COOL

Cool Design Specification *(Draft)*

Arthur A. van Hoff
The Turing Institute

June 1991

Summary

Cool is an interactive UI development environment based on TNT and HyperNeWS. It is used to interactively design and test OpenLook applications in the TNT environment. It will implement most of the UI design features as proposed in the HyperNeWS 2.0 design documents.

The aim of this document is to establish a TNT platform for Cool which implements most of the proposed HyperNeWS 2.0 features. The TNT API remains the same except for some enhancements and extensions which are essential steps towards a completely interactive Cool system. This platform will later be the basis for a full Cool implementation with all the bells and whistles of HyperNeWS.

Contents

List of Figures

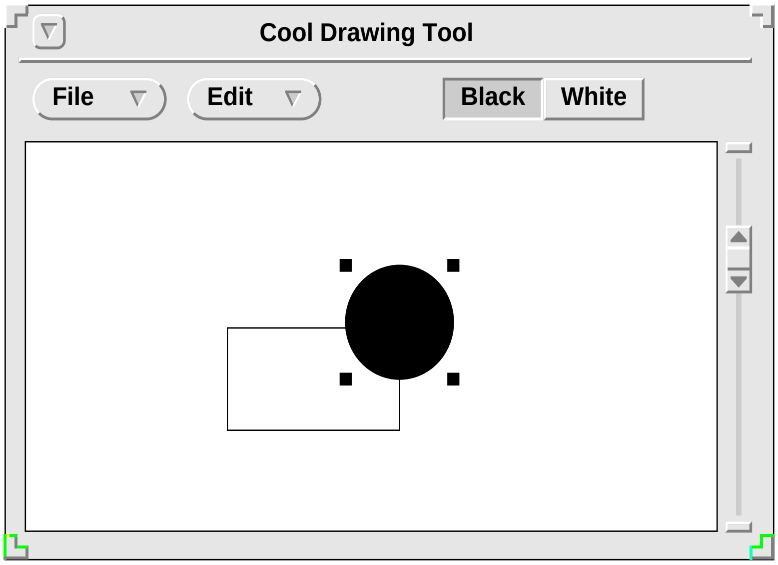

Figure 1: A Cool drawing tool

1 Introduction

The HyperNeWS 2.0 project has been successful in providing us with a pool of ideas which can be used in Cool. The demonstrator implementation was completely separate from TNT, which makes it unacceptable as a valid starting point for Cool. There would be too much duplication of effort and the maintenance would be a nightmare.

So, a way must be found in which Cool can be firmly embedded in the TNT environment. Cool should be an enhanced version of TNT. A better name should be *TNT++*. However TNT is a UI toolkit while Cool is an interactive UI development environment. This means that the two will never be the same. Some level of compatibility must be achieved, here are some guidelines:

- The existing TNT-API must remain the same both syntactically and semantically. It may be extended but not changed.
- Existing TNT applications should not be affected. In the worst case there should be a well defined upgrade path for existing TNT applications
- The performance of TNT should not be seriously affected by the introduction of new Cool features. However, we must be prepared to accept a small penalty in performance in exchange for the increased functionality.

I will discuss the Cool implementation of some of the HyperNeWS 2.0 features. From the HyperNeWS 2.0 project we know that TNT lacks some features which are essential to the implementation of Cool:

- *Parent scope* A mechanism in which properties are inherited along the container hierarchy.
- *Persistence* Because of the interactive nature of Cool applications it becomes necessary to be able to save and restore UIs and UI components.
- *Classing mechanism* Some extensions to the TNT classing mechanism are required when persistence is introduced. In HyperNeWS 2.0 two new flavors of classes were introduced *plug-in* classes and *shared* classes.

In this document I will discuss the incorporation of these features into the TNT system making it a better starting point for Cool. All proposed source code changes relate to version 2.0 of TNT running under OpenWindows version 2.0. The source code fragments should be seen as proposed changes and are not intended as final implementations. Most changes are made using `/installmethod` calls. I chose to use this very inelegant technique because it can be used to override existing TNT methods without having to modify the TNT source. In real life some of the changes will be made in the TNT source while other changes can be put into mixin classes which are mixed in with TNT classes.

1.1 Cool is easy to use

A lot of the features discussed here require the introduction of programming tools and mechanisms. Unlike TNT there needs to be only a small published API for Cool since most of the Cool facilities are not programmed. An interactive system has the advantage that the most complex operations are performed though tools with a well defined user interface. These tools hide the underlying complexity of the software and they require far less documentation. Some examples of these tools are: *drawingtool*, *property browser*,

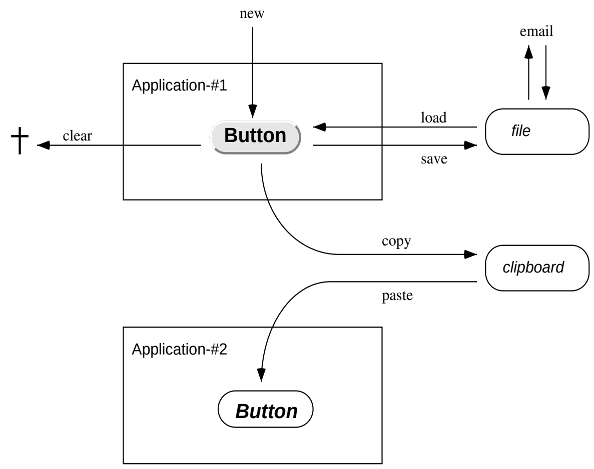

Figure 2: An upwardly mobile UI component

resource browser, *class browser*, *object warehouse*, *script editor*, *structure editor*, *message monitor*, *debugger* etc.

It is only when the Cool user wants to create new UI components or when he actually wants to write applications that he will use the programming environment provided by Cool One of the aims of Cool is to lessen the need for programming experience. This makes the system accessible to many more people. Making the system easier to use and to program is achieved as follows:

- *Interactive UI design*. This of course is the major achievement of Cool. It allows the interactive creating and modification of OpenLook compatible UIs without any programming knowledge.
- *Increased component reusability*. It will be very easy to copy and paste components and even whole applications around. Each of these components may have a very complex internal structure which requires external clients. This complexity remains hidden from the user.
- *Scripting using PdB*. The introduction of a high level scripting language will lower the threshold for first time user to attempt programming in the Cool environment.
- *Simple structure*. A simple container hierarchy together with the `Send` primitive for communication guarantee an easy to understand programming environment for the user.
- *High level tools*. The introduction of tools such as the property browser and the script editor etc. will lower the threshold for users to modify and program the system.

1.2 Cool components are upwardly mobile

In an interactive system UI components will be much more mobile that the current TNT components. It will be possible to cut and paste components across window and application boundaries. Also components stored in files may be moved around in the file system or even across the network. Ways must be found in which the Cool system can guarantee that components and application can function in this dynamic environment.

Components must be flexible enough to adapt to new environments. One of the mechanism which makes this possible is called *parent scoping* (see Section 2.1). It allows components to use values exported by their parent containers. For example a button which is copied from a 2D black and white application and which is pasted into a 3D pale-green window should appear in the appropriate colors using the same font as the other buttons in the window.

Not only should a UI component be able to adapt to its environment but it also should be able to communicate with its new parents. This requires the implementation of a `send` primitive which is able to address objects symbolically (see Section 2.2) and which can send messages along the container hierarchy (see Section 2.3).

Application boundaries vanish once individual UI components can have client programs associated with them (see Section 2.4). Copying and pasting components now means incorporating complex client software into your application at no extra cost. There is great potential for a share-ware community based on sharing UI components instead of complete UIs[1].

Another advantage is that these component applications are much easier to reuse because they have a well defined interface with the outside world (see Section 2.3). It will be much easier to copy, modify and reuse complex components such as drawing surfaces, editors etc.

A good example are the `Jot` text objects. Currently there is no common `Jot` server and every TNT client must implement a rather complex interface. In Cool the `Jot` text objects will use a common `Jot` server. This means that there is no longer a need to program each of them. The user can copy and paste these objects around without having to worry about the internal complexity of them. He will use just use the simple programming interface or he may decided just to modify the object through its property editor.

