ViSta™

THE VISUAL STATISTICS SYSTEM

Developing Plugins

Forrest W. Young

The Visual Statistics Project

www.visualstats.org

Forrest W. Young
Psychometric Laboratory
University of North Carolina
CB 3270 Davie Hall
Chapel Hill, NC 27599-3270 USA
forrest@unc.edu

Pedro Valero & Gabriel Molina
Institute of Traffic and Road Safety
Universitat de València
C/ Hugo de Moncada 4. Entresuelo.
Valencia, 46010, España (Spain)
pedro.valero-mora@uv.es
gabriel.molina@uv.es

June, 2002 - ViSta 7

first, you see your data for what they seem to be
    then, you ask them for the truth -
    are you what you seem to me?

you see with broad expanse
    yet ask with narrowed power
        you see and ask and see
        and ask and see ... and ask

with brush you paint the possibilities
    with pen you scribe the probabilities
for in pictures we find insight
    while in numbers we find strength

Young, Forrest W. “ViSta - The Visual Statistics System: Developing Plugins” (June, 2002). The Visual Statistics Project. Chapel Hill, NC and Valencia, Spain.

Developing Plugins

Forrest W.Young

Abstract

This monograph is designed to guide those who are developing new plugins for ViSta, the Visual Statistics system (Young, 1996). The monograph overviews the inheritance structure of ViSta’s data, transformation, model and plugin object systems; discusses ViSta's global variables; presents details on the methods, functions and messages that can be used with statistical objects; reviews the steps taken during the construction of an instance of a model object; and presents a detailed example of how one of ViSta’s model object plugins was developed. This monograph assumes you are familiar with Lisp-Stat (Tierney, 1990).

1 About Developing New Features for ViSta

ViSta is a freely redistributable software program, based on a moderated open source development model. That is, those who wish to enhance ViSta can do so. Some of the code is open to all, while other portions require prior approval to gain access.

The code consists of a core engine plus plugins and addons. This design lets ViSta be both stable and expandable: The core is stable, while the plugins and addons provide the path to growth. This architecture also provides for an obvious oganization of ViSta developers into Application developers and System developers. Applications developers develop new plugins and addons, whereas system developers can also enhance ViSta's core engine.

Application Development: All of the code and tools that you will need to be a ViSta Application Developer are on your machine. You can proceed to develop your plugin or addon application without coordinating your efforts with those of other application developers.

However, please join the vista@unc.edu newsgroup for information on what other folks are doing, and to leave information about what you are doing. This way, duplication of effort can be avoided, and the user and developer community can be kept up-to-date.

If you wish, you can submit your plugin to us for distribution by visualstats.org. You are, of course, free to distribute your plugin independently from visualstats.org. Submitting a plugin to us is much like submitting a paper for publication. When your plugin is ready, just email the installation module to forrest@unc.edu.

The information you send should include code, data, examples and documentation. The ViSta Editorial Board will review and make a recommendation concerning distribution. If it is approved for distribution it will be included on the visualstats.org website (and on our mirror sites), and links will be made so that it can be downloaded.

System Development: ViSta System Developers have access to the entire source code, and can make changes to any portion of the system. But this isn't a free-for-all. Rather, because of the critical nature of systems development, and the importance of the core engine to the entire system, the system development effort is a coordinated effort. The effort is coordinated through the use of CVS, a version control system.

CVS permits individuals who are part of a widely distributed development effort to work independently and simultaneously on a common set of code. The code is on your machine, where your CVS client coordinates your development with that of other developers, all the while permitting you to work independently from other developers. When you have completed your changes, the central CVS server will review all code changes to detect changes which conflict with those made by other system developers. You will then need to resolve these conflicts before the changes are accepted.

Contact Forrest Young if you wish to download ViSta from the ViSta CVS server at the University of North Carolina at Chapel Hill.

2 ViSta's Statistical Object System

ViSta's statistical object system is based on Lisp-Stat's (Tierney, 1990) object-oriented programming system. The statistical object system consists of prototype objects which have methods (pieces of LispStat object-oriented code) and information designed with statistical data analysis in mind. The prototype objects permit the instantiation of the statistical objects which are represented on the WorkMap by icons. The objects use the methods to send and receive messages that access and manipulate the information. We discuss the statistical object system in this chapter.

ViSta's prototype statistical objects fall into four major families: data objects, transformation objects, model objects and plugin objects. The inheritance relationships between these objects is shown in Figure 1. While there are two different kinds of data prototypes — multivariate and matrix (table data objects have been discontinued), there are many different transformation, model and plugin objects, each specialized for a specific statistical situation. For example, there is a plugin model object for analysis of variance prototype, two for regression, one for univariate analysis prototype, etc.

The statistical prototypes are related to each other hierarchically to take advantage of the Lisp-Stat object system's inheritance feature. The hierarchical structure is shown in Figure 1.

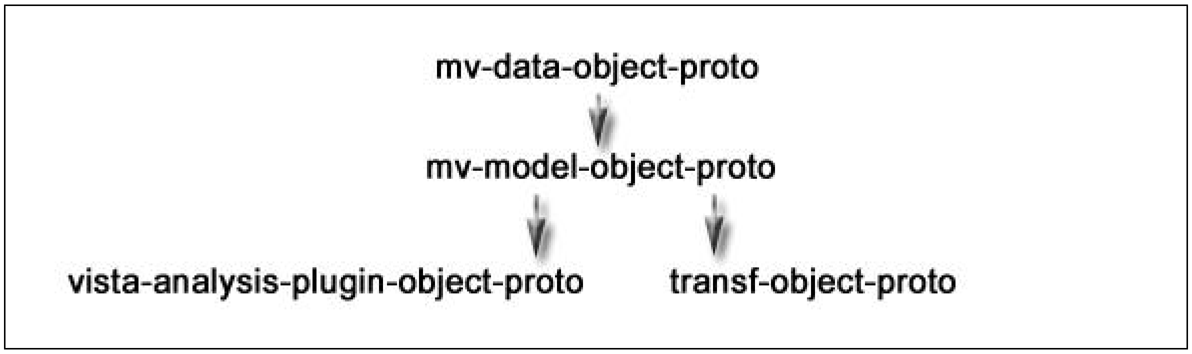


Figure 1: Statistical Object Prototype Hierarchy

3 Global Variables

ViSta defines a number of global variables — variables whose values are always available for use. We present them in this section

One of the statistical objects is always the "current object". The current object is indicated in the workmap by the high-lighted icon. It is also always pointed to by the global variable `*current-object*`, to which messages can be sent. The current object can be changed by clicking on a new icon on the workmap, by choosing a new object from the data or model menus, or by using the `setcd` or `setcm` functions described in sections 4.2 and 5.1. The global variables `*current-data*` and `*current-model*` are also discussed in those sections. Note that there are several single-characters symbols which have been defined to be global statistical object variables. In particular:

- the symbol @ is equivalent to *current-object*
- the symbol $ is equivalent to *current-data*
- the symbol $$ is equivalent to *current-model*

The name of the object as it appears as the icon title, when the icon is selected, is also a symbolic reference to the object.

