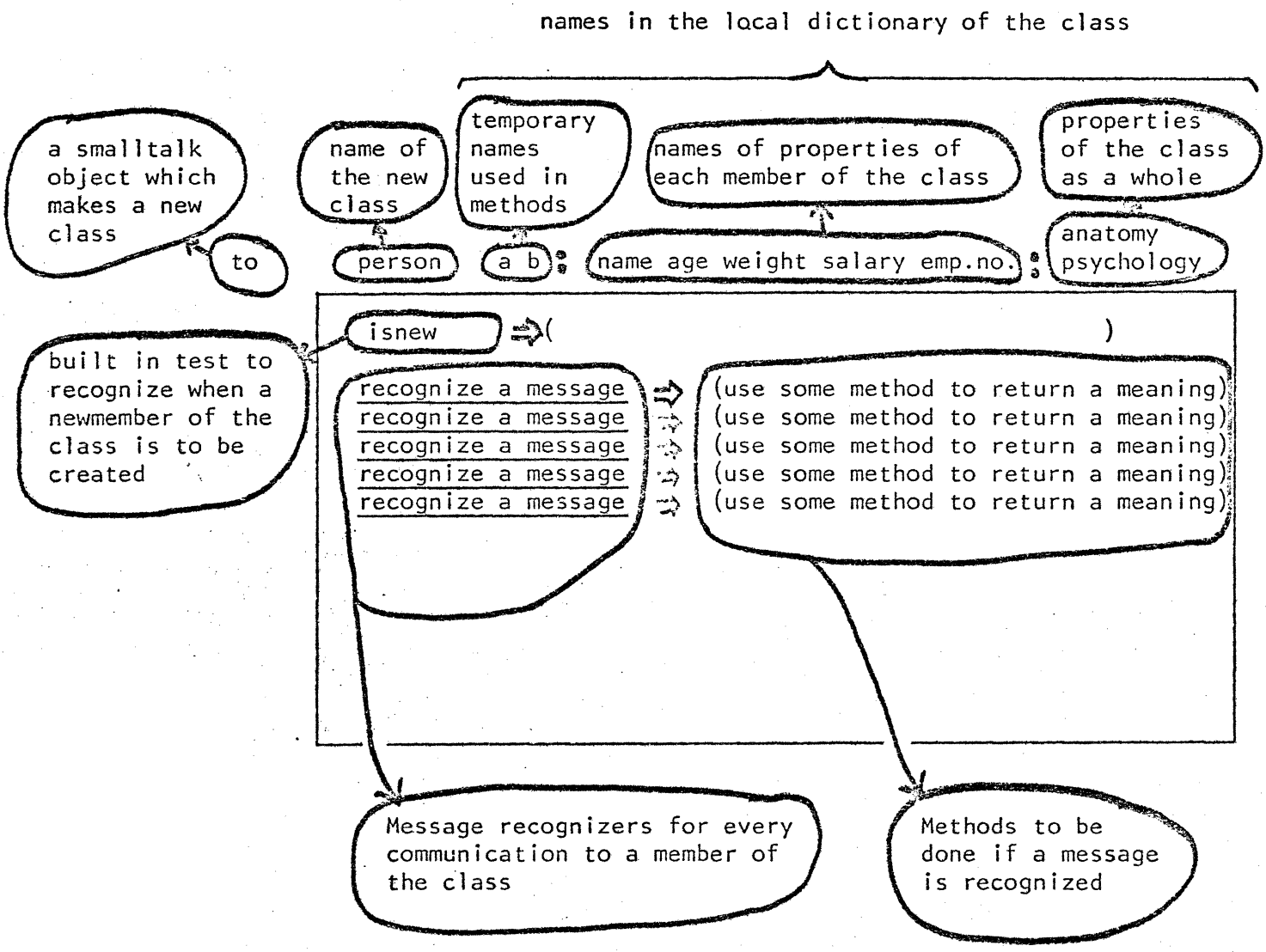
c. *The font editor* allows us to have considerable flexibility in choosing iconic representations for symbols. Unfortunately, the plastic tops of the keys do not change so readily. Here are the current correspondences between *SMALLTALK* symbols and keys.

---

[1] Only those messages which are relevant to the August class for non-programmers have been included; see SYSDEFS for the complete story.

" , % , : ,
! , ? , <linefeed> ,
. ; , ? ~ , $

MODEL OF A SMALLTALK CLASS DESCRIPTION



In the description which follows, '□□□', ' ', '...', are not currently part of SMALLTALK but are used to stand for arbitrary SMALLTALK objects such as '3', 'x', '←', '(a b c)', and so forth.