1.3 Cool Application structure

Cool will make it possible to interactively design OpenLook UIs and UI components. The design and prototype of HyperNeWS 2.0 proved that this can be done effectively in a direct manipulation environment. The interactive design process is quite separate from the application and it therefore becomes necessary to store the UI components separate from the application source code. Currently a TNT application has the following structure[2]:

1. Define new classes using sub classing.
2. Create, modify and initialize UI components.
3. Interaction and client communication.

The first two steps which define, create and initialize UI components must be formalized so that they can be separated from the application source code to allow interactive design and editing. The resulting application structure will be:

1. Locate and connect to already existing UI components
2. Interaction and client communication.

1.4 Cool Component structure

Lets define what constitutes a UI component once it is no longer part of the application source code as was proposed in the previous section. Currently a TNT UI component can be created using the following programming techniques:

[1] This is similar to the HyperCard system where external resources have become the commodity of trade.

[2] This may vary between application and must be seen as an abstraction.

1. Create new UI component classes using subclassing using `classbegin` and `classend`.

2. Install new methods in UI component instances using `/installmethod`.

3. Modify parameters of UI component instances through the API or directly.

Each of these must be represented in Cool. Additionally they must be structure so that they can be manipulated and edited interactively separate from the application.

1. *Creating new UI component classes using subclassing*
   In HyperNeWS 2.0 the concept of *plug-in* classes was introduced to allow dynamic subclassing in order to create new UI components. In Cool these will be called *resource* classes (see Section 3 and Section 4.1). The resource manager is used to load sub classes when required. These classes are stored in files separate from the UI and are loaded when required.

2. *Install new methods in UI component instances*
   It is important that small modifications to UI components can be made easily. In TNT this is done using `installmethod`, Cool uses scripts. A script is the source text of the modification in either PostScript or PdB. The HyperNeWS 2.0 concept of *shared* class was introduced to allow efficient scripting (see Section 4.2). In Cool there are called these *script* classes.

3. *Modifying parameters of UI component instances*
   It must be possible to modify TNT component parameters either through direct manipulation or through dialogs. The results of these modifications must be persistent which means that UI components must be savable and reloadable after they have been modified by the user (see Section 5).

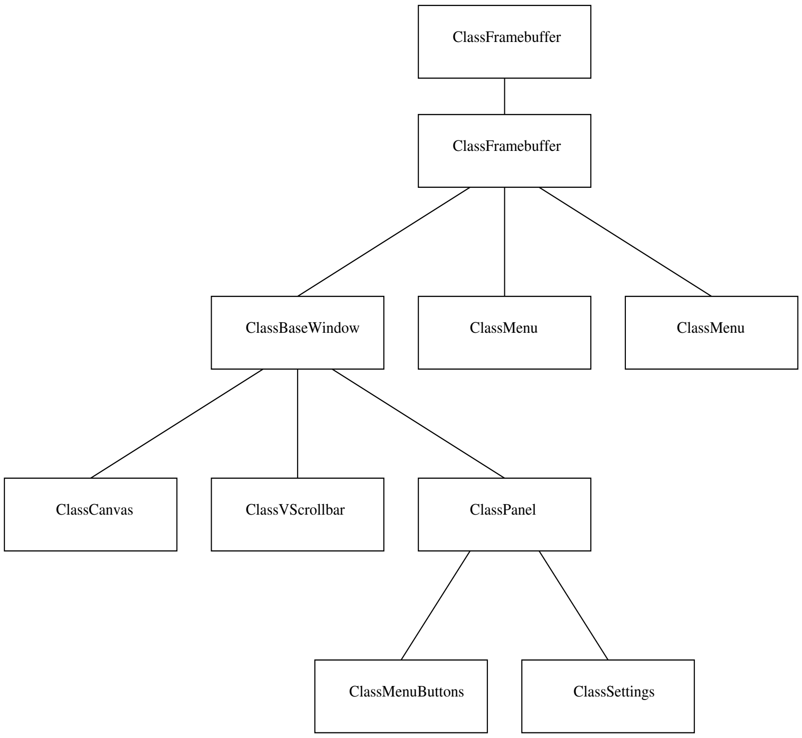

Figure 3: The Canvas container hierarchy of the drawing tool

2 Container hierarchy

The container hierarchy of TNT is based on the canvas hierarchy. Each component has a `Parent` field which points to the component's parent canvas. This 2 dimensional type of containment is similar to that required by Cool however there are examples where logical containment is more desirable.

For example, in TNT all windows have the framebuffer as their parent container. It is desirable that sub windows (such as notice popups) inherit from their super window and not from the framebuffer. The same is true for menus, property windows and help windows. They should inherit from their invoker and not from the framebuffer directly. In this way each application may specify a style which varies in some ways from the overall style, this helps the user to identify the windows and other components belonging to an application.

A logical container hierarchy can be achieved by adding a new field to each component called `@parent`[3]. It points to the logical parent container. Also, each component has a list of members (the `/Members` instance variable). The following methods are added to `ClassDrawable` to manipulate the container hierarchy (see also Appendix B.1):

- `/@parent     % -- container (Variable)`
  The logical parent container of an object.

- `/Members     % -- [member...] (Variable)`
  A variable containing an array of all the members of this object.

[3]The '`@`' character indicates that the variable contains an object and it may be used as a symbolic address. See also Section 2.2.

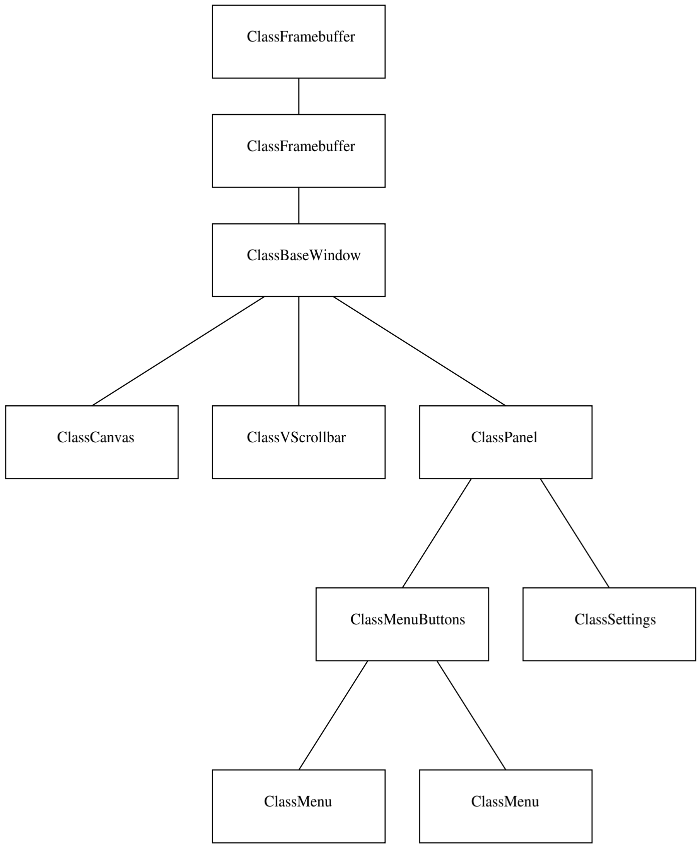

Figure 4: The logical container hierarchy of the drawing tool

- `/setcontainer % container|null --`
  Set the logical container of a an object. It will invalidate the object using `/invalidatecontainer`.
- `/container % -- container|null`
  Get the logical container of an object.
- `/invalidatecontainer % --`
  Invalidate this container and all the members it contains. This method will enumerate all members and send an `/invalidatecontainer` message to them. It must be called when an object changes a value which is accessible by its members.
- `/members % -- arrayofmembers`
  Return an array of all the members of this object.

Note that the `/setcontainer` method invalidates the object and its members. This is required to propagate the fact that new data is now accessible by the members of the container. The members get a chance to reinitialize their internal data. The `/setcontainer` method must be called when the logical container hierarchy is changed. This occurs in TNT in the following ways:

- The `/reparent` method of a canvas or a region is called.
- A window's subwindow status changes because `/addsubwindow` or `/removesubwindow` was called.
- The basewindow of a notice is changed using `/setbasewindow`.
- The invoker of a menu is set using `/setinvoker`[4].

As a result some extra work has to be done every time the container hierarchy is changed.

The references to parents and members must be softened so that the objects are destroyed when they become obsolete. There is no noticeable performance penalty except for the fact that objects are now invalidated more often than before.