There are several global variables that specify directory paths. These include `*default-path*`, which points to the directory containing the LispStat load module; `*vista-dir-name*` whose value is the directory containing the basic vista code and sub-directories. The variables `*data-path*`, `*help-path*` specify the paths to ViSta's data and help directories.

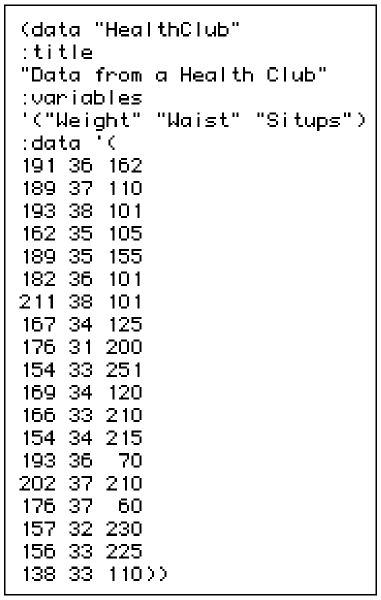
In addition, there are global variables whose values point to various windows and menus. These include `*workmap*`, `*listener*`, `*obs-window*` (also referred to by `*mat-window*`), `*var-window*`, `*edit-menu*`, `*file-menu*`, `*command-menu*`, `*data-menu*`, `*trans-menu*`, `*analyze-menu*` and `*model-menu*`. In addition, `*vista*` points to the vista system object, and `*help-window*` points to the help message window. Many of these are nil until used.

4 Data Objects

Data objects are used to define data used by ViSta. In this section we discuss the `data` function for defining data objects, and the keywords which can be used with the function. We then discuss data object types, the concept of the current data object, and the messages which can be sent to an existing data object. Note that the `(load-data)` function may be used to load data into ViSta from a ViSta datafile, the `(open-data)` function loads data into ViSta from a ViSta datafile and then displays it as a datasheet, and the `(import-data)` function imports data contained in a text file. Each of these functions takes an optional string argument to specify the name of the file containing the data. Thus, it is possible to load data from the data directory, by using the global variable `*data-dir-name*`, with the statement

    (load-data (strcat *data-dir-name* "cars.lsp"))

These menu items are all discussed in Chapter 3.



```
(data "HealthClub"
:title
"Data from a Health Club"
:variables
'("Weight" "Waist" "Situps")
:data '(
191 36 162
189 37 110
193 38 101
162 35 105
189 35 155
182 36 101
211 38 101
167 34 125
176 31 200
154 33 251
169 34 120
166 33 210
154 34 215
193 36  70
202 37 210
176 37  60
157 32 230
156 33 225
138 33 110))
```

Figure 2: Data function for multivariate data

4.1 The Data Function

The several ways of creating data that we discussed in section 3.3 of Chapter 3 of Young (1996) all create data by using the `data` function. This is an object-constructor function which creates an instance of a data object. It can create either multivariate or matrix (table data objects are no longer supported). The specifics of the `data` function for each type of data object are explained in the next two subsections.

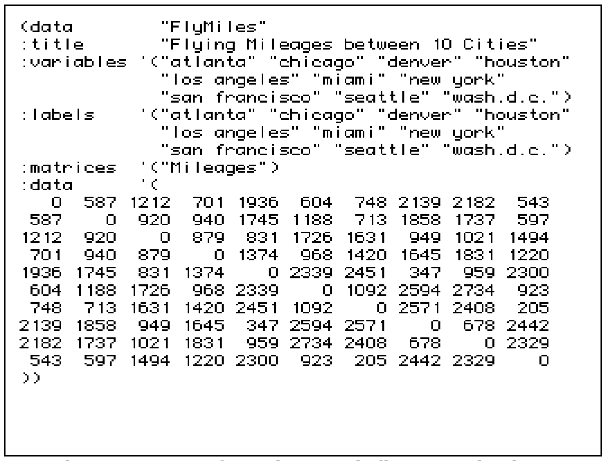
4.1.1 Multivariate Data

An example of the `data` function being used to create an instance of a multivariate data object is shown in Figure 2. The `data` function has one initial required argument and two required keyword arguments. The initial required argument is a string that is used to name the data. In the Figure, this is "`HealthClub`". Any characters may be used in this string, including spaces. In addition to the name, you must also use the `:variables` and `:data` keywords, following each with a list of elements. The `:variables` keyword specifies the names of the variables and (indirectly) the number of variables. It is followed by a list of character strings which are the variable names. The `:data` keyword specifies the list of data values. If *n* is the number of variables, the first *n* elements of the list are the values for the first observation (row) of the data, the next *n* elements are the values for the second observation, etc. The total number of elements of the data list must be an exact integer multiple of the number of variables.

There are several optional keywords which may be used with the `data` function, one of which, the `:title` keyword, is shown in the example. This keyword is followed by a character string to specify the data-object's title, which is used in various windows. When not specified, the data title is `Untitled Data Object`. The `:types` keyword, which is followed by a list of character strings, one for each variable, specifies the type of each variable. The type may be "`category`", "`ordinal`", or "`numeric`". If this keyword is not used, all variables are assumed to be numeric. Finally, the `:labels` keyword, which is followed by a list of character strings, specifies the label for each observation (row of the data). If not used, the observation labels are `Obs0`, `Obs1`, etc.

4.1.2 Matrix Data

An example of a `data` function for creating an instance of a matrix data object is shown in Figure 3. A matrix data object is a data object whose data consist of one or more matrices of data. ViSta matrices are not as general as those in Lispstat: They must be square, and the row and columns must refer to the same set of things, which are named by the `:variables` keyword.



```
(data       "FlyMiles"
:title      "Flying Mileages between 10 Cities"
:variables '("atlanta" "chicago" "denver" "houston"
            "los angeles" "miami" "new york"
            "san francisco" "seattle" "wash.d.c.")
:labels    '("atlanta" "chicago" "denver" "houston"
            "los angeles" "miami" "new york"
            "san francisco" "seattle" "wash.d.c.")
:matrices  '("Mileages")
:data      '(
   0  587 1212  701 1936  604  748 2139 2182  543
 587    0  920  940 1745 1188  713 1858 1737  597
1212  920    0  879  831 1726 1631  949 1021 1494
 701  940  879    0 1374  968 1420 1645 1831 1220
1936 1745  831 1374    0 2339 2451  347  959 2300
 604 1188 1726  968 2339    0 1092 2594 2734  923
 748  713 1631 1420 2451 1092    0 2571 2408  205
2139 1858  949 1645  347 2594 2571    0  678 2442
2182 1737 1021 1831  959 2734 2408  678    0 2329
 543  597 1494 1220 2300  923  205 2442 2329    0
))
```

Figure 3: Data function to define matrix data.