MESSAGE FORMS	*INTENTIONS*
(□□□...)	(,), may be used to group *SMALLTALK objects.*
√√√ □□□	*Send* √√√ the *message* □□□ ... (messages can be terminated by '.')
☞...	*Consider* the next thing *literally*--acts like '...' in English
◁...	*Look* to see if next thing is *literally* in the message. If not, return *false.*
∫	*Receive* the *value* of the next expression in the message.
∫	*Receive* the next thing in the message *literally.*
⇑ ...	*Return* the *value* of the next thing to the *sender.*
... ⇒(□□□)	If the *value* of ... is *not false*, then *evaluate* □□□ and *leave next higher level.* Otherwise continue to the right.

Objects will return themselves to the sender if no explicit ⇑ ... is given.

to name x y: printname

isnew ⇒(∫ ...)	Receive text of the new name into 'printname'. Return a unique reference to this object. A name starts with a letter and consists of any number of further letters and digits.
◁ is ⇒ ◁ ? ⇒ (...)	Answer is 'name'
◁ name ⇒ (...)	Answer is 'true'
∫.⇑ false	Answer is 'false'
◁ print ⇒ (...)	The text in 'printname' is sent to the display
◁← ⇒ (∫ ...)	Look up this name in nearest dictionary and *enter* the value ∫
◁ eval ⇒ (...)	Lookup this name in nearest dictionary and *evaluate* the value
◁ chars ⇒ (...)	Return the text in printname
◁ = ⇒ (∫ ...)	If this name is not the same as ∫,return false:

to number x y: val : nprint

'nprint' is common to all numbers and helps print them.

isnew ⇒ (§...)	Receive textual form of the number which is made into an internal form that is understood by the ALTO and entered into 'val'. A number starts with a digit and consists of zero or more further digits.
◁ is ⇒ ◁ ? ⇒ (...)	Answer is 'number'
◁ number⇒ (...)	Answer is 'true'
§·ﬂ false)	Answer is 'false'
◁ print ⇒ (...)	The internal form in 'val' is made back into text and sent to the display.
◁ + ⇒ (§...)	Return the *sum* of the number and §
◁ - ⇒ (§...)	Return the *difference* of the number and §
◁ * ⇒ (§...)	Return the *product* of the number and §
◁ / ⇒ (§...)	Return the *quotient* of the number and §.
◁ rem ⇒ (§...)	Return the *remainder* of the number and §
◁ < ⇒ (§...)	Return *false* if the number is not less than §, otherwise return the number
◁ > ⇒ (§...)	Return *false* if the number is not less than §; otherwise, return the number
◁ = ⇒ (§)	Return *false* if the number is not equal to §; otherwise, return the number

---

Number has quite a few more messages (see SYSDEFS).

to turtle z: x y dir pen ink

isnew ⇒ (...)	creates a new turtle
◁ is ⇒ ◁ ? ⇒ (...)	Answer is 'turtle'
◁ turtle ⇒ (...)	Answer is 'true'
∫.∫ false	Answer is 'false'
◁ print ⇒ (...)	Prints the current turtle state as text
◁ go ⇒ (∫...)	from the current position, traces ∫ distance in current direction
◁ right ⇒ (∫...)	changes the current direction by ∫ degrees
◁ goto ⇒(∫ ∫...)	from the current position, traces to x y position: ∫∫. Returns distance.
◁ penup ⇒ (...)	picks the "pen" up. No ink will flow
◁ pendn ⇒ (...)	puts the "pen down. Ink will flow
◁ home ⇒ (...)	take the turtle "home"; currently x y position 256 256.
◁ ink ⇒ (◁ white ⇒(...)	On the CSL graphics color
◁ black ⇒(...)	display, ink can be any of 256 colors.
◁ up ⇒ (...)	sets direction to 0.