2.1 Parent scope

In HyperNeWS 2.0 the implementation of parent-scope was proposed. This is a mechanism whereby values of variables are inherited along the container hierarchy. This should be implemented to achieve the following goals:

- Inheritance along the container hierarchy. This type of inheritance is quite intuitive and is easy to maintain.
- Increased component mobility. When a component is placed in a gray background it should adopt the correct colors to achieve the 3D effect.
- Intuitive property value management. Changing the colors of a window changes the colors of the objects inside the window too.

[4]This causes a bug since menus are displayed before their invoker is set. Any visual changes will not have effect until the menu is show again in the same context.

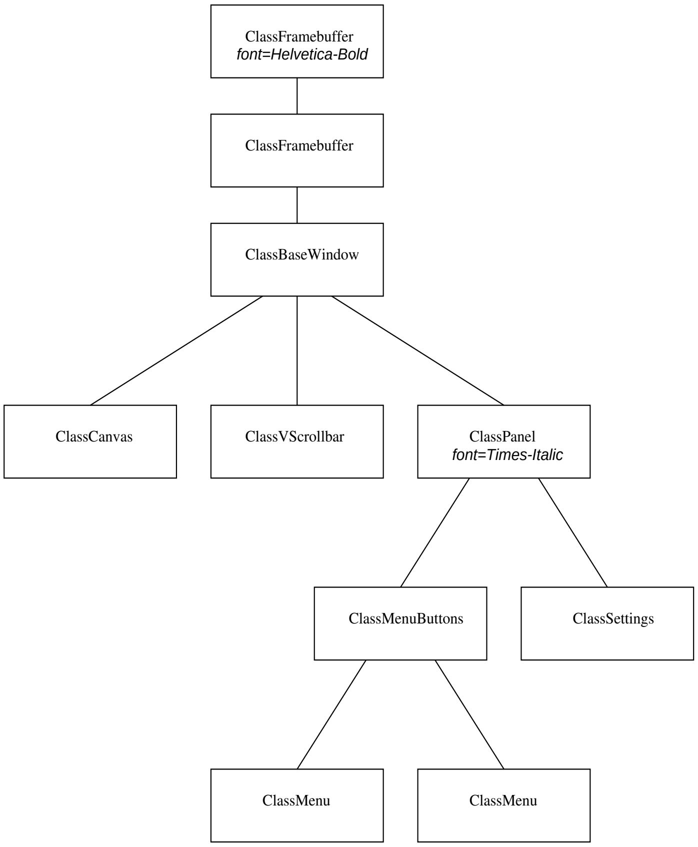

Figure 5: An example of a scoped font

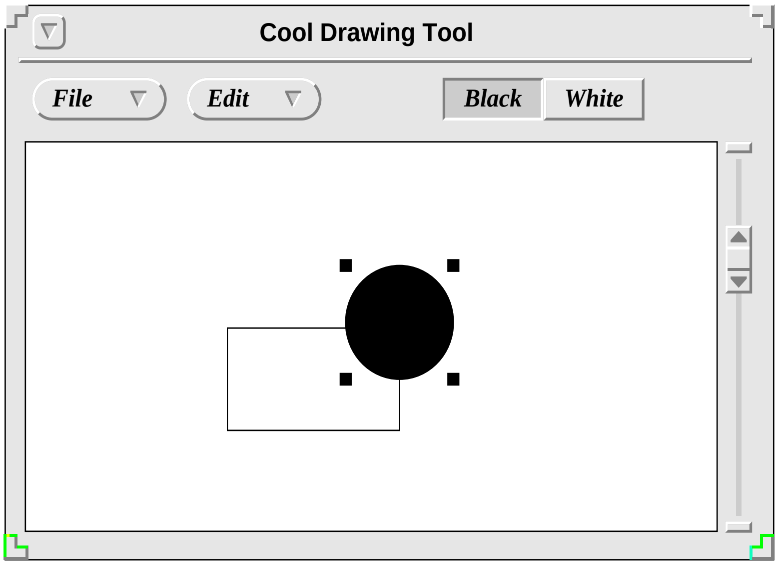

Figure 6: The resulting UI with the scoped font

- Powerful implementation tool. Interally there are many situation where scoped variables can be used.
- Support persistence. Scoped variables fit in well with persistence. Most scoped variables should be persistent so that their values are saved with the component.
- Avoid class modification. Changing a class variable is fine only if you can enumerate all the instances of the class which are going to be affected.

Scoped variables are a powerful concept. They can be used to manage the graphical aspects of a UI instance, such as colors and fonts but this mechanism can also be used to find elegant solutions to problems such as visibility, shared menus, notifiers, tracking etc. The increase mobility of UI components requires that when a component which is moved to a new context it must be able to adapt to its surroundings e.g. it must take on the same color, font etc. This is all achieved using parent scope.

TNT modifies class variables to change class defaults. This strategy is rather dangerous in an interactive system such as Cool since it is hard to predict the resulting visual damage of such an action. Scoped variables should provide a cleaner and more intuitive way to manipulate defaults and object properties. The modification of scoped variables is not addressed here since there will be a Cool scoped variable manager which provides high level interactive access to the scoped variables which the user should be allowed to change.

The routines that deal with scoped variables should make use of the logical container hierarchy as discussed above. Several possible implementations are discussed in Appendix A none of them are completely satisfactory. Each solution has drawbacks which are unlikely to be resolved. I have chosen to implement and test the ‘‘Dynamic lookup’’ solution. This method requires the implementation of a server routine which searches the linked list of parent instances for the scoped variable. The PostScript solution (described in Appendix B.2) is slow but it is fast enough for testing.

Not using separate localdicts creates the problem of name space clutter. This can be avoided by using a naming convention. The scoped variable naming convention used here is that all scoped variables must start with a special character: a ‘`&`’ for scoped variables, and a ‘`@`’ for scoped names (see Section 2.2). The following operators are provided to implement scoped variables: (see Appendix B.2):

- `linkwhere   % dict linkkey key -- dict true | false`
  This operator searches a linked list of dictionaries which are linked together using the `linkkey` field in each dictionary. If `key` is found in a dictionary of the linked list then the search is aborted and the dictionary and `true` are returned. If the key is not found or if the search fails otherwise, `false` is returned.
- `linkget    % dict linkkey key -- value`
  Lookup a variable in a linked list of dictionaries using `linkwhere`. If the variable is not found in the linked list of dictionaries it is executed in the current context (the back-stop).
- `linkknown  % dict linkkey key -- flag`
  Return true if the `key` is known in the linked list of dictionaries.

This implementation provides for a back-stop. This is useful since TNT components often have no parent while the UI is being created. In Appendix B.3 some of the TNT variables are modified so that they become scoped. This may mean a minor change to the TNT API since some instance variables are now no longer variables but instead refer to the corresponding scoped variable. The definition of a scoped variable should have the form:

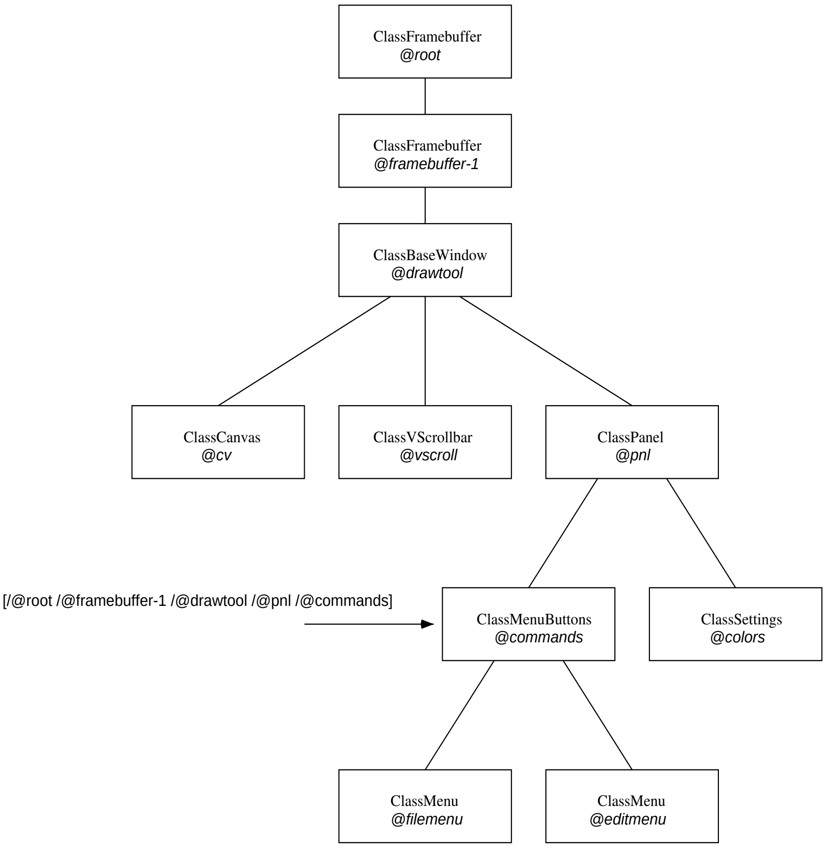

Figure 7: The logical container hierarchy with names

```
/&NrBeeps 4 def %the backstop

/NrBeeps { % -- n
     self /@parent /&NrBeeps getscope
} def

/SetNrBeeps { % n --
     /&NrBeeps exch promote
} def
```