Matrix data objects are defined in exactly the same way as multivariate data objects, except that the `:matrices` keyword must be used. This keyword, which is followed by a list of character strings, specifies that the data are matrix data, and specifies the names (and, indirectly, the number) of the matrices. The :`matrices` keyword is required for matrix data, and must not be used for other types of data. The `:shapes` keyword, which is followed by a list of character strings, one for each matrix, specifies the shape of each matrix. The shapes may be `"symmetric"`, or `"asymmetric"`. If shapes are not specified, then all matrices are assumed to be symmetric. In the example shown in Figure 3, both the `:shapes` and the `:types` keywords are not needed, because they both specify characteristics of the data which are assumed by default. If the data consists of several matrices, then the `:data` list consists of all the values for the first matrix, followed by all the values for the second matrix, etc.

4.2 Current Data and Data Object Names

One of the data objects is always the "current" data. The current data is indicated in the data menu by the checked menu item, if data object names are being shown. It is also indicated on the workmap as the high-lighted data icon, if one is high-lighted. The current data's object identification can always be found in the global variable `*current-data*`, to which messages can always be sent. The current data can be changed by clicking on a new data icon, or by choosing a new item of the data menu. Typing $data will present a list of all data objects.

The `data` function defines a variable whose name is the name of the data object, and whose value is the object's identification information. Using this name, the current data can be changed from the keyboard by using the `setcd` (set current data) function. For example, if there is a data object named `cars`, you make it the current data with `(setcd cars)`.

4.3 Data Object Messages and Functions

Messages can be sent to data objects. The messages can be sent to `*current-data*` or to the name of a specific data object. For example, the message `(send *current-data* :data)` returns the list of data values for the current data. If there is a data object named `cars`, you could type `(send cars :data)` to see a listing of its data values, whether or not it is the current data.

Slot Information Messages: There is a group of messages[1] which can be used to obtain information about the data object. For example `(send cars :title)` causes the data-object to tell you its title. In addition to the `:title` message, messages in this group are `:nobs`, which returns the number of observations in the data object; `:nvar`, which returns the number of variables in the object; `:variables`, which returns the names of the variables; `:labels`, which returns the labels of the observations; `:types`, which returns the type of each variable; `:data`, which returns the data as a list; `:data-matrix`, which returns the data as a matrix; and `:name` which returns the name of the data object. The messages `:obs-states` and `:var-states` return the state of each observation or variable in the observation window (the states can be normal, selected or invisible). For matrix data, the message `:matrices` can be used to obtain the names of the matrices, and you can use the messages `:nmat, :mat-states` and `:shapes` (to see whether the matrices are symmetric or asymmetric). For table data, you can use the messages `:nways`, `:nclasses`, `:ncells`, `:cellfreqs`, `:classes`, `:source-names`, `:level-names`; and `:indicator-matrices`.

Data Menu Item Messages and Functions: There is another group of messages which perform the same actions as those performed by items of the data menu. The name of each message corresponds to the name of a menu item. For example, the data menu's Save Data menu item corresponds to `(send *current-data* :save-data).` The messages include `:save-data`, which saves data into a ViSta datafile specified in a dialog-box; and `:create-data`, which creates a new data object from the active portion of the current data. These two messages can be followed by a string argument to name the file or data object. If the string is not specified, a dialog box is presented. The message `:browse-data` shows the datasheet; `:edit-data` shows the datasheet and enables editing of the data; `:report-data` creates a report (listing) of the data; `:list-variables` shows a window with a list of the variables in the current data (you can also use `:list-vars` or `:list-var`); and `:list-observations` (or `:list-obs`) shows a window with a list of the observations in the current data. For matrix data you may use `:list-matrices` (or `:list-mats` or `:list-mat`) to show a window with a list of the matrices in the data. For table data you may use `:list-cells`.

Each message in the previous paragraph also has a short form, called a generic function. For example, `(save-data)` is short for `(send *current-data* :save-data).` The long-form message can be used to send messages to a specifically named data object that is not the current data. The short-form generic function always sends the message to the current data.

There are four menu items which have a short-form that effects the `*current-data*`, but do not have a long-form message that can be sent to data objects that are not current. These are `(visualize-data)`, which visualizes the active data, `(merge-variables)`, which merges variables using the active data in the current data and the active data in the previously current data; `(merge-observations)`, which merges observations using the active data in the current data and the previously current data, and `(merge-matrices)`, which merges matrices using the active data in the current and previously current data (both have to be matrix data). The merge functions take an optional argument which is a character string that is used to name the new data object. If no character string is present, a dialog box is presented to obtain the name.

You can also use the `:summarize-data` message or the `(summarize-data)` function to see a summary report (listing of summary statistics) of the data. The message and function each have five keyword arguments,

1. These messages are slot-accessor messages, and therefore can also be used to change the information in data object slots. This should not be done since ViSta will not work correctly if the information is changed.

each of which must be followed with `t` or `nil` (the default). The arguments are `:moments`, `:quartiles`, `:ranges`, `:correlations`, `:covariances`, and `:dialog`. The last argument determines if a dialog box is presented to obtain the desired types of statistics. The others determine which statistics are reported if no dialog is shown.

Active Data Messages: There are numerous messages which deal with the “active” data: i.e., the subset of the current data which is specified by selections in the windows which list observations, variables and matrices. The active data consists of the subset of data elements whose variable (or matrix) names and observation labels which appear in the "Vars" (or “Mats”) and “Obs” windows, or, if any names or labels are selected in a window, those which are selected as well as visible. The concept of active data does not apply to table data, which may not be subsetted.