Currently there is already a turtle instance at the conversational level which does not require a name, so "go 100," etc., will work directly. "Erase" will erase the turtle area only.

to repeat
  ∫... Evaluates ∫ over and over.

to done
  ◁ with ⇒ (∫...) Terminates the nearest "repeat" returning value ∫
  ... Terminates the nearest "repeat" returning nothing

to again
  ... Starts the nearest "repeat" again.

```
to do N
  ∫ ∫...  Repeats ∫ over and over ∫ number of times.
          'N' contains current loop count.
          'done' and 'again' work.

to for   variable start stop step exp
  ☞ variable ← ∫.                receive a name to vary
  ☞ start ← (◁← ⇒(∫)1)           get start number (1 if not mentioned)
  ☞ stop ← (◁ to ⇒ (∫)start)     get stop
                                 number (same as start if not mentioned)
  ☞ step ← (◁ by ⇒ (∫)1)         get step number (1 if not mentioned)
  ◁ do.                          'do' is optional
  ☞ exp ← ∫. ...                 get expression to be iterated and iterate it.
                                 'done' and 'again' work.

to vector

  isnew ⇒ (∫...)                 Creates a new vector, ∫ in length

  ◁ is ⇒ { ◁ ? ⇒ (...)           Answer is 'vector'
         { ◁ vector ⇒ (...)      Answer is 'true'
         { ∫ . ↑ false           Answer is 'false'

  ◁ print ⇒ (...)                Prints out current contexts of the vector
  ◁ length ⇒ (...)               Answer is the number of elements in this vector
  ◁ eval ⇒ (...)                 Answer is the SMALLTALK evaluation of
                                 the contents of this vector.

  ◁ [ ⇒ { ☞ lower bound←∫.
         { ◁ to ⇒ { ☞ upper bound←∫.
                  { ◁] ⇒          { ◁← ⇒          { ◁ all ⇒ (∫...)
                                                  { ☞ vector2← ∫.◁ [☞lbnd2←∫.to.☞ubnd2←∫.◁]
                                                  { ...
                                  { ◁ find ⇒      { ◁ first ⇒ { ◁ non ⇒ (∫...)
                                                              { ∫...
                                                  { ◁last ⇒   { ◁ non ⇒ (∫...)
                                                              { ∫...
                                                  { ...
                                  { ...
         { ◁]⇒   { ◁←⇒ (∫...)
                 { ...
```

to string

```
isnew ⇒ (§...)          Creates a new string, § characters long

◊ is ⇒ ◊ ? ⇒ (...)      Answer is 'string'
       ◊ vector ⇒ (...) Answer is 'true'
       § . ↑ false      Answer is 'false'

◊ print ⇒ (...)         Prints the current content of the string

◊ [ ⇒ @ lowerbound←§
      ◊ to ⇒ @ upper bound←§.
             ◊] ⇒ ◊← ⇒ ◊ all ⇒ (§...)
                       @ vector2←§.◊[@ lbnd2←§.to.@ubnd2←§.◊]
                       ...
                  ◊ find ⇒ ◊ first ⇒ ◊ non ⇒ (§...)
                                     §...
                           ◊ last ⇒ ◊ non ⇒ (§...)
                                     §...
                           ...
                  ...
      ◊]⇒ ◊←⇒ (§...)
          ...

◊ length ⇒ (...)        Answer is current length (in characters)
                        of this string

◊ eval ⇒ (...)          SMALLTALK will evaluate the contents of this
                        string as though it were typed in at the Keyboard

◊ = ⇒ (§...)            returns false if the string and § do
                        not contain the same character sequence

◊ + ⇒ (§...)            Makes a new string consisting of this one
                        and § appended.

◊ file ⇒ (...)          Fills string from the keyboard until a
                         ! is pushed.
```

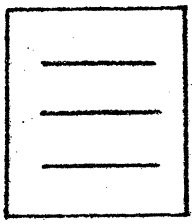
UTILITIES

to mx (...) returns the current x location of the mouse

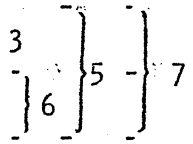
to my (...) returns the current y location of the mouse

to button n

§...



0 means no buttons on
1 check first button
2 check secondbutton
4 check third button



to in x w y h

§ § § §... Returns *true* if mouse is in the box; otherwise false.

to core (...) tells you how much room is left. Anything larger than 500 is good.

to kbck (...) keyboard check. If a character has been typed and is waiting 'kbck' will be true, otherwise false.

to kbd (...) waits until a character has been typed and then returns the character.

to read (...) shows a prompt (⍰), lets you type until a doit (!) and returns with a vector made up from what you typed.

to sp (...) prints a space

to cr (...) prints a carriage return

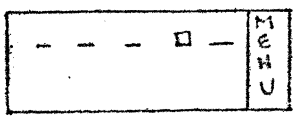
to show def prints out the class definition in a pretty format

§...

to edit *def*

§ . (◇ title ⇒ (...) starts edit with title line of the definition

... starts edit with body of definition



a *level* of the definition will be displayed (sub lists are represented by □); also a menu of commands which include

