

An OO Approach to Implementing Drag-and-Drop Functionality in FRAME Applications

Mark Olson, Trilogy Consulting Corporation, Kalamazoo, Michigan

INTRODUCTION

Almost every GUI involves the operation of moving, copying or linking one object to another. One of the most natural ways of executing such a transfer operation is via *drag-and-drop*. One of the many new features added by SAS® in its 6.11 release is the ability to implement drag-and-drop functionality into FRAME applications.

This paper presents an enhanced way of programming drag-and-drop in SAS/AF®. By developing a library of drag-and-drop classes and implementing an object that controls the interaction between the drag site and the drop site, FRAME developers can add the powerful technique of drag-and-drop to their applications with ease.

After reviewing the requirements of the base class drag-and-drop library, the implementation of a non-visual object called a "representation processor" will be introduced. This representation processor is the key to making the implementation of drag-and-drop simple. It not only controls the interactions between the drag site and the drop site, but also aids in reducing the amount of code and re-testing needed when new drag and drop sites are added to your application.

DRAG-AND-DROP FUNCTIONALITY

Within an application any object that can be selected and dragged across the screen with a mouse is considered a *drag site*. Likewise, any object that can receive the contents of a drag site is considered a *drop site*. Within the SAS system, a FRAME developer has the ability of defining drag sites and drop sites from any subclass of the Widget or Frame classes, aside for the slider and scroll bar classes which do not support drag and drop. Objects can be defined as both drag sites and drop sites. The drag site and drop site of a drag and drop transaction do not necessarily need to be in the same frame.

Drag Sites

There are four tasks that must be done in order to fully define a widget as a drag site:

- 1) Define at least one data representation for the drag site using the `_ADD_DRAG_REP_` or `_SET_DRAG_REP_` methods. A discussion of these data representations will

follow below.

- 2) Define at least one drag operation for the drag site using the `_ADD_DRAG_OP_` method. Choices being copy, move, link. Copy is the default operation.

- 3) Supply the drag site data to the drop site via the `_GET_DRAG_DATA_` method. A detailed discussion of this method will follow below.

- 4) Enable the site as a drag site via the `_ENABLE_DRAG_DROP_SITE_`. By default a drag site is enabled.

At the heart of a drag and drop transaction is the transfer of data from one object to another. The drag site is the owner of this data and it is the drag site's ultimate goal to deliver this data to the drop site. This goal is achieved within the `_GET_DRAG_DATA_` method. It is here that the drag site massages the data into the format which the drop site expects.

Drop Sites

There are four tasks that must be done in order to fully define a widget as a drop site:

- 1) Define at least one data representation for the drop site using the `_ADD_DROP_REP_` or `_SET_DROP_REP_` methods. A discussion of these data representations will follow below.

- 2) Define at least one drop operation for the drop site using the `_ADD_DROP_OP_` method. Choices being copy, move, link. Copy is the default operation.

- 3) Define what the drop site is to do with the data supplied by the drag site. This is done in the `_DROP_` method.

- 4) Enable the site as a drop site via the `_ENABLE_DRAG_DROP_SITE_`. By default a drop site is enabled.

The ultimate goal of the drop site is to perform some process with the data supplied by the drag site. The data is extracted from a data list that was prepared by the drag site in the `_GET_DRAG_DATA_` method and then processed in the `_DROP_` method of the drop site. The format of the data on the data list is defined by what is called a data representation.

DATA REPRESENTATIONS

Both the drag site and the drop site have a collection of what are called data representations. As stated above, you add representations to these collections via the `_ADD_DRAG_REP_` and `_ADD_DROP_REP_` methods. These data representations can represent something as simple as a string or as complicated as a SCL list. Data representations can be thought of as contracts between the drag site and the drop site.

Example 1

Let's say that we have a listbox (the drag site) containing a list of dataset names and a graphics widget (the drop site) on a FRAME. The idea is that the user will drag a name off of the listbox onto the graphics widget to display the dataset in some graphical manner. Both the listbox and the graphics widget will have a data representation named "REP1". The data representation "REP1" will define that the name of the dataset will be put onto the data list (which will be discussed below in detail) as a named item with the name "data". It is the drag site's responsibility for putting the name of a dataset onto a data list as a named item, named "data" in the `_GET_DRAG_DATA_` method. In return, the drop site will know to pull the named item "data" off of the data list to retrieve the name of the dataset that is to be graphed.