The messages `:select-variables`, and `:select-observations`, and the short-form functions `(select-variables)` and `(select-observations)` can be used to select the variables or observations which are to be active. Similarly, with matrix data you may select matrices with the message `:select-matrices` or the function `(select-matrices)`. Each message or function must be followed by a list of strings specifying the names of the variables or matrices or the labels of the observations that are to be activated. For example, you can select all of the variables with

```
   (select-variables (send *current-data* :variables)).
```

There are several messages that provide information about the selected subset of data. Many of these messages take an argument that is a list of symbols (not strings) that specifies the variable “type”. The type symbols are `labels`, `category`, `ordinal`, `numeric` or `all` (in upper, lower or mixed-case). The messages using these symbols return information about the data elements whose active variables have types which match the specified types. For example, you could type

```
   (send cars :active-data ‘(ordinal category))
```

to see a list of the data elements whose variables are active and are either ordinal or categorical. The messages that use type-symbols are `:active-data` (to get a list of the active portion of the data), `:active-data-matrix` (to get a matrix of the active portion of the data), `:active-nvar` (to find out how many active variables there are), `:active-variables` (to obtain a list of the names of the active variables), and `:active-types` (to obtain a list of the types of the active variables). In addition, the messages `:active-matrices`, `:active-shapes`, and `:active-nmat` take an argument that is a list of the shape symbols `all`, `symmetric`, and `asymmetric`. Finally, there are two active data messages `:active-labels` and `:active-nobs` do not have an argument. These messages can be used to get a list of the labels of the active observations, or the number of active observations. Note that there are no short-form functions for any of the active data messages mentioned in this paragraph.

Miscellaneous Messages: The message `:variable` reports values of the variable whose name is the argument of the message. The messages `:means`, `:standard-deviations`, `:variances`, `:skewnesses`, `:kurtoses`, `:minimums`, `:medians`, `:maximums`, `:ranges:`, `:mid-ranges`, `:interquartile-ranges`, and `covariance-matrix` report simple statistics of the active data. There is no short-form function for these messages.

4.4 Variable types and data types

Each variable in a ViSta data object has a "type", referred to as the "variable type" of the variable. Each data object also has a "type", called the "datatype". Don't confuse these two "types". Variable types are fundamental. They are imutable characteristics of the variables, there being only two types: Category and Numeric (ordinal is no longer supported). On the other hand, datatype is a characteristic of a dataset (or, more accurately, of a collection of variables) which is derived from the variable types and from other information.

Each ViSta data object has a datatype. The datatype determines ViSta's default analyses and visualizations, and limits the choice of actions you can take to those that are likely to be reasonable. Datatypes are meant to simplify the user's experience... they allow ViSta to make a more educated guess about what should be done with the data, and provide an unobtrusive way of guiding the user by limiting choices.

There are 11 data types recognized by ViSta. The datatype depends on whether the values for numeric variables represent quantity, frequency[2] or relatedness, whether the data contain missing values, and on the mix of variable types. The 11 data types are:

1. MISSING - The data have one or more NIL elements
2. MATRIX - A datatype where the basic datum is a relation. The observations and variables refer to the same things, the data elements are relational, specifying the relation (correlation, distance, covariance) between pairs of the things.

Six of the datatypes require the basic datum to be a quantity:

3. CATEGORY - There are only category variables
4. UNIVARIATE - There is one variable and it is non-frequency numeric
5. BIVARIATE - There are two variables and they are both non-frequency numeric
6. MULTIVARIATE - There are more than two variables. All are non-frequency numeric
7. CLASSIFICATION - There is exactly 1 non-frequency numeric variable and there are 1 or more category variables
8. GENERAL - When none of the above defintions is satisfied the datatype is "general" (for quantity data)

Three of the datatypes are defined for a basic datum that is a frequency:

9. FREQUENCY - All the variables are numeric frequency variables
10. FREQCLASS - There is exactly 1 numeric frequency variable and there are 1 or more category variables
11. CROSSTABS - There are 1 or more numeric frequency variables 1 and or more category variables

Formally, the datatypes are defined to be

1. MISSING if the data contain one or more NIL elements.
2. MATRIX if matrices are present without NIL elements.
3. If neither of the above is true, then the datatype is defined according to the number of category and numeric variables in the data, and by whether the data are frequencies or not. Specifically:

Table 1: Datatype as a function of number and type of variables

Number of Category Variables	Frequency Data?	Number of Numeric Variables			
		0	1	2	>2
0	**No**	-	univariate	bivariate	multivariate
0	**Yes**	-	frequency	frequency	frequency
>0	**No**	category	class	general	general
>0	**Yes**	category	freq-class	crosstabs	crosstabs

2. Note that the definition of "frequency" data has been retro-fitted to ViSta and is a bit of a kludge. Specifically data are frequencies if explicitly declared so in the datacode, or if all numeric variables are named "Freq".

5 Model Objects

Model objects are used to define models in ViSta. There are several functions for constructing instances of model objects, one function for each of the model object prototypes given in Figure 1. As shown in the figure, each prototype inherits methods and information from `mv-model-object-proto`, the general model object prototype that contains methods and information useful to all model objects. This prototype in turn inherits all of the methods and information in the `mv-data-object-proto` that were discussed in the previous section. Thus, the messages discussed in section 4 can be sent to model objects as well as to data objects.

5.1 Current Model and Model Object Names

One of the models is always the "current" model. The model's object identification is in the global variable `*current-model*`, to which messages can always be sent. The current model is indicated in the model menu by the checked menu item, and is shown in the workmap when a model icon is the high-lighted icon. The current model can be changed by clicking on a new model icon or choosing a new menu item in the model menu. Every model has a name which appears below the model's icon and in the model menu. The name can be used in the `setcm` function to change the current model from the keyboard. If there is a model named `pca-cars`, you can type `(setcm pca-cars)` to change the current model to `pca-cars`. You can also get information about a model by sending messages to any model. For example, you could type `(send pca-cars :title)`.

5.2 ViSta Model Object Messages and Methods

As noted above, all model objects have many methods inherited from `mv-data-object-proto` and from `mv-model-object-proto`. However, some methods that are needed by all model objects are not inherited from the ancestor objects. These must be uniquely defined for each individual model object so that their actions are appropriate to the particular model. These methods are called ViSta model object methods. These methods are:

1. `:options` — shows a dialog box to obtain values for the options of the analysis method. It places options values in slots that are unique to the model object.
2. `:analysis` — performs the analysis. It reads the information in the options slots and places results in analysis slots that are unique to the model object.
3. `:save-model-template` — used by the Save Model menu item.
4. `:create-data` — used by the Create Data menu item.
5. `:report-model` — used by the Report Model menu item.
6. `:visualize-model` — used by the Visualize Model menu item.

If you are planning on writing your own ViSta model object, you will have to write these methods yourself, as well as some additional methods and functions. We show an example of how this is done in section 7.

5.3 Steps Taken During Object Construction

When the user requests a data analysis (by using the Analysis menu, a toolbar button, a script, or by typing in the listener) ViSta takes a series of actions. These actions begin with the invocation of the model's constructor function.

The model's constructor function first checks on the validity of its argument values. If they are valid it then issues the `:new` message to the prototype model object. This message must be followed by the following set of arguments: The first several arguments are the parameters specific to the particular model (such as `covariances` for the principal components model). Then, there must be five arguments: 1) an integer which specifies the number of the model's method-button; 2) the object identification information for the data object being analyzed; 3) the model's title; 4) the model's name; and 5) a logical value indicating whether a dialog box is to be presented to obtain parameter values.