Command	'Bugs'	Explanation
Add		Appends what you type to the end
Insert	.	Inserts what you type in front
Replace	..	Replaces what you type between bugs
Delete	..	Deletes between the two bugs
Move	...	Move what is between the two bugs in front of final bug
Up	.	Raises a sub list one level
Push	..	Pushes what is between the two bugs down one level
Enter	.	Enters a sub list
Leave	.	Leaves a sublist
Exit		Exits from the editor

to addto def newdef adds more message handlers to a class definition
  ▯ ▯... addto number↑ (◁ max ⇒ (↑ x←▯.
    x ‹SELF ⇒ (◁ SELF) ⋔ x))

to obset ... i : vec size end

  isnew ⇒ (▯...) creates a new obset ▯ long.

  ◁ is ⇒ (◁ ? ⇒ (...) Answer is 'object'
    ◁ obset ⇒ (...) Answer is 'true'
    ▯ . ⋔ false Answer is 'false'

  ◁ print ⇒ (...) The contents of this obset are printed out.

  ◁ ← ⇒ (▯...) puts ▯ into the obset
  ◁ delete ⇒ (▯...) removes ▯ if it is there

  ◁ map ⇒ (▯...) takes the message and runs it 'length'
    number of times. A typical way to use it for an obset '*st*'
    might be, st map ↑(vec [i] hasmouse)
    This would cause every object in the obset '*st*' to be
    asked the question *hasmouse*

An obset which exists in the system when you start up, is '*defs*' which, every time you use *to*, is sent the name of the new class definition.

THE MIGHTY DISPFRAME

This class definition has been done and redone (by Diana Merry) many times in an attempt to build a very generally useful set of objects which will box text in rectangles, justify at word boundaries, find pointed at characters and words, and so on. Not every feature is presented here. Diana and SYSDEFS can tell you the rest.

There are two boxes: a 'window' and a 'frame'. The text (which is held in 'buf') will be automatically boxed within the frame boundaries. What you see on the screen is governed by the window boundaries: if they are equal to or larger than the frame you will see everything; otherwise, only that part of the frame which is in the window will be seen.

'reply' will contain useful things which Diana has discovered such as:

reply=	0	some of the window is visible and some of the frame
	1	window is not on display. x> 606 and/or y> 808
	2	frame is completely out of the window
	11	frame height has been automatically increased to accommodate the text in 'buf'
	12	means that window bottom has been overflowed and that scrolling has happened
	13	means both 11 and 12

to dispframe input: winx winwd winy winwd frmx frmht buf ...
reply editor :...

isnew ⇒ (▯▯ ▯ ▯ ▯...)	The first four numbers set both the window and frame boxes to the same size. The final part of the message is the string which goes into 'buf'.
◁ ← ⇒ (▯...)	If ▯ is a character or a string it will be automatically appended to 'buf'
◁ show ⇒ (...)	Show clears the intersection of window and frame and displays 'buf' boxed in the frame
◁ clear ⇒ (...)	clears the intersection of window and frame and clears 'buf'.
◁ fclear ⇒ (...)	just clears the intersection of window and frame. Leaves buf alone
◁ scroll ⇒ (...)	scrolls 1 time. The old first line is lost.
◁ mfindc ⇒ (▯ ▯...)	...mouse find character... ▯ *and* ▯ are x and y

locations (usually gotten from the mouse). If these coincide with a visible character in this frame the index of that character in 'buf' will be returned. -1 means you are in the frame but *after* all the characters. -2 means you are not in the frame at all.

◁ mfindt ⇒ (§ §...) ...mouse find token... Works just like 'mfindc' except words are looked for.

◁ frame ⇒ (...) draws a box around the frame.

◁ knows ⇒ (ev) calls SMALLTALK from inside the frame.

FILES AND FILING

Another area which is handled *very* comprehensively by this system. Class 'file' will hardly be presented here. Instead the four or five things which will cover most needs are shown. Steve Weyer and SYSDEFS can tell you more.

to filout

(◁ pretty ⇒ (...)) If *pretty* is there file looks nicer but filout takes longer.

§... a string for a file name.

(◁ add ⇒ (...)) If *add* is there, definitions will be added. to end of file. Otherwise, the old stuff in the file will be clobbered.

§... If this is present, it is a vector of names of class definitions. Otherwise, all the names in *defs* will be read out.

to filin

(§...) The file named by the string message will be *read* into SMALLTALK just as though you typed it.

to type

§... A string for file name. Types a file (on the screen) from one which was previously filed out.

to dir (...) prints out the SMALLTALK part of your file directory. *dpO list* will show the complete directory