When a drag and drop transaction begins, the system looks for a matching representation in the drag site representations collection and the drop site representations collection, taking the first match. If there is a tie then the first match in the drop site collection is chosen. If no match is found then the drop site that the mouse is over, is not a valid drop site for the drag site selected. Once a match is found, seven drag and drop methods are set into motion automatically.

DRAG-AND-DROP METHODS THAT RUN AUTOMATICALLY

These seven methods are used to do a variety of tasks, from changing the cursor to a different shape while over the drop site to executing the main functionality desired on the drop action. They are set into motion once the mouse button is pushed down on the drag site until the mouse button is released on the drop site. It is important to remember that you do not have to explicitly invoke these methods by issuing a call send. The system automatically fires these methods off at the appropriate time. Because of this you can not see most of these methods execute in the debugger. Thus, they can be hard to debug if they are large and complicated. The following is a list of the seven methods

and a short description of each. For detailed information on these methods, including their syntax and parameters see the widget and/or Frame section of the FRAME Class Dictionary (SAS Institute 1995) or Appendix 3 of FRAME Application Development Concepts (SAS Institute 1995).

`_START_DRAG_` (drag site)

This method is where it all starts. When the user first pushes the mouse button down onto the drag site this method is run on the drag site. Again, it is fired off automatically by the system. You can use this method to change the appearance of the drag site, indicating to the user that the drag and drop transaction has begun. Or you can implement any other preprocessing that might need to be done.

`_RESPOND_TO_DRAG_ONTO_` (drop site)

This method is run on the drop site whenever the cursor is dragged over the drop site. This could happen many times within one drag and drop transaction. This method can be used to change the appearance of the drop site or the cursor to indicate to the user whether or not the drop site is a valid one for the drag in progress. Common functionality, which happens automatically within the SAS System, would be to change the cursor to: a circle with a diagonal line through it to indicate the drop site is not valid, some icon to indicate that the drop site is valid and that a move transaction of data will occur on the drop or some icon with a plus sign on it to indicate that the drop site is valid and that a copy transaction of data will occur on the drop. For a good example of this functionality do some drag and drop within the windows file manager.

`_RESPOND_TO_DRAG_OFF_` (drop site)

This method is run on the drop site whenever the cursor is dragged off of the drop site. Again, this method could be run many times within a drag and drop transaction. This method is primarily used to restore any appearance changes that would made in the `_RESPOND_TO_DRAG_ONTO_` method.

`_GET_DRAG_DATA_` (drag site)

This is one of the key methods of the drag and drop process and will only run once within a drag and drop transaction. It is run on the drag site when the drop event actually occurs, that is, when the user releases the mouse over the drop site. This method is where the drag site prepares the data to be passed to the drop site in the format defined by the data representation in question.

`_VALIDATE_DROP_DATA_ ( drop site)`

This method is run after the `_GET_DRAG_DATA_` method and before the `_DROP_` method. It gives the developer an opportunity to programmatically validate the drag data. If the data is invalid for any reason the developer can set the "RC" parameter, the seventh of the method to "C" and cancel the drag and drop transaction.

`_COMPLETE_DRAG_ (drag site)`

This method is run on the drag site. It gives the drag site a final chance to clean itself up.

`_DROP_ (drop site)`

This method is run on the drop site and is the last method to run in the drag and drop sequence. This is where the developer will define what it is that the drop site is to do with the drag data, whether it be to populate a data table or a graphics widget, or to fire off another application.