The constructor function's `:new` message invokes the prototype's `:isnew` method. The first few arguments of this method correspond to the arguments that were used in the constructor function's `:new` message which are unique to the model. Then (`&rest args` can be used for the last several arguments. This method creates a new model instance. Then, the values of the arguments that are unique to the model must be saved in the instance's slots. The method must then apply `call-next-method` to `args`. This calls `mv-model-object-proto's :isnew` method with the arguments that it needs. Then `mv-model-object-proto's :isnew` method takes the following actions:

1. The `mv-data-object-proto`'s `:isnew` method is called via `call-next-method` with arguments that are the model's data, variable list, title, observation labels, variable types and object name.
2. The model's analysis icon is added to the workmap when the `:copy-tool-icon` message is sent to the `*toolbox*`.
3. The `mv-data-object-proto`'s dialog slot is set to `t` or `nil`, depending on the value of `dialog` that was used in the constructor function. This determines whether a dialog box will be presented.
4. The model's `:options` method is used. If `dialog` is `nil`, it does nothing. If it is `true`, the `:options` method shows the dialog to get values for the options, which may or may not be different than those gotten from the constructor function. The `:options` method then puts these values, whatever they are, in the model object's slots, ignoring the values that are already there. The `:options` method returns `nil` when it is not used right, or when it is canceled, so that the analysis will be halted. Otherwise, the value returned by the `:options` method is ignored.
5. Now `mv-model-object-proto` determines whether to proceed with the analysis. It proceeds when the `:options` method returned a non-nil value.
6. The `mv-data-object-proto`'s `:data-object` method saves the object identification information about the data that are being analyzed in a slot that exists in `mv-model-object-proto`.
7. The `:analysis` method is invoked by `mv-model-object-proto`. The `:analysis` method uses whatever is in the model object's slots to carry out the analysis. Note that the `:analysis` method doesn't directly use the dialog results, rather, it uses information that has been put into slots by the dialog. More generally, none of the ViSta system methods use the results of any other method directly. All communication between methods is done via slots.
8. The `mv-data-object-proto`'s `:new-object` method updates the system to recognize the new model object. It adds a model icon to the workmap and a menu item to the model menu. The `*current-model*` and `*current-object*` global variables are updated to point to the new model. The variable and observation windows are cleared. A variable is created with the name of the new model and value of the model's object identification.

6 Model Plugins

Plugin model objects - called plugins - inherit from model objects, adding a few essential methods which make it possible for the developer to interfacing the new plugin with ViSta's core system, without needing to know anything about how the core system works. The result is that you just place your plugin's code in a particular part of ViSta's directory hierarchy and ViSta will load it in and interface it with the core system. ViSta's core engine will, in turn, take the actions necessary to place a new menu item in the analysis menu, a new button on the toolbar, new help in the help topics dialog, etc.

Writing a model plugin involves writing a very small amount of additional code beyond what is already required for writing a model object. The code involves defining seven variables and two different form of one function. Together the variables and function-forms interface your plugin with ViSta's core engine.

Before we illustrate plugins with an example, we discuss the function that needs to be written. The function, called the **plugin creator function**, has two forms: These forms are called the **plugin loader function**, which is used to load in your code, and the **plugin constructor function**, which is used to do analyses and construct model objects. The two forms have identical names and arguments, but different bodies.

The Rules: While the functions are simple, there are several rules that must be followed for the code to work right. We go over those rules next. It is recommended that while you read this you also look at the corresponding principal component code files, since they will be used to illustrate the plugin interface.

- The loader and constructor function names must not redefine any functions already defined. You must always check that the names you want to use have not be already used.
- The loader and constructor functions MUST have the same names. The names MUST be the same as the value of MENU-NAME, but with blanks in the MENU-NAME replaced by dashes in the function names.
- The loader and constructor functions MUST have the same arguments. All arguments must be keyword arguments. At least two keyword arguements are required. These two arguments are:

  DATA a symbol which, by default, is bound to *current-data*

  DIALOG a logical variable whose value must be T if the analysis dialog should be shown, NIL otherwise.

  The loader and the constructor function MAY have any number of additional keyword arguments as is needed by your analysis. The entire set of required and additional arguments must be the same for the loader and constructor functions.
- The loader function must comprise the entire executable contents of the file it is in, which is called the PLUGIN LOADER FILE. The loader file must be located in ViSta's PLUGINS directory. All files in the PLUGINS directory are automatically loaded in by ViSta, and all these files must be PLUGIN LOADER files. Note that it is mandatory that you prepare a separate PLUGIN LOADER file.

As mentioned above, the loader function loads in the code for your plugin. During the process of doing this, it also loads in the constructor function. Thus, the loader function modifies itself to become the constructor function. The architecture of the two function-forms is as it is so that code to do the analysis is not loaded until needed, thereby improving the efficiency of ViSta's startup process and, to a lesser degree, its overall behavior. Note, however, that the menu item, tool and model prefix are automatically loaded at startup.

7 An Example: The Principal Components Plugin Model Object

In this section we show the steps that were taken to write ViSta's Principal Components plugin model object prototype. This example can be followed by the developer who wishes to develop a new plugin, as the steps are the Isame for all objects. The code shown here can be found in `p0102.2-princomp.lsp` file in the plugins directory, and in the `pcaplug` subdirectory .

Step 1: Define your plugin's local environment. The following seven plugin system variables MUST be bound as described (see the example code in Figure 4). The first variable must be globally scoped, the rest must be local.

```
(setf *pca-menu-item-name* "Principal Components")
(let* ((plugin-subdirectory "pcaplug")
       (plugin-file "pcamain.lsp")
       (toolbar-button "PrnCmp")
       (workmap-icon "PCA")
       (data-types '("multivariate" "general"))
       (variable-types '("numeric"))
       )
  (send *vista* :prepare-plugin-environment
        plugin-subdirectory
       plugin-file
       *pca-menu-item-name*
       toolbar-button
       workmap-icon
       data-types
       variable-types))
```

FIGURE 4. Plugin Environment Code

1. **MENU-NAME** - A string specifying the plugin's menu item name (not more than about 20 characters). Both the name and the value of this variable are crucial, since they are used by ViSta's plugin system to identify, locate, load and run your plugin.