2.2 Addressing objects

In an interactive system it is necessary that every UI component can be addressed by a symbolic name. The address must have the form of a path name because of the tree structure of the container hierarchy. Such a path name can be relative (not starting at the root) or it can be absolute (starting at the root). The component addresses can be used in many ways:

- During editing to identify which object is being edited.
- As a target address to send messages to.
- Components which communicate to other UI components.

- Client programs which communicate with UI components.
- Identify objects uniquely inside debuggers and error handlers.

To implement an addressing mechanism it is necessary that every object is given a name and that these names are unique within the parent container of the object. TNT supports two separate mechanisms for binding a symbolic name to an object. The first is implemented in `ClassObject` by the `/name` and `/setname` methods. The second is the naming of `ClassBag` clients. The `/addclient` method is used to define named clients in the `ClientDict` instance variable. These client names are used to address objects during layout.

Cool must use the first mechanism because the client names already have a well defined meaning for layout. Cool requires symbolic names which do not necessarily relate to layout. The `/setname` method is used to assign names to objects when required. It must be changed so that it will not allow duplicate names within the same parent container. The `/setcontainer` method in `ClassDrawable` must be changed so that it can generate unique names for unnamed objects and so that it can make sure that duplicate names don't occur (see Appendix B.4). The following methods are defined in `ClassDrawable` to implement unique symbolic names for objects.

- `/setname    % name --`
  Assign a new name to an object. This is only successful if the name is not yet defined in the parent container of the object. If the name does not already start with a '`@`' character it will insert one.
- `/setcontainer    % container|null --`
  Change the container of an object. Remove the name of the object for the old parent container and define the name of the object in the new parent container. A new unique name is assigned to the object if the object does not already have a name or when the old name would cause a name clash in the new parent container.

Each name is made a scoped variable in the parent container (the same technique is used in HyperNeWS 2.0). This make the lookup of an object name very efficient since it can use the same mechanism as is used for scoped variables. To distinguish object names from other scoped variables they are prefixed with the special character '`@`'. This clearly marks a PostScript name as an address. It also allows some objects to have generic names such as `/@parent`, `/@window`, `/@canvas`, `/@inputfocus` etc. The following methods are added to `ClassDrawable` to implement symbolic addressing. The implementation is given in Appendix B.5.

- `/findobject    % null|instance|name|path -- instance|null`
  Find an object given its path name. `null` is returned if the object is not found.
- `/address    % -- path`
  Return the address of an object (that is the full path name).

Binding an object to an address is done using the `/findobject` method (see Appendix B.5). An address can either be a single PostScript name or it is a path (an array of PostScript names). The root of the address is determined by its first element, consequent elements define which child is addressed. Some examples of valid addresses in the UI of Figure 7 are:

```
[/@root /@framebuffer-1 /@drawtool]
[/@drawtool /@pnl /@color]
/@drawtool
```

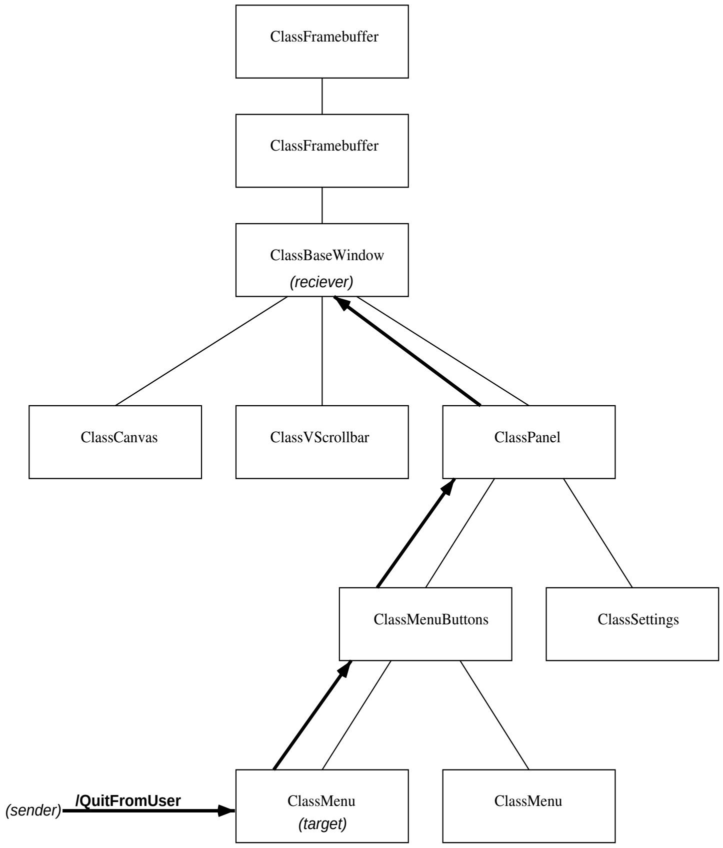

Figure 8: Sending a message through the logical container hierarchy

2.3 Send

HyperNeWS 2.0 makes use of the `Send` primitive. This primitive is used for communication between UI objects and for communication between UI objects and clients. The following methods are added to `ClassDrawable` (see also Appendix B.6):

- `/Send % arglist message address --`
  Send a message to an instance. The address is either an instance, an object name, a path or `null`. The arguments must be in an array so that they can be discarded if the message is not delivered.

- `/SendSender % -- instance`
  Return the object that initiated the current message.

- `/SendTarget % -- instance`
  Return the object to which the current message was initially send.

- `/SendReciever % -- instance`
  Return the object that received the current message.

- `/SendMessage % -- name`
  Return the current message name.

- `/SendArgs    % -- arglist`
  Return the current message arguments.

The `Send` primitive will deliver a message to the specified target object only if the object exists. If the target object exists it is used as an initial delivery target (see Appendix B.7). If the target does not have a method to handle the message then the message is passed on to the parent of the target. This is repeated until an object is found which does understand the message otherwise the message is discarded (see Figure 8). The resulting object is called the *receiver*. The message is executed in the receiver's context with the message arguments on the stack. To summarize, the `Send` primitive has the following features:

- *No guaranteed delivery* When the address of the target is not known or when no parent of the target understands the message then it is discarded.
- *No return value* The `Send` primitive cannot be used to obtain a return value from an object because the message may not be delivered.
- *Robust* The caller of `Send` will not be affected in any way if the receiver of the message causes an error.
- *Client communication* If an object has a client then `Send` can pass the message to the client. See Section 2.4.
- *Error handling* Each object may provide an error handler which is called when an error occurs within the context of a message (see Appendix B.8).
- *Logging* If required the `Send` primitive can log messages in a message log (see Appendix B.9).

The `Send` primitive makes the implementation of reusable objects very easy. If an object requires external functionality it should do so by sending a message to itself. The message is passed along the container hierarchy until some object is found that knows how to respond to the message. Take for example a ''quit'' button. It should send the message `/QuitFromUser` to itself. The message is intercepted by the window container which will close and quit. The quit button can be copied and pasted into another environment where it remains functional.

The default behavior of `ClassControl` needs to be modified so that it uses `Send`. This is shown in Appendix B.10. The default notifier of a control is now `/Action`. If the target is `null` and the notifier is a non-executable then the notifier message is `Send` to the control itself.

2.4 Client communication

A mechanism for client communication will be added to the already exiting CPS and Wire interfaces. The Cool client interface will mainly rely on the Wire protocol. To establish communication with the UI a client program opens a connection to an UI component. It may communicate with the UI through the UI component. The connection's execution always occurs in the context of the connected UI component.

The client may communicate with the component by expressing an interest in handling messages which are not handled by the component itself. The Cool client interface will intercept the requested messages and execute callbacks in the client as required. Figure 9 shows how this can be done. This can be implemented by modifying the `/deliverparent` method (see Appendix B.7). The message communication to the client can be efficient because the Wire interface can be used to allocate usertokens for the requested messages.

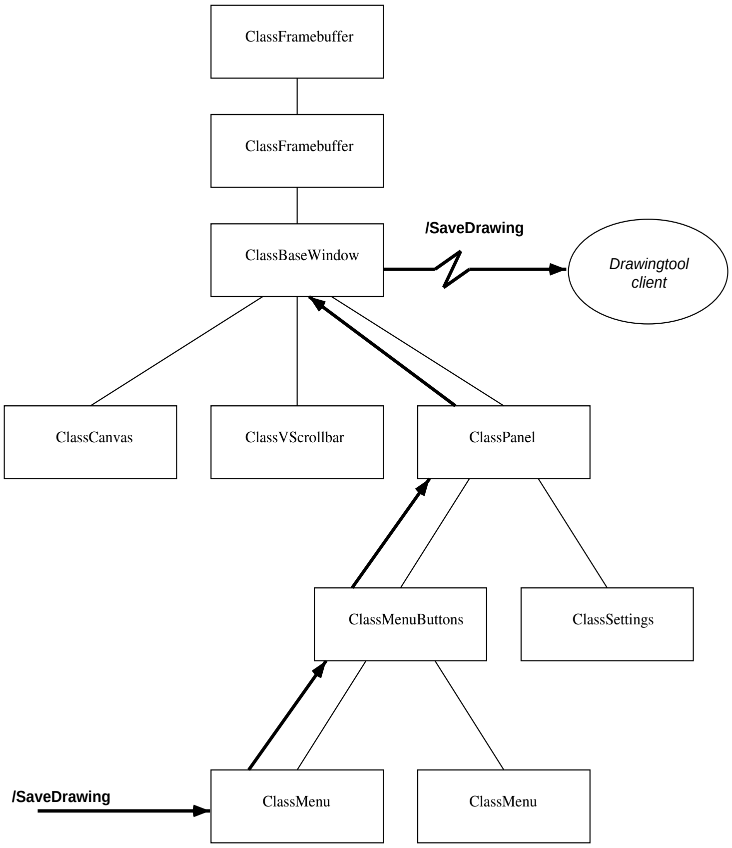

Figure 9: A message intercepted by a client program

It must remain possible to use the standard CPS and Wire techniques for efficient byte encoded communication. Also it must be possible to have *implementor* clients attached to objects which are used to maintain the internal data of the object (for example Jot objects).

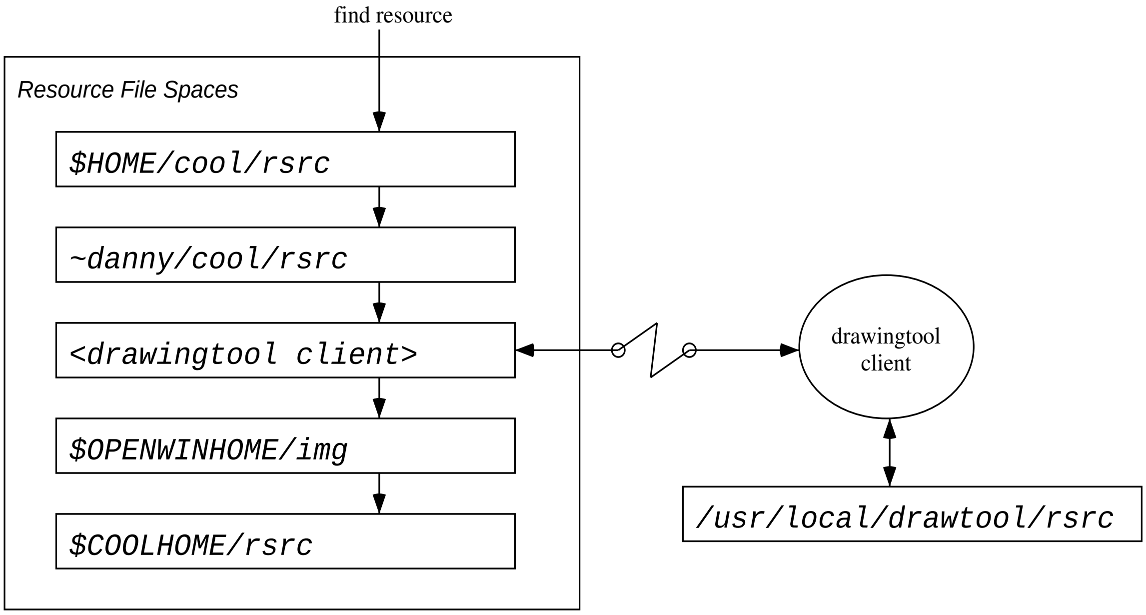

Figure 10: Obtaining a resource from a remote client

3 Resources

Cool will have to access files to load and save resources such as classes, UI components, graphics etc. File access is very awkward in NeWS because PostScript programs all execute in the server file space. Application UIs may want to access files which are in client file space. For this purpose a resource manager is used. The resource manager is used to located resource files. Location of files is done by searching several file spaces. The user must be able to specify in which order the file spaces are searched.

A client with a private set of resources may register with the resource manager which resources the client can provide. When the resource manager requires one of these resources it will ask the client to provide a file descriptor from which the resource can be loaded[5]. The resource manager is not responsible for the loading of the resource, it is merely there to bind a resource identifier to a file descriptor.

Appendix B.11 shows the implementation of a resource manager which can search file spaces as defined in a search path. It cannot yet handle client file spaces as discussed above. It provides the following functionality:

- `resourcemgr % -- instance`
  Return the system resource manager.

- `/find % name -- location file true | name false`
  A method of the resource manager which is used to find a resource given its name. If the resource is found it returns the location where the resource was found, the file descriptor from which the resource can be read and `true`. If the resource is not found the resource name and `false` is returned.

A resource can be anything which is stored in a file. For example classes, icons, image data, postscript source. The resource manager must be extended so that it can also save resources. The resource manager needs a Cool resource browser so that the user easily and specify and browse the resource file spaces.

[5]This can be done using the Wire service.

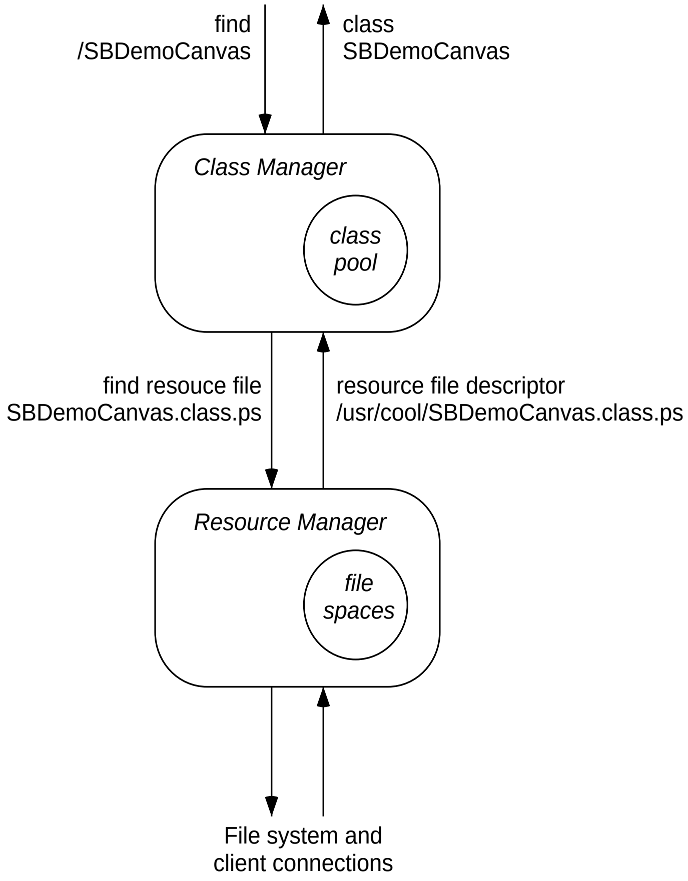

Figure 11: A class manager

4 Classes

In the following sections implementation of *resource* classes and *script* classes is discussed. Note that these classes are ordinary TNT classes with some added flavor. Both types of class are automatically deleted when they are no longer used. This is achieved using the `ObsoleteService` (see Appendix B.12).

4.1 Resource classes

A resource class is a class which is loaded when required and deleted when it becomes obsolete. The concept is similar to the *plug-in* classes that were used in HyperNeWS 2.0. The class is stored in a resource file which is accessed using the resource manager (see Section 3). The resource name of the class is derived from the classname and has the form ''*classname.class.ps*'' (see Figure 11). If the class is not found by the resource manager then the `systemdict` is checked since the requested class may be a TNT class.

The resource classes are loaded by a class manager which is called `classmgr`. It also takes care that the classes are unloaded when they are no longer required. The class manager understands the following method: (see Appendix B.13):

- `/find     % classname -- class|null`
  Find a class using the resource manager. If the class is not already loaded an attempt will be made to load it from the file *classname.class.ps*. `null` is returned when the class is not found.

A resource class file should contain the following:

```
[superclassnames .... ] instancevars classbegin
....
classend
```

The file must contain an ordinary TNT class definition except that the class is left on the operand stack. Also the superclasses are not directly referenced, instead their names are given so that they may be loaded using the class manager. Below is an example of a trivial resource class with one method `Hello`.

```
/ClassObject [] classbegin
/Hello { % --
    console (Hello world\n) fprintf
} def
classend
```

In the following example the above class is loaded from the file ‘‘hello.class.ps’’ using the `/find` method of the class manager. Assuming that the resource manager locates the class an instance is created and the `/Hello` method is called.

```
/hello /find classmgr send
/new exch send
/hello exch send
```

4.2 Script classes

In an interactive environment is it required that the user can make interactive changes to the UI components. This direct method of programming UIs is called scripting and is implemented in Cool using *script* classes. This concept is similar to the *shared* classes used in HyperNeWS 2.0. A script class is a class which contains its own source text so that the user may edit it. They are called script classes because these classes contain a script which is shared by as many UI components as possible. The following methods are added to `ClassObject` to implement script classes (see also Appendix B.14):

- `/scriptclass  % -- class|null`
  Return the script class of an instance or return null of the instance does not have a script class.

- `/getscript    % -- script|null`
  Return the script of an instance or return `null` if it has none.

- `/setscript    % script --`
  Set the script and change the script class of an instance. This means that the `/ParentDictArray` of the instance is changed.

To demonstrate that script classes can be applied to any TNT class, the `/sillyfooter` method in the `/demo` method of `ClassButtons` is replaced by a script. Note that the `/sillyfooter` method must be `undef`ed from the window instance. The script class is installed below the instance in the `ParentDictArray`, methods which are installed in the instance therefore have priority over methods in the script class.

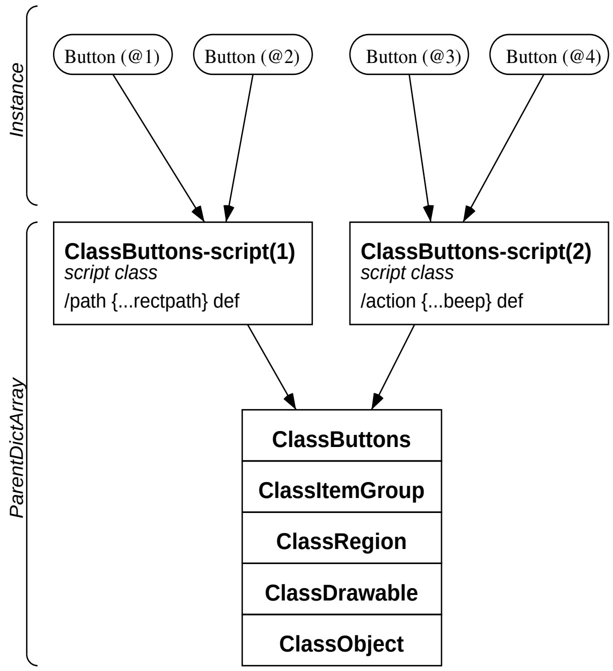

Figure 12: An example of script classes

```
/demo ClassButtons send
1 index /sillyfooter undef
(/sillyfooter { % index buttons => -
     /item exch send
     (The % is extremely intresting...) sprintf ()
     /setfooter self send
} def) /setscript 3 index send
```

Figure 13: TNT component states

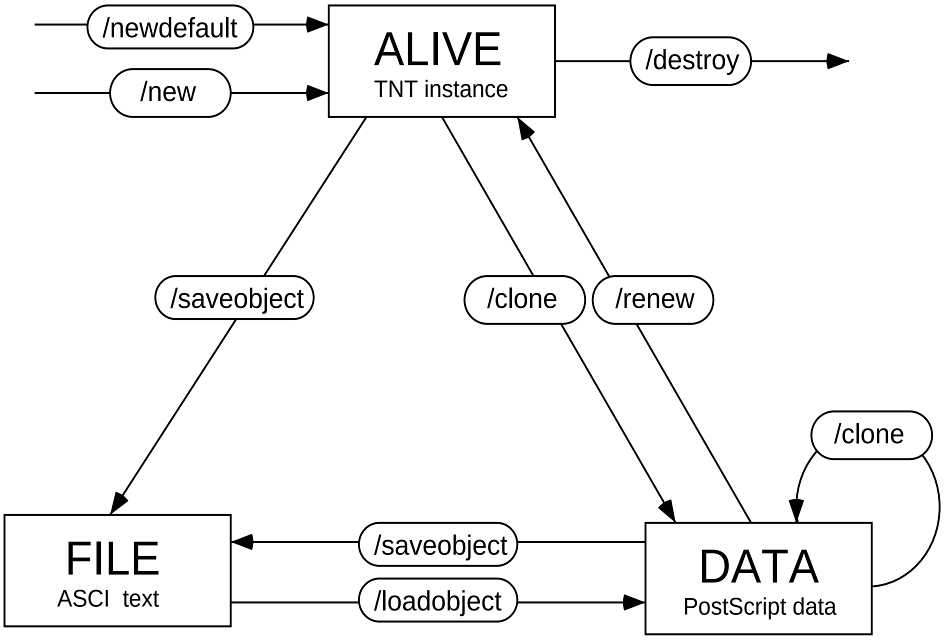

Figure 14: UI component states in Cool

5 Persistence

This section will discuss the problems involved in making TNT UI components persistent. A persistent UI component is a component which can be saved to a file, loaded from a file, copied and pasted etc. The features of a component are described by its instance variables. Each of these is marked as persistent or not. A persistent variable is retained when the object is copied or saved.

It is not necessary that every TNT class be persistent. Only the relevant UI component classes need to be persistent. And of those only some of the information needs to be saved. For example when saving a window, its eventmanager doesn't need to be saved since it can be recreated easily from scratch. What needs to be saved is the label of the window its icon image etc.

5.1 Persistent states

A persistent object can be in one of three states:

- *ALIVE* The object is in working order and is part of the UI. It is a part of TNT, its event manager is running.
- *DATA* The object is represented by a PostScript data structure which contains only the persistent data of the object. This complex object consists of uninitialized instances.
- *FILE* The object is represented as PostScript text in a file. The file contains only the persistent data of the object. When loaded it becomes *DATA*.

TNT knows only the first of these states (see Figure 13). In Cool these states must be represented and the transitions between them must be well defined. Figure 14 show these states and the transitions between them.

5.2 Creating new UI components

In an interactive environment it is necessary that the user can create new instances of UI component classes without having to program how they are created. Strictly speaking this is not concerned with persistence. It requires the introduction of a uniform mechanism to create a UI component from its class. The `/newdefault` method creates a new instance given a parent. It takes only one argument and it this is the same for every UI component class:

- `/newdefault   % parent -- instance`
  A class method to create an initialized instance of a UI component class. The instance should be ready to use, all its parameters are set to the most frequently used defaults.

In Appendix B.15 you will find the `/newdefault` methods for the TNT UI components. See also Appendix C.6 for a demonstration of how this method is used.

5.3 Persistent variables

Some mechanism is needed to mark persistent variables. Therefore a method called `/persist` is added to `ClassObject`. It is used to mark instance variables as persistent. (see also Appendix B.16).

- `/persist    % name|listofnames --`
  Mark an instance variable (or a list of instance variables) as persistent. The variable will be copied and/or saved only if the instance variable is promoted.

Note that the persistent instance variables are only copied and/or saved when they have been promoted. Persistence is inherited when subclassing, it therefore requires the modification of `classbegin`. If a variable is persistent in one of its super classes then it is persistent in this class. For example the instance variable `/Width` is declared to be persistent in `ClassRegion`. This means that it also persistent in `ClassButtons`, `ClassSettings` and every other subclass of `ClassRegion`.

The order in which variables are declared to be persistent affects the order in which they are saved when written to a file. This may not seem significant but it is important when using SCCS to maintain the file. Changing the order of the persistent variables may cause major changes in the file which affects the efficiency of SCCS.

5.4 Cloning

A clone is a copy of the persistent UI component data. Cloning is used when the user is editing UI components. Whenever a component is copied to the clipboard, or whenever it is duplicated then a clone is created. This clone is a PostScript *DATA* structure. The clone of an instance is either a dictionary or a canvas object (depending on the original) containing only those instance variables that were marked as persistent. The following methods are added to `ClassObject` in Appendix B.17:

- `/clone % -- clone`
  Create a copy of the instance and its persistent data. Reference checking avoids cyclic objects or objects that are referred to more than once being copied twice.

- `/cloneinstance % clone -- 'clone`
  Called when an instance has been cloned. It may be subclassed in order to perform some special action when cloning.

- `/clonething % any -- 'any`
  Clone an PostScript data object. It may not be possible to clone every NeWS object. For example `eventtype` objects cannot be cloned their clone is `null`. This method is usually called from the `/cloneinstance` method to clone data objects.

5.5 Renewing

A new protocol must be introduced to create an *ALIVE* instance from a PostScript *DATA* instance. The method is called `/renew` (see Appendix B.18).

- `/renew % parent --`
  Renew an instance from data.

The `renew` methods is similar to `/new` except that the instance is already created and it is already in a state where most of its parameters have been assigned a value. It may be subclassed in order to perform some special operation. An example program which use the `/renew` method in combination with the `/clone` method can be found in Appendix C.7.

5.6 Saving

The `/saveobject` method (see Appendix B.19) is used to save an instance to a file.

- `/saveobject % file --`
  Save an instance to a file. Care is take that objects which are referenced more than once are saved only once. This avoids problems with cyclic data structures.

- `/saveinstance % file --`
  Save the content of an instance. May be subclassed to perform some special operation when saving.

- `/savething % file any --`
  Save any PostScript data to a file. Note that not all PostScript data can be saved. For example `eventtype` objects are not saved, `null` is saved instead. Also path objects cannot be saved since they cannot be enumerated.

5.7 Loading

Appendix B.20 describes how UI components can be loaded from a file after being saved using the `/saveobject` method.

- `loadobject      % file -- data`
  A PostScript *DATA* object is loaded from the file. Cyclic and multiple references are recreated where appropriate. Note that `/renew` must be used before the object can be used as a Cool instance.

Note that you must send a `/renew` message to the resulting *DATA* object before it can be used. An example program with the `/saveobject` method and `/loadobject` method in combination with the `/renew` method can be found in Appendix C.8.

6 Conclusions

It is possible to build Cool on top of TNT. It all depends on how much of the changes can be incorporated into TNT without affecting existing TNT development.

It must be stressed that each of the proposed TNT extensions are more or less essential to the success of Cool. Each of these may not seem essential on its own but together they form the back bone of Cool. You can compare it to a Mercedes car. If you strip all the luxuries, including the paint, the wind screen wipers and the soft chairs, you will end up with something that is still a car but nobody will use it. Hopefully this will not happen to Cool.

Persistence is a major thread to the success of the Cool system. It exposes the interior of TNT by saving it to a file. Any major change to TNT or Cool will render previously developed UIs useless, unless upgrading programs are provided.

The relation between Cool and TNT software is important. It will be fairly easy to port an existing TNT program to Cool if the programmer follows a few simple steps. Porting from TNT to Cool.

- *subclassing*
  Create a resource class for each subclass in the TNT source. Make sure that the class manager is used to access the classes. Small subclasses can be put into script classes.

- *installmethod*
  Instead of using `installmethod` use script classes.

- *resources*
  Any images icons etc. used by the program should be loaded through the resource manager.

- *UI components*
  Save the UI components. Either recreate them from scratch using Cool or run a conversion program which takes a TNT application window and saves it in a file as a Cool UI component.

- *client program*
  The client should use the Cool client interface which will be based on the Wire service. This means that the Wire and CPS functionality can still be used.

  1. Connect to NeWS.
  2. Tell the resource manager which resources are available.
  3. Locate and connect to the required UI components.
  4. Event loop.

Each of these steps is fairly new, but you should realize that some of the proposed functionality will already be part of TNT before the introduction of Cool. This will eliminate some of the steps above.

Porting from Cool to TNT will be much harder. It means that a lot of functionality must be thrown away. It is up to the programmer to decide whether that is possible.

6.1 Remaining problems

Here is a list of the problems that still have to be solved.

1. *Interactive editing*
   Creating and positioning of components within the UI.

   - Change `ClassTrackService` so that in edit mode, tracking events can be captured by Cool before they get send to the UI.
   - Visual selection feedback in both color and black and white. The problem here is that some canvases are not transparent. It is hard to draw selection rectangles which overlap opaque canvases.
   - The interaction between editing and layout must be formalized so that the user is allowed limited editing capability if automatic layout is in effect. Perhaps it is required to introduce new types of layout for Cool.
   - There are no facilities to handle overlapping UI components. There should be a `/tofront`, `/toback` methods for every UI component. It is not clear how this will affect layout.

2. *Multiple pages*
   It should be possible for windows to have more than one 'page' or card. The requirement for multiple pages becomes acute when a lot of information must be presented to the user. This is often the case and the ''window with multiple pages'' model is successful because it is easy to grasp. Each window must have a background page (the window itself) which is always visible plus several pages which overlay the background page.

   - Pages can be implemented by inserting another layer of containment in `ClassWindow`. Assuming that this can be done without causing inconsistencies with the TNT API for window objects. The existing API would always affect the current page of the window.
   - There must be a protocol which allows components to perform actions when they become visible. In HyperNeWS 2.0 an `/onopen` message or an `/onclose` message is sent to an object when its visibility changes. This object has a chance to start or conect to a client when it receives the `/onopen` message while the `/onclose` message can be used to kill or disconnect from the client.
   - There is no satisfactory way for an object to find out whether it is visible or not. Note that a canvas may be mapped but at the same time invisible because one of its parents is unmapped. This can be solved by introducing a scoped variable for visibility.
   - It will be very expensive to have many objects in a window when they all use canvases. In HyperNeWS we found that users build interfaces easily consisting of up to 500 components. This would mean that there would be very many inactive canvases floating about.
   - `ClassNotice` is not a subclass of `ClassWindow` it can therefore not have multiple pages. Either the same functionality is added to `ClassNotice` or `ClassNotice` must be made a subclass of `ClassWindow`.

3. *Single button objects*
   It seems likely that may users will want to create single button objects each with their own script which are combined together into a group of buttons which has the same properties as the existing `ClassButtons` object. Perhaps `ClassButtons` can be changed so that it actually creates instances for each item in the itemlist. The same applies to `ClassSettings` and `ClassItemGroup`.

4. *Robust UI components*
   If the user has been editing the UI it should not crash before it can be saved. Also there may be many windows and components sharing the Cool LEM which makes such a crash much more dangerous. In HyperNeWS this is solved by using the `Send` primitive whenever user code (from resource classes or scripts) is executed. This means a performance penalty. Unless `Send` is implemented in the server an alternative solution must be found. Increased use of `Send` may also affect existing TNT programs.

5. *Multi line text*
   Multi line text editing and display is needed for scripting. Using Jot has the disadvantage that the actual text data is stored on the server side. This is a general problem with clients that hold data which is to be stored with the UI. Jot will have to override the `/cloneinstance` and `/saveinstance` methods so that first retrieve the text data from the jot client.

6. *Tool events*
   There will be several tools for editing and monitoring changes within the Cool system. This requires that in some cases these tools are warned that their data has changed. For example the class browser should be told when a resource class is loaded or unloaded. This can be done using the event mechanism.

6.2 Future plans

The implementation of a complete Cool system will consist of the following steps.

1. *Embedding in TNT* The issues which are discussed in this document must be incorporated into TNT.

2. *Interactive editing* This will probably mean changes to the `TrackService`. Also changes to `ClassLayout` must be anticipated.

3. *Client communication* This is required to communicate with Jot for example.

4. *Property windows* The design and implementation of property windows for each of the UI component classes.

The result of all this work will be an an operational system where you can create and edited UI interfaces and connect them to clients etc. This system is much too bare to present to the user. A lot of the tools which were discussed must now be implemented using Cool. While the tools are being implemented some TNT applications can be ported to the new system. The tools that are essential:

- *Object warehouse*. A collection of predesigned UI components ready for use. The user can copy and paste objects from this tool into his own application.

- *Drawing tool*. Essential tool to provide graphical input. It can be used to design displayitem objects.

- *Color pallet*. Tool to choose colors or gray scales. This may include a triangular or circular color spectrum from which the user can pick new colors.

- *Property browser*. Similar to the system properties window but it is used to modify properties of any object inside the Cool UI. It is specifically useful to set and examine scoped variables.

- *Error handler*. A tool to catch and display errors. It should provide enough feedback to the user so that the cause of the error may be retrieved.
- *PostScript debugger*. A high level interface to the existing PostScript debugging tools.
- *Message logger*. A tool which logs the messages that go through the system. This tool is essential when developing new UIs and applications.
- *Resource browser*. Allow the user to add and remove directories which are searched by the resource manager. Also provide feedback as to where resources can be found or not.
- *Client browser*. A tool to monitor the existence of clients. There may be many clients connected to Cool at one time, this tool can be used to locate and kill ones that are no longer needed.
- *Class browser*. Examine the class hierarchy including resource and script classes. For each class the user can ask for a list of the possible methods, where these originate etc.

A Parent scope implementation notes

A.1 HyperNeWS 2.0

Insert the parent instance dictionaries in the `ParentDictArray` so that name lookup will search the instance, its classes and its parent instances (in that order). For example:

dictionary stack
instance
class ...
parent instance ...
...

- *Advantages* It is very fast since it is based on standard name lookup. No special operators for access/assignment are needed.

- *Disadvantages* Every promoted variable is more or less a scoped variable. Adding parent instances dictionaries to the `ParentDictArray` can be both powerful and dangerous. There must be system defined defaults (backstops), therefore the use of scoped variables must be restricted to a fixed set of system defined scoped variables. The `ParentDictArrays` can no longer be shared[6].

A.2 Local dicts first in `ParentDictArray`

Each instance has a local dictionary containing its scoped variables. The local dictionaries of the instance and of its parents are inserted into the object's `ParentDictArray`. Name lookup will search the instance, its local dict, the parents local dicts and its classes. For example

dictionary stack
instance
own local dict parent local dict ...
class ...
...

- *Advantages* It is a much cleaner solution than the previous one since the scoped variables are clearly separated from the instance/class variables. Class variables do not override the scoped variables and can be used as backstops. Access is possible without the use of a special operators

- *Disadvantages* Every object must contain an additional dictionary to store the scoped variables. This will add at least 120 bytes to each instance. Assignment requires a special operator.

[6] In the HyperNeWS 2.0 design document a server based implementation is proposed which solves this problem.

A.3 Local dicts below class dicts in `ParentDictArray`

Same as the previous solution but the local dictionaries are added to the `ParentDictArray` in a different order. Name lookup will search the instance, its classes, its local dict and the parents local dicts.

dictionary stack
instance
class . . .
own local dict parent local dict . . .
. . .

- *Advantages* Same as the previous solution. Class variables now override scoped variables.
- *Disadvantages* Class variables can not be used as a backstop. Therefore the use of scoped variables must be restricted to a small set of system defined variables.

A.4 Recursive send

To find a scoped variable recursively search the container hierarchy using `Parent send`. The chain of sends is terminated when the value is found.

```
/getscoped {
    % variable-name -- value
    dup promoted? {
        self exch get
    } {
        /getscoped Parent send
    } ifelse
} def
```

- *Advantages* Simple implementation.
- *Disadvantages* Very slow. Also there is no backstop and therefore there cannot be an unrestricted set of scoped variables.

A.5 Dynamic lookup

To access a scoped variable use a special routine which dynamically looks up the variable in the parent instances of an instance. If the variable is *not* found it assumed to be defined in the current context.

```
/getscoped {
    % instance scoped-variable-name -- value
    {
        2 copy known {get exit} if
        exch /Parent get
```

```
        dup null eq {pop cvx exec exit} if
        exch
    } loop
} def
```

- *Advantages* A scoped variable is defined by its method of access. Any promoted instance variable can be accessed as if it is a scoped variable. No mucking about with the `ParentDictArray` or additional dictionaries saves a lot of space. Class variables can be used as backstops.
- *Disadvantages* It requires a server based implementation otherwise it will be too slow.

A.6 Dynamic lookup with local dicts

Same as the previous solution except that the instances now contain local dicts to contain the scoped variables.

- *Advantages* Scoped variables are clearly separated from other variables.
- *Disadvantages* Same as the previous solution. In this solution it is possible to share empty dictionaries therefore the additional memory requirements may be acceptable. Special operators for access and assignment of scoped variables are needed.

A.7 Dynamic distribution

It is possible to distribute the value of a scoped variable to all the children. This a sort of super-promote, it promotes the variable not only in the current instance but also in all its children. Each instance must defined which scoped variables it 'imports' and which ones it 'exports'.

- *Advantages* Access will be immediate since all scoped variables are promoted.
- *Disadvantages* Assignment and promoting will be very slow. The administration required to keep the scoped variables and their values consistent with the scoped variables of parents and children will be enormous. Also each instance contains a copy of every scoped variable which it may use, this makes the instance dictionary rather large.

A.8 Shared arrays

If two variables refer to the same executable array containing a single value then it is possible to change the return value of both variables by changing the content of the shared array. This can be used to share values along the container hierarchy. A scoped variable which is inherited shares its value-array with the same scoped variable in the parent. Imagine that `/a`, `/b` and `/c` represent the same scoped variable at increasing levels in the container hierarchy:

```
/a {4} def
/b /a load def
/c /a load def

% change the value of /a, /b AND /c
/a load 0 5 put
```

- *Advantages* Access is immediate. Assignment at the root of the tree is immediate to.
- *Disadvantages* When a new value is assigned to a scoped variable which is not the root of the inheritance tree, all value-arrays of its children have to be changed. Each scoped variable must be known in advance. No backstop possibility therefore only a system defined set of scoped variables is allowed.

A.9 Shared nested arrays

Some of the disadvantages of the previous approach can be eliminated using nested executable arrays. Each executable array refers to the previous one indirectly. Example, `/a`, `/b`, `/c` and `/d` represent the same scoped variable at increasing levels in the container hierarchy:

```
/a {4 exec} def
/b {//a exec} def
/c {//b exec} def
/d {//c exec} def

% assign a new value to /a, /b, /c AND /d
/a load 0 5 put

% assign a new value to /c AND /d
/c load 0 6 put

% to link them back to the same value
/c load 0 /b load put
```

- *Advantages* Access is fast. Assignment at all levels is immediate.
- *Disadvantages* Access becomes a little slower as the number of nesting levels increase. Each scoped variable must be known in advance. No backstop possibility therefore only a system defined set of scoped variables is allowed.

B TNT extensions

B.1 Container hierarchy

```
%
% Implement a logical container hierarchy.
%

ClassDrawable /@parent null put
ClassDrawable /Members nullarray put

/setcontainer { % container --
        @parent 1 index eq {pop} {
                @parent null ne {
                        @parent /Members 2 copy known {
                                2 copy get [exