Writing Presentations for DesigNet

Written by A. Boulanger
24 December 1984

1. Introduction

This is a tutorial on writing presentations for DesigNet. There should be a implementation manual on the presenter but this code is not released and considered experimental. For now this tutorial is offered to enable you to write new presentations if needed.

1.1 The Presenter and Presentations

The presenter provides a naive user view of objects. The objects presented could be frames, or flavor instances, atoms with properties, etc. What the objects could represent could be real world objects, actions on objects, actions applied to an object, pieces of discourse, etc. One view of the presenter is a text splicer. From represented objects, relationships between objects, and pieces of text stored within *presentations* on objects, the presenter splices text together into a hopefully coherent whole. There are more sophisticated ways of going from representations to discourse such as going to an intermediate "parse tree" form first (Weiner ref), but the spirit of the presenter is one of a poor man's presenter. The way it is implemented, it can't handle such things as pronounal reference etc.

The presenter also provides a *hypertext* (Ref) view of the world. Presentation is mouse sensitive and this allows the user to explore the presented object or relationships between the presented object and other objects to the degree he wants.

The presenter also provides a mechanism to see the world from multiple views. This is implemented as a system of **presentation keys** that one associates with pieces of text stored in the presentation. When presenting an object, one selects which keys to use, unlocking the right combination of text for the occasion. The associated text for a key is termed a *presentation paradigm* or *PP* for short.

The text is built up as a list or strings, symbols, objects, or sublists. The window

associated with the presenter knows how to justify text. The presentation can be thought of as a set of commands for formatting the associated *presenter* window. Strings are broken up into words and output. If a symbol which has a value being an instance, or the instance itself, the :**present** message is sent to it. Side effects are possible by having a function call as part of the text. The function could highlight something on the graphic screen for example. The function calls could affect the associated window as well. The text justifier is designed to figure how to justify text from the current cursor position, not from an internal model of where the current position is. There is a command to do a bulletized or itemized list. There is a command to make something mouse sensitive. There is a conditional command as well as a command that is conditional on a subkey. The conditional subkey is a way of communicating context to recursed presentation.

The list of commands are:

EVAL-AND-REPLACE *eval-form directions*
Evaluate the form to return a presentation to be used with **DIRECTIONS**.

ITEMIZE *directions eval-form &rest itemize-options*
Constructs an itemized or bulletized list. **DIRECTIONS** are relayed to any presented objects. **EVAL-FORM** is a form to eval to get a list of presentations to be itemized. **ITEMIZE-OPTIONS** are two optional arguments. The first one, marker, can be the atom **number** - which indicates to use numbers for the itemization, a character - use that character in the current font, a list (character font) - use that character in the specified font. The default is a triangular bullet. The second argument is a cutoff for the number of list items and defaults to 99.

SERIES *directions eval-form &rest series-options*
Not yet implemented, but is for comma separated lists.

PRESENT *evaled-object &rest directions*
Present the object using directions.

FORMAT *format-string &rest format-args*
Format and output the resulting string.

WHEN *test &rest subpart*
If **TEST** is true then **SUBPART** is processed.

KEY *key &rest subpart*

If KEY is a member of the directions for this presentation, then subpart is processed.

MOUSE-ITEM *string evaled-alist-key evaled-item*

Make string mouse sensitive. EVALED-ALIST-KEY is evaluated to get the associated alist-key for the string (see window documentation if you don't know what this is). EVALED-ITEM is evaled to get the associated item for the string.

In addition if there is what looks to be a function call to the following functions, they are done:

PROG	PROG1	PROGN
DO	DOLIST	EVAL
LOOP	MAPC	SEND
APPLY		

These functions can be used to do something on the graphic screen, or effect the formatting of the window used for the presentation. Thus, the user has tremendous control over the format of the presentation.

The special variable **stream** is bound to the window that presentation is going to. The special variable **object** is bound to the instance being presented if what is being presented is a flavor instance. There are four special symbols that is used to say the name of the object being presented within the presentation for that object:

*	Name of object uncapitalized.
**	Name of object capitalized.
***	Same as * but made mouse sensitive.
****	Same as ** but made mouse sensitive.

There is a secondary message that is used on occasion to help build up a presentation. The :say message is sent to objects to return a form that is in the format of a presentation. This is a slight generalization of the normal meaning of :say adopted locally which is to return a string saying something about an object.

There is one other part of a presentation that directs how the mouse sensitive name of an object is wired up to the mouse. This is called the **mouse-pps** for mouse presentation paradigms. Within this there is an association between a PP and what should go into the mode line and the pop up menu when choosing it. The first entry is the default when you left click. An example is:

```
(:mouse-pps (:all "Present location, nodes, pads, hosts & terminals"
             :location "Present the location"
             :nodes "Present the nodes at site"
             :pads "Present pads at site"
             :hosts "Present the hosts at site"
             :terminals "Present the terminals at site")
```

The model of a presentation is that it has a prolog, a body, and an epilog. Each of these are built up separately considering each of the presentation keys.

This format for presentation is a first pass and may change later. Note that the syntax is ambiguous in one respect. Empty strings can be used to disambiguate. See the example presentation below

Below is a description of the syntax for presentations (comments from the file NET-DESIGN;INTERFACE;PRESENT LISP):

Presentation Syntax

```
The format of the presentation is:
(
 (:prolog PART)
 (:body PARTS)
 (:epilog PART))
At least one of a prolog, body, or epilog must be present.
```

```
The syntax for PARTS is:
(optional PART PART PART ... )
or
((key1 PARTS)
 (key2 PARTS)
 (key3 PARTS)
      .
      .
 (keyn PARTS))
where optional is ITEMIZE, SERIES, PRESENT, WHEN, FORMAT, or MOUSE-ITEM


The syntax for PART is:
(SUBPART SUBPART .....)
or
((key1 PART)
 (key2 PART)
 (key3 PART)
      .
      .
 (keyn PART))

key1, key2,..., keyn are dispatched to with corresponding names in the directions
structure. (See below.)

 +++ revise this....
 A SUBPART can be a:
 - string: output string
 - a list with the head element being: PROG PROG1 PROGN DO DOLIST
   EVAL LOOP MAPC SEND APPLY: evaluate form.
 - EVAL-AND-REPLACE,SERIES,ITEMIZE,PRESENT,FORMAT,WHEN,KEY,MOUSE-ITEM:
   special command
 - other list: treat it as a PART
 - symbol: get its value and if there is no value convert to string and output else if:
       - value is an object: send ':PRESENT to it.
       - value is a string: output it.
       - value is a file pathname: open file and dump contents to user.
```

For DesigNet, the presentation for design objects is stored on the **instance-presentation** instance variable of the associated class. There are two other presentation associated instance variables on design classes: **prototype-presentation** and **class-presentation**. Prototype presentation, is for presenting a *prototypical* member of the class. Class presentation is for presenting the class itself. These are not used currently.

Presentation for Sites

```
((:prolog ((:simple (****))
           (:location
            (**** "is located at"
                  (format nil "~A. " (send object ':say-location))))
           (:all
            ("" (present object :force  :location :nodes
                        :pads :hosts :terminals :multiplexors
                        :neighborhood-cache)))))
 (:body   ((:nodes
            ("" (when (send object ':nodes)
                  **** "has the following nodes:"
                  (itemize (:simple) (send object ':say-set 'nodes)))))
           (:pads
            ("" (when (send object ':pads)
                  **** "has the following pads:"
                  (itemize (:simple) (send object ':say-set 'pads)))))
           (:hosts
            ("" (when (send object ':hosts)
                  **** "has the following hosts:"
                  (itemize (:simple) (send object ':say-set 'hosts)))))
           (:terminals
            ("" (when (send object ':terminals)
                  **** "has the following terminals:"
                  (itemize (:simple) (send object ':say-set 'terminals)))))
           (:multiplexors
            ("" (when (send object ':multiplexors)
                  **** "has the following multiplexors:"
                  (itemize (:simple) (send object ':say-set 'multiplexors))))) .
           (:neighborhood-cache
            ("Cost to nearest neighbors of"
             **** (format nil "for ~A service is:" *service-for-neighborhood-cost*)
             (itemize (:simple) (send object :say-neighborhood-cache))))))
 (:mouse-pps (:all "Present location, nodes, pads, hosts & terminals"
              :location "Present the location"
              :nodes "Present the nodes at site"
              :pads "Present pads at site"
              :hosts "Present the hosts at site"
              :terminals "Present the terminals at site")))
```

A typical say method for site is:

```
(defmethod (site :say-neighborhood-cache) ()
  (loop for (cost . site) in neighborhood-cache
        collect '("" (present ',site :simple) ,(format nil "~A" cost))))
```

This produced for an example instance:

An Example Presentation

Top of Presentation

Vanilla Site H K Ma is located at 4422 V and 1269 H. Vanilla Site H K Ma has the following nodes:

C 30 E N1 H K Ma

Vanilla Site H K Ma has the following hosts:

- X25 Host H1 H K Ma
- X25 Host H2 H K Ma
- X25 Host H3 H K Ma

Cost to nearest neighbors of Vanilla Site H K Ma for 9600-DDS service is:

- Vanilla Site Boston 747
- Vanilla Site London Ma 774
- Vanilla Site New York{1} 1022
- Vanilla Site Newark Nj 1030
- Vanilla Site Chicago{1} 1579
- Vanilla Site Chicago{2} 1606
- Vanilla Site Atlanta 1645
- Vanilla Site Chicago{3} 1660
- Vanilla Site Chicago{4} 1687
- Vanilla Site Elgin Il 1885

Presentation

Bottom of Presentation