   The menu-name variable must be globally defined. It must not have the same name as any other globally defined variable. In particular, do not use MENU-NAME. Rather, make up a name which incorporates a part of the name of your plugin, as shown in the example.
2. **PLUGIN-SUBDIRECTORY** - A string of up to 8 characters specifying the name of the subdirectory containing the plugin code. The directory name must simultaneously follow the rules of directory naming for the MSDOS, MACINTOSH and UNIX operating systems. Use only lower-case letters, since the three operating systems treat upper-case letters differently. Also, do not use spaces, slashes, back-slashes, periods, commas or other special characters.
3. **PLUGIN-FILE** - A string of up to 8 characters specifying the name of the file that contains the PLUGIN CONSTRUCTOR function. You should include the .lsp extension. The filename must simultaneously follow the rules of directory naming for the MSDOS, MACINTOSH and UNIX operating systems. Use only lower-case letters, since the three operating systems treat upper-case letters differently. Also, do not use spaces, slashes, back-slashes, periods, commas or other special characters
4. **TOOLBAR-BUTTON** - Specifies the name of the plugin's button on the workmap's toolbar (a string of 5 or 6 characters - there is not room for more).
5. **WORKMAP-ICON** - Specifies the plugin icon's workmap name (a string of 2 or 3 characters, not more)
6. **VARIABLE-TYPES** - Specifies the variable types that are used by the plugin (a string list). The possible variable types are "numeric" and "category". "Ordinal" is not supported.
7. **DATA-TYPES** - Specifies the data types that are used by the plugin (a string list). The entire set of datatypes is "univariate" "bivariate" "multivariate" "category" "class" "freqclass" "crosstabs" "general" "matrix" and "missing". The message (send $:determine-data-type) determines and returns the datatype of $, the current data. More information about datatypes is available from ViSta's help topics. The show-datatypes item of the developer's menu shows all of the datatypes.

ViSta uses the values of PLUGIN-SUBDIRECTORY, PLUGIN-FILE and TOOLBAR-BUTTON to construct the global variables *NAME-plugin-constructor-file* and *NAME-plugin-path*, with NAME replaced with the value of TOOLBAR-BUTTON. In this example, ViSta creates the global variables *PRNCMP-PLUGIN-PATH* and *PRNCMP-PLUGIN-CONSTRUCTOR-FILE* which point to the directory and file in question.

Step 2: Define your plugin's loader function.

Recall the rules for writing loader functions that are presented in section 6: The loader function must have the same name and arguments as the constructor function (defined next). The loader function must be located in the PLUGIN LOADER FILE which is itself located in the PLUGINS directory. It's filename must follow the usual naming conventions. An example of a loader file is given in Figure 5

The plugin loader function MUST take the following four actions in the order specified.

1. define a documentation string. This should be identical to the constructor funtion's documentation string, except that each string should indicate which of the two functions is being documented.
2. display a copyright notice (optional). Please start it with a semi-colon and space, followed by CopyRt (with that capitalization), a colon, another space, and then the TOOLBAR-BUTTON name.
3. load the code containing the constructor function.
4. invoke the constructor function. Since the constructor and loader functions have the same name, the loader function appears to recursively invoke itself. However, since it has read in the constructor function, the constructor function is invoked rather than the loader function.

Step 3: .Define your plugin's constructor function: You must create a constructor function to construct an instance of the model plugin prototype. This is the function that is actually used to analyze the data. It is used by

```
(defun principal-components
    (&key
     (covariances nil)
     (data *current-data*)
     (dialog nil))
"ViSta plugin function to perform Principal Components analysis.  With no
arguments, calculates correlations among all active numeric variables in the
current data and performs a principal components analysis of those vari-
ables. Keyword arguments are:
:COVARIANCES followed by
   t    causes the analysis to be performed on covariances, or
   nil  causes the analysis to be performed on correlations, the default.
:DATA   followed by the data-object to be analyzed (default: *current-data*)
:DIALOG followed by t (to display parameters dialog box) or nil (default)
[loader function]"
    (format t "; CopyRt: PrnCmp Copyright (c) 1998-2002, by Forrest W.
Young~%> ")
    (load *prncmp-plugin-constructor-file*)
    (principal-components
     :covariances covariances
     :data   data
     :dialog dialog))
```

FIGURE 5. Plugin Loader Function

```
(defun principal-components
  (&key
     (data       *current-data*)
     (dialog      nil)
     (covariances nil))
"ViSta plugin function to perform Principal Components analysis.  With
no arguments, calculates correlations among all active numeric vari-
ables in the current data and performs a principal components analysis
of those variables. Keyword arguments are:
   :COVARIANCES followed by
      t    causes the analysis to be performed on covariances, or
      nil  causes the analysis to be performed on correlations, the default.
   :DATA   followed by the data-object to be analyzed (default: *current-
   data*)
   :DIALOG followed by t (to display parameters dialog box) or nil (default)
   [constructor function]"
  (send pca-model-object-proto :new *pca-menu-item-name* data dialog
        (not covariances))
   )
```

Figure 6: Constructor Function

the menu system, by the toolbar, by the data analyst via the keyboard and by script files. Recalling the rules for writing constructor functions, we prepare a constructor function for principal components which is simply:

The default argument values are such that if the analyst types the statement `(principal-components)` the analysis will be performed on correlations computed from the active numeric variables in the current data object. The dialog box will not be presented. On the other hand, the analysis menu generates the statement `(principal-components :dialog t)`, so that the dialog box is presented to see if the user wishes to perform the analysis on correlations or covariances.

The plugin constructor function does just one thing: It invokes the plugin's NEW method via the `send` statement at the end of the constructor function. The `send` statement should appear as it does here for other models, except for the presence of `(not covariances)`, which must be replaced by arguments corresponding to the keywords which are specific to the model. In particular the `send` statement must always have the following three arguments: The first argument must be your plugin's globally defined menu-item-name. The next two arguments must be DATA and DIALOG. The arguments that are unique to your plugin follow these three arguments.

The constructor function must be located in the plugin-constructor-file, which itself must be located in *plugin-subdirectory*, which is a subdirectory of the plugin directory.

STEP 4: Define the plugin's prototype object: Your plugin's defproto function must define your proto so that it inherits from `vista-analysis-plugin-object-proto`. The `defproto` statement for the Principal Components model object is:

```
(defproto pca-model-object-proto
  '(scores coefs eigenvalues svd corr corcovmat
          var-rel-contrib obs-rel-contrib)
  () vista-analysis-plugin-object-proto)
```

The slots `scores, coefs, eigenvalues, svd, corr, corcovmat, var-rel-contrib, and obs-rel-contrib` hold information specific to this model. The prototype inherits from `vista-analysis-plugin-object-proto`.

STEP 5: Define the :new message and :isnew method: The plugin constructor function issues the :NEW message. This message invokes the plugin's isnew method, thereby creating a new instance of the plugin object. This is done by the Lisp-Stat object system, there are no statements in the `:isnew` method that correspond to this action.

The :isnew method for principal components is shown in Figure . first thing that explicitly takes places in the

```
  (defmeth pca-model-object-proto :isnew (*pca-menu-item-name* data dialog corr)
    (cond
      ((> (send current-data :active-nvar '(numeric))
          (send current-data :active-nobs))
       (error-message (format nil "Note: Cannot analyze data ~
                   with fewer active observations (~d) ~
                   than active numeric variables (~d)."
                           (send current-data :active-nobs)
                           (send current-data :active-nvar '(numeric))))
       (send *toolbox* :reset-button tool-name))
      ((< (send current-data :active-nvar '(numeric)) 2)
       (error-message (format nil "Cannot analyze data ~
                   that has less than two active numeric variables."))
       (send *toolbox* :reset-button tool-name))
      (t
       (send self :corr corr)
       (call-next-method *pca-menu-item-name* data dialog))))
```

FIGURE 7. The :isnew method

`:isnew` method is that the data are checked to see if there are more active variables than active observations or less than two active numeric variables. If so, an error message is issued, the appropriate method button (number 6) is reset, and the analysis does not take place. Note that the `(error-message)` function displays the message in a dialog box. If the data pass the error checking, the value (`t` or `nil`) of the `corr` argument is stored in the `corr` slot. Then `call-next-method` is applied to each of the remaining arguments, calling the `isnew` method for `mv-model-object-proto`.

This general flow should be followed in any other model's `:isnew` method: First, check on the validity of argument values. If they are not valid, use `(error-message)` to report an error. Then reset the button. If they are valid, store the model abbreviation and argument values in their slots. Then, apply `call-next-method` to the remaining arguments.

Step 6: Define slot-accessor methods. There must be a slot accessor method for each slot specified in the defproto function. These methods are not shown here to save space.

Step 7: Define ViSta system methods. As indicated in section 5.2, each model prototype must have certain methods for ViSta to work properly. These methods must be named `options` and `analysis` (which are used by the `isnew` method of `mv-model-object-proto`), and `save-model-template`, `create-data`, `report-model` and `visualize-model` (all used by the menu system). We discuss these methods here, and indicate how similar methods would be defined for other model objects.

Options: The `options` method for the principal components prototype is given below.

```
(defmeth pca-model-object-proto :options ()
  (when (send self :dialog)
        (let ((result nil)
              (dialog-value (choose-item-dialog
                "Analysis Options:"
              '("Analyze Covariances"
                "Analyze Correlations") :initial 1)))
          (when dialog-value
                (when (= 1 dialog-value) (setf result t))
                (send self :corr result))
          dialog-value)))
```

This method first checks the value of the `dialog` slot to see if a dialog is to be presented. This action should always be taken first since when a script file job is in progress, or when a model is being loaded, the dialog should not be presented. The model's constructor function should set the value of dialog to nil by default, as it does in this example. When appropriate, the dialog is then presented to see if the analysis is to be performed on correlations or covariances. The dialog sets the value of the `corr` slot for later use by the `:analysis` method. Note that the results of the dialog are communicated to the analysis method through the slot. This should always be this way for all model prototypes. Then, the value that is returned by `choose-item-dialog` (which is `nil` when the dialog is canceled) is used as the value returned by `:options`, so that the analysis can be canceled when the dialog has been canceled. The `:options` method should always return `nil` in this situation for all models. Indeed, it should return `nil` whenever the options dialog is not properly used. For an example of a more complex dialog, look at the code for the multivariate multiple regression options dialog.

Analysis: The `analysis` method for the principal components proto is:

```
(defmeth pca-model-object-proto :analysis ()
  (let* ((left-alpha 1)
         (data (send self :data-matrix))
         (nobs (select (array-dimensions data) 0))
         (prepped-data
           (if (send self :corr)
             (/ (normalize (center data) 1)
                (sqrt (1- nobs)))
             (/ (center data) (sqrt (1- nobs)))))
         (svd (sv-decomp2 prepped-data))
         (svd (if (< (sum (col (select svd 2) 0)) 0)
                   (list
                    (* -1 (select svd 0))
                    (select svd 1)
                    (* -1 (select svd 2))
                    (select svd 3))
                 svd))
```

```
          (scores (matmult
            (select  svd 0)
            (diagonal (^ (select svd 1) left-alpha))))
          (eigenvalues (^ (select svd 1) 2))
          (coefs (matmult
            (select  svd 2)
            (diagonal (^ (select svd 1)(1- left-alpha)))))
          )
    (send self :svd svd)
    (send self :coefs coefs)
    (send self :scores scores)
    (send self :eigenvalues eigenvalues)
    t))
```

The details of this method are not important for this example, other than to note that all of the analysis results (i.e., `svd`, `coefs`, `scores` and `eigenvalues`) are computed inside a `let*` statement (so that they are only locally defined) and then saved for later use by placing them in appropriate slots. The method communicates its results indirectly through slots, not directly by returning them. The method returns t.

Save-Model-Template: Every model object needs to have a method named `save-model-template`, a method which is used by the model menu's `save-model` method. The method contains a template of the code that creates the model object. The method must always have as an argument the object identification information for the data object used by the model, since the data must be saved along with the model. The `mv-model-object-proto`'s `:save-model` method takes information placed into the template (as explained below) and places it in a file. When the file is loaded back into ViSta by `(load-model)`, a principal components analysis is performed on the data that were also saved in the file.

For principal components, the method is:

```
(defmeth pca-model-object-proto :save-model-template
                     (data-object)
  `(principal-components
    :title      ,(send self :title)
    :name       ,(send self :name)
    :dialog      nil
    :covariances ,(not (send self :corr))
    :data (data  ,(send data-object :name)
                 :title     ,(send data-object :title)
                :variables ',(send self :variables)
                :types     ',(send self :types)
                :labels    ',(send self :labels)
                :data      ',(send self :data))))
```

Note the unusual backquote syntax (which is explained briefly by Tierney on pp. 98 and 120, as well as on page 197 in the discussion of saving objects). To be clear, the character in front of `` `(principal-components `` is a backquote. This character in front of a list causes all elements of the list to be quoted, except those preceded by commas, which are treated in the normal fashion. Note that the data for the principal components function come from the model object via the various `(send self` functions. However, the data's name and title must come from the original data object that was analyzed.

The `mv-model-object-proto`'s `:save-model` method places the `principal-components` function, with the functions following commas being replaced with their results, in a file. When the file is loaded back into ViSta by `(load-model)`, a principal components analysis is performed on the data that were also saved in the file.

Create-Data: For principal components, just as for any other model object, the method which creates output data objects must work in such a way that it can be used by the menus, can be typed at the keyboard, or can be contained in a script file. The method is shown below.

There are several things to point out about this method. First, note that the method has optional key-word arguments which determine whether the dialog box is displayed, and which determine which output data objects will be created when the dialog box is not displayed. These optional arguments permit the menu system to generate:
`(send *current-model* :create-data :dialog t)`.
This causes the dialog box to be shown. On the other hand, these arguments permit a script to contain the statement:
`(send *current-model* :create-data :scores t :coefs nil)`
which creates one output data object of scores without the user having to intervene by responding to a dialog box. For the system to work in these ways, any `create-data` method for another model object must have a similar construction.

Also, note the statement, near the beginning of the function:
`(if (not (eq current-object self)) (setcm self))`.
This statement must be used at the beginning of the method to assure that the current model is properly selected. Finally, `(not (not desires))` causes the method to return `nil` when the dialog is canceled, and `t` when it is not canceled.

```
(defmeth pca-model-object-proto :create-data
  (&key (dialog nil) (scores t) (coefs  t) (input  nil))
  (if (not (eq current-object self)) (setcm self))
  (let ((creator (send *desktop* :selected-icon))
        (desires (list (list
             (if scores 0)(if coefs 1)(if input 2)))))
    (if dialog
        (setf desires
           (choose-subset-dialog
              "Choose Desired Data Objects"
              '("Component Scores"
                "Component Coefficients"
                "Analyzed Input Data")
              :initial (select desires 0))))
    (when desires
         (when (member '0 (select desires 0))
               (send *current-model*
                     :pca-scores-data-object creator))
         (when (member '1 (select desires 0))
               (send *current-model*
                     :pca-coefs-data-object  creator))
         (when (member '2 (select desires 0))
               (send *current-model*
                     :create-input-data-object
                          "PCA" creator)))
    (not (not desires))))
```

This method calls three other methods which actually create the data objects. One of these (`:create-input-data-object`) is already defined by `mv-model-object-proto`. It creates a copy of the input data, taking into consideration which variables are selected and active. The other two methods are specific to the principal components model. One of these is shown here (they are nearly identical):

```
(defmeth pca-model-object-proto)
          :pca-scores-data-object (creator)
  (data (strcat "Scores-" (send self :name))
    :created creator
    :title (strcat "PCA Scores for " (send self :title))
    :data (combine (send self :scores))
    :variables
```

```
         (mapcar #'(lambda (x) (format nil "PC~a" x))
         (iseq (min (send self :nvar) (send self :nobs))))
    :labels (send self :labels)
    :types (repeat "Numeric"
         (min (send self :nvar) (send self :nobs)))))
```
Note that the `pca-scores-data-object` method consists entirely of a data function to create a new data object (this function was discussed earlier in this chapter). The arguments of the function were described earlier in the chapter. except for `:creator`. This argument specifies which workmap icon object represents the object which is creating the data object, so it's argument must be the object identification of the appropriate icon. This information is used to construct the workmap.

Various functions are used to create the values for the arguments of the function. The name is created by concatenating a meaningful prefix (in this case "Scores-") with the model object's name. The title is created by concatenating a similar prefix with the model object's title. The data consist of a list of scores, the combine function being used to convert the scores matrix to a list. The variable names are a combination of the string "PC" and a sequence number. The labels and types are self-explanatory. If this method is copied for new model objects, then these new model objects will produce data objects that conform with other ViSta data objects.

Visualize-Model: We do not present the `visualize-model` method here because it is too long. The important aspect of it for those constructing the method for other models is that it has no arguments, and is used by the menu-system and from the keyboard by the statement

```
   (send *current-model* :visualize-model).
```

Report-model: The `report-model` method for the principal components prototype is shown below. This function is used by the menu-system, as well as by the user from either the keyboard or from script files, by the statement:

```
   (send *current-model* :report-model).
```

The method uses three special functions which ensure that the report works correctly under the various operating systems, and that it's appearance is consistent between models. These functions are `report-header`, `display-string`, and `print-matrix-to-window`. In constructing other reporting methods, you should use these functions in place of standard printing functions since they write to the Macintosh text window or to the MS-Windows or X-Windows listener in a consistent fashion. The statement

```
   (report-header (send self :title))
```

determines how and where the report should be printed, which varies from one operating system to another. It displays the title and returns a value which identifies the place that the information should be written (a text window or the listener). In the example, the variable w gets this value. The `display-string` and `print-matrix-to-window` functions are used to report the desired information in the desired way. The `display-string` function takes a string as its first argument, and `w`, the window identification, as its second argument. The `print-matrix-to-window` function has two required arguments: The first is the matrix which is to be printed, the second is `w`, the place it is to be printed. This function also has two optional arguments. One is `:labels`, which must be followed by a list of strings. These are used to label the rows of the matrix. The other is `:decimals`, which must be followed by an integer. The integer determines the number of decimals that are printed following the decimal point.

Finally, note that the method contains the statement

```
   (if (not (eq current-object self)) (setcm self))
```

to ensure that the proper object is reported.

```
(defmeth pca-model-object-proto :report
                         (&key (dialog nil))
  (if (not (eq *current-object* self)) (setcm self))
  (let* ((w (report-header (send self :title)))
         (labels (send self :labels))
         (vars   (send self :variables))
         (scores (send self :scores))
```

```
     (coefs  (transpose (send self :coefs)))
     (eigenvalues (send self :eigenvalues))
     (proportions (/ eigenvalues (sum eigenvalues)))
     (fitmat (transpose
         (matrix (list 3 (min (send self :nobs)
                              (send self :nvar)))
                 (combine eigenvalues proportions
                          (cumsum proportions)))))
     (lc-names
        (mapcar #'(lambda (x) (format nil "PC~a" x))
              (iseq (send current-model :nvar)))))
(display-string
 (format nil "Principal Components Analysis of") w)
(if (send self :corr)
  (display-string (format nil " Correlations~2%") w)
  (display-string (format nil " Covariances~2%") w))
(display-string
  (format nil "Model: ~a~2%" (send self :name )) w)
(display-string
  (format nil "Variable Names: ~a~2%" vars) w)
(display-string
  (format nil "Fit Indices for each Component:~
           ~2% Eigenvalue  Proportion  ~
               Cum Propor Component~%") w)
(print-matrix-to-window fitmat w
            :labels lc-names :decimals 5)
(display-string
  (format nil "~%Coefficients (EigenVectors):~%") w)
(print-matrix-to-window coefs w
           :labels lc-names :decimals 3)
(display-string
  (format nil "~%Component Scores:~%") w)
(print-matrix-to-window scores w
           :labels labels :decimals 3)
w))
```

8 References

Tierney, L. (1990) *Lisp-Stat: An Object-Oriented Environment for Statistical Computing & Dynamic Graphics*. Addison-Wesley, Reading, Massachusetts.

Young, F.W. (1996) *Vista:The Visual Statistics System.* Research Memorandum 94-1(b). L.L. Thurstone Psychometric Laboratory, Univ. N. Carolina.