THE `_GET_DRAG_DATA_` AND `_DROP_` METHODS RELATIONSHIP

As we said earlier, it is the goal of the drag site to deliver its data to the drop site. This is done within the `_GET_DRAG_DATA_` method. Within this method the drag site will prepare the data in the format defined by the data representation being used for the drag-and-drop transaction in question. The data list is created by the system. The programmer adds data to the list in the `_GET_DRAG_DATA_` method. The drag site will use the data list as a parameter of this method as the vehicle for the delivery of the data to the drop site. This data list is global and should be used cautiously. In particular, local lists must be copied to a global list before they are added to the data list. Within the `_DROP_` method of the drop site the data can easily be extracted from the data list (which is also a parameter of this method) because the data representation will tell the `_DROP_` method how the data was formatted by the drag site.

THE SAS SYSTEM IMPLEMENTATION OF DRAG-n-DROP

Example 2

We wish to have a frame with: a listbox named "datasets" containing a list of dataset names, graphics widget named "statsgph" for displaying statistics about the data and an

icon named "appl" representing an application such as SAS INSIGHT® that will use the data to perform further analysis. The listbox in our example will be a drag site and the graphics widget and icon will be drop sites. The user will drag a dataset name from the listbox onto the graphics widget to display statistics about that data. Likewise, the user will drag a dataset name onto the icon to launch that application to do further analysis.

The Representations

We will need two data representations, say "LST_GRP" for the representation between the listbox and the graphics widget and "LST_ICON" for the representation between the listbox and the icon. The listbox will have both of the representations "LST_GRP" and "LST_ICON" in its representation collection. The icon will just have "LST_ICON" in its collection and the graphics widget will just have "LST_GRP" in its collection. We can define these representations in the INIT section of the frame by using the `_ADD_DRAG_REP_` and `_ADD_DROP_REP_` methods.

```
INIT:
call notify('datasets', '_ADD_DRAG_REP_',
            'LST_GRP');
call notify ('datasets', '_ADD_DRAG_REP_',
            'LST_ICON');
call notify ('statsgph', '_ADD_DROP_REP_',
            'LST_GRP');
call notify ('appl', '_ADD_DROP_REP_',
            'LST_ICON');
```

Return;

`_GET_DRAG_DATA_ and _DROP_`

We will concentrate on the `_GET_DRAG_DATA_` and `_DROP_` methods since they are the primary methods in the drag and drop process. First we need to override these methods. There are two ways to do this: one is to use the `_SET_INSTANCE_METHOD_` method and the other is to subclass the widgets. Let's assume for this example that the widgets are subclassed. Now we have full reign over the functionality of these methods.

For now the `_DROP_` methods of the icon widget and the graphics widget are straightforward because they only have one data representation to deal with. Thus, we can just put our code in for that representation.

For the listbox widget we have another story. It is the drag site's responsibility to prepare the data in the proper format, depending on the data representation, for the drop site. What makes it interesting in our example is the fact that this

drag site will be dragging to two different drop sites with different data representations. Thus, the `_GET_DRAG_DATA_` method must contain logic that will determine how the data is to be formatted based on the data representation in question.

```
GETDD: /* _get_drag_data_ */
      method
      rep
      op $40
      data
      x
      y 8
;
  If rep = "LST_GRP_H" then do;
    /* processing for drop on graphics widget */
  end;
  else if rep = "LST_ICON" then do;
    /* processing for drop on icon widget */
  end;
endmethod;
```

Miscellaneous

In our example we will use the default drag operation for the drag site which is "copy". Also, for both of our drop sites we will use the default drop operation which is "copy". All of our drag-and-drop widgets are enabled by default so we have now defined every thing necessary to perform a drag and drop transaction between the listbox and either the icon or the graphics widget.

The Process

First the user presses the mouse button down onto the listbox which is our drag site. This action triggers off the `_START_DRAG_` method of the listbox. The SAS system automatically changes the cursor to a default shape at this point. We could have put code in this method to change the cursor to something other than the default shape, or something more elaborate to indicate to the user that the drag and drop process has begun. The user now drags the cursor over to the graphics widget. When the cursor crosses onto the graphics widget the `_RESPOND_TO_DRAG_ONTO_` method of the graphics widget (the drop site) is triggered. The SAS system automatically changes the cursor to indicate whether or not the drop site is valid. Again, we could have put code in this method to do more than just the default actions. Now the user releases the mouse button with the cursor over the graphics widget. This triggers the `_GET_DRAG_DATA_` method of the listbox. The system determined that the data representation "LST_GRP_H", is the representation to be used. Thus, the code in the `_GET_DRAG_DATA_` method for this representation is run. This code will put the data onto the data list in the proper format and the

`_VALIDATE_DROP_DATA_` method of the drop site will run next. In this method we could have verified that the data was properly formatted for this representation and canceled the drag and drop process if it was not. Next, the `_COMPLETE_DRAG_` method of the listbox triggered. In this method we could return the cursor to its original shape if we had manually altered the shape. Now the `_DROP_` method of the graphics widget is triggered. In this method we retrieve the data from the data list and process it accordingly.

The Downfalls

There are two primary downfalls with the SAS implementation of drag and drop functionality. First, every time you want to add drag and drop you must either subclass the widgets that you want to use or use the `_SET_INSTANCE_METHOD_` method on those widgets. Using the `_SET_INSTANCE_METHOD_` method can become complicated if you are implementing more than a few drag and drop widgets and you are overriding more than one or two methods. You also lose the ability to inherit these methods from other subclasses of these drag-and-drop widgets.

The main downfall of the implementation is the dependency of the logic within some of the drag and drop methods on the data representation. The `_GET_DRAG_DATA_`, `_VALIDATE_DROP_DATA_`, `_COMPLETE_DRAG_` and `_DROP_` methods are all dependent on the data representation parameter.

What are the consequences of this dependency? First, these methods must contain an if-else if-else structure to execute different code for the different data representations. This means that every time a new representation is added to a drag and drop widget, you must modify the method code for that widget. Once you modify the code you need to re-test it. One of the main concepts of object oriented programming is polymorphism. If the logic for the different data representations was polymorphic, adding a new representation would not require re-testing of existing code.

Second, all of the logic for each of the representations is in each one of these methods. This can lead to complex, hard to manage methods. Again, polymorphic classes for each representation would create smaller pieces of representation specific code, making the implementation of drag-and-drop easier to manage.

A DIFFERENT APPROACH

To improve on the implementation above we must first make

it easier to access the drag-and-drop methods of a widget, and most importantly, we must sever the dependency of the drag-and-drop methods on the data representation. The first problem has an easy solution: Create a library of drag-and-drop widgets. The solution to the second problem involves creating a non-visual object that I call a *representation object*. This object encapsulates the logic for each data representation between a drag site and a drop site. Put the two solutions together and you get a simple way of implementing drag and drop functionality into your Frames applications.

The Drag-and-Drop Widget Library

The library should consist of widgets most likely to be used to perform the drag-and-drop. Widgets fitting this description would include: listboxes, the graphics widget, SAS/Graph output widget, image icons. This is by no means an exhaustive list, the library can be added to as needed. To add a widget to the library you need to subclass the widget down from the SAS-supplied widgets. Then the four drag and drop methods that are dependent on the data representation (`_GET_DRAG_DATA_`, `_VALIDATE_DROP_DATA_`, `_COMPLETE_DRAG_`, `_DROP_`) will be defined as below in order to delegate the responsibility of the method to a *representation object*.

```
GETDD: /* _get_drag_data */
method
  rep
  op $40
  data
  x
  y 8
;
  rep_pro_c = loadclass ('dragdrop.reppros.'||rep);
  call send (rep_pro_c, '_new_', rep_pro_o);
  call send (rep_pro_o, '_get_drag_data_',
    _self_, op, data, x, y);
  call send (rep_pro_o, '_term_');
endmethod;
```

```
VALDD: /* _validate_drop_data */
method
  rep
  op $40
  data 8
  task $7
  x
  y 8
  rc $1
;
  rep_pro_c = loadclass ('dragdrop.reppros.'||rep);
  call send (rep_pro_c, '_NEW_', rep_pro_o);
  call send (rep_pro_o, '_validate_drop_data_',
    _self_, op, data, task, x, y, rc);
```

```
call send (rep_pro_o, '_term_');
endmethod;
```

```
COMPLETE: /* _complete_drag */
method
  rep
  op $40
  data 8
  code $6
;
  rep_pro_c = loadclass ('dragdrop.reppros.'||rep);
  call send (rep_pro_c, '_NEW_', rep_pro_o);
  call send (rep_pro_o, '_complete_drag_',
    _self_, op, data, code);
  call send (rep_pro_o, '_term_');
endmethod;
```

```
DROP: /* _drop */
method
  rep
  op $40
  data 8
  task $7
  x
  y 8
;
  rep_pro_c = loadclass ('dragdrop.reppros.'||rep);
  call send (rep_pro_c, '_NEW_', rep_pro_o);
  call send (rep_pro_o, '_drop_',
    _self_, op, data, task, x, y);
  call send (rep_pro_o, '_term_');
endmethod;
```

Note that the above methods are defined the same across all drag-and-drop widgets. Thus, instead of expliciting defining these methods for each drag-and-drop widget in your library, you could define the methods in one SCL entry for all of the drag-and-drop widgets and point to this SCL entry within the method definition window. Therefore, the code would exist in one and only one place, making enhancements easier to implement.

The first line of code in these methods loads the appropriate representation class. This is dynamically driven by the "rep" parameter which is equal to the data representation being used. The second line creates an instance of the representation class. The third delegates the responsibilities of the method off to the representation object. The last line destroys the representation object in order to eliminate memory leaks. We could eliminate the last line of code if we made the representation class a singleton class. For more information on the implementation of a singleton class see Kevin Tyler Brown. (1995). "The Singleton Class in Screen Control Language", *Proceedings of the Twentieth Annual SAS® Users Group International Conference*. Cary, NC: SAS Institute.

Inc.

A detailed discussion of the *representation class* will follow. After the latter is complete, adding drag and drop functionality does not involve any modification to the drag and drop widgets within the library. The developer will merely drop widgets from this library onto their frames and define a *representation class* to perform the work for the drag-and-drop widgets.

Representation Classes

A representation class is a non-visual class that will contain all four drag-and-drop methods that are dependent on the data representation. There will exist one representation object for each data representation between a drag site and a drop site. The name of the representation class will be exactly the name of the data representation in which it is intended to be used for. This is extremely important because this is the mechanism we use to dynamically delegate methods to different representation objects depending on the data representation. This is how we overcome the downfall of having if-else if-else structures within the drag-and-drop methods that are dependent on the data representation. This structure is replaced by having the logic for each data representation inside a representation object and delegating to this object from the drag site and drop site methods.

Example 3

Let's assume that we have a listbox, an extended text entry field and a video player widget in our drag-and-drop widget library. Now we build a new frame and drop one of each of these widgets onto the frame. The idea is that the user can drag the name of a video from the listbox (named *vidlist*) onto the video played (named *vidplay*) to play the video or the user can drag the name of a video onto the extended text entry field (named *descript*) to display a description of the video. We need to define two data representations, one between the listbox and the video play, let's call it "LST_VID" and one between the listbox and the extended text entry field, let's call it "LST_TEXT". "LST_VID" defines that the drag site will supply the full path to a video file and the drop site will play that file when it is dropped. "LST_TEXT" defines that the drag site will supply the description of the video being dragged and the drop site will display that description.

NOTE: The video names displayed within the listbox will represent non-visual video objects which have instance variables containing the full path to the video file and a description of the video. The objects will also have methods defined to set and retrieve instance variables. I will not go into any detail about these video objects so please assume that they are defined.

Now we need to create one representation class for each of the data representations defined. Recall that the names of these classes must correspond to the names of the data representations. So we create two new classes named "LST_VID" and "LST_TEXT".

No if-else if-else structures are involved in any of the methods for the representation objects. Each representation object encapsulates the workings of one data representation. Note also that we did nothing to the actual drag-and-drop widgets except drop them onto the frame as we would any other widget.

In the INIT section of the frame we will set up the widgets for drag and drop by adding their drag representations and drop representations.

INIT:

```
call notify ('vidlist', '_ADD_DRAG_REP_',
             'LST_VID');
call notify ('vidlist', '_ADD_DRAG_REP_',
             'LST_TEXT');
call notify ('vidplay', '_ADD_DROP_REP_',
             'LST_VID');
call notify ('descript', '_ADD_DROP_REP_',
             'LST_TEXT');
```

•
•

Return;

When the user begins the drag and drop process the seven drag and drop methods go into action. Let's begin by dragging a video name from the listbox to the extended text entry field to view the description of the video. We will concentrate on the four methods that are dependent on the data representation and in particular, the `_GET_DRAG_DATA_` method of the drag site and the `_DROP_` method of the drop site. The first one of these four methods to execute is the `_GET_DRAG_DATA_` method of the listbox (the drag site). The system has already decided that the data representation for this drag and drop operation is "LST_TEXT". So the "rep" parameter in the `_GET_DRAG_DATA_` method is equal to "LST_TEXT". Thus, inside the method shown above, the representation object class "LST_TEXT" is loaded and an instance of this class is created. The drag site now delegates its `_GET_DRAG_DATA_` method off to the representation object by calling this representation object's `_GET_DRAG_DATA_` method. Within this method shown above, we get the description of the video and put it onto the data list as a "named" item named "video_description". Note that the representation object is terminated as the last step in the `_GET_DRAG_DATA_` method. This is to eliminate memory leaks. Now when the `_DROP_` method of the drop site (in this case, the extended text entry field) is

executed it loads the "LST_TEXT" representation class and an instance of this class is created. The drop site now delegates its _DROP_ method to the representation object by calling its _DROP_ method. This method pulls the video description off of the data list and displays the description within the extended text entry field.

Now the user drags a video name from the listbox to the video player. The system decides that the data representation is "LST_VID". Now the same process as above occurs except that within the _GET_DRAG_DATA_ method of the drag site and the _DROP_ method of the drop site (in this case, the video player) the "LST_VID" representation object class will be loaded and delegated to instead of the "LST_TEXT" representation object class. Again, this is the key to overcoming the downfall of having if-else if-else logic within the methods of the drag-and-drop objects and to having the implementations of the different data representations defined polymorphically.

The real benefit of this implementation comes through if we were to add another data representation to your system now. Say we added an icon that represented a application used for editing a video file. Let's call the data representation between the listbox (drag site) and the icon (drop site) "LST_ICON". In the standard implementation the developer would have to go into both the drag site and drop site widget methods and modify them to handle this new data representation. This means all of these methods must be re-tested. With the above implementation, the developer would just create a new representation object class named "LST_ICON" and define the necessary logic within its methods. The drag and drop widget methods themselves would never need to be touched!

CONCLUSION

This implementation takes some up front work to get the drag and drop widget library into place. However, once you have this library set up, adding drag-and-drop widgets to your SAS/AF Frames applications is no different than adding any other widget. You just drop them onto the frame! Using the representation objects to do the dirty work for the drag and drop widgets helps keep things simple by encapsulating the workings of the different data representations. This also eliminates the need for re-testing when a new data representation is added to the system. The representation objects also keep the chunks of method code small and easy to manage and eliminate the if-else if-else logic otherwise needed.

This implementation has plenty of room for extension and simplification. For example, the representation objects can be inherited from other representation objects in some cases,

creating a hierarchy of representation objects. This would promote code reuse.

Creating a drag and drop widget library and introducing the concept of the representation object into your Frames development process will greatly reduce the effort needed to implement drag and drop functionality in your Frames applications.

REFERENCES

SAS Institute, Inc.(1995), SAS/AF Software: FRAME Application Development Concepts, Version 6 First Edition, Cary, NC: SAS Institute Inc.

SAS Institute, Inc.(1995), SAS/AF Software: FRAME Class Dictionary, Version 6 First Edition, Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

Thanks to Andy Norton, Cheryl Olson, Sally Goostrey and Lorie Livingston for reviewing this paper.

NOTES

SAS, SAS/AF, and SAS/Insight are registered trademarks of SAS Institute, Inc. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

AUTHOR CONTACT

Mark Olson
Object Technology Group
Trilogy Consulting Corporation
5278 Lovers Lane
Kalamazoo, MI 49002

(616) 344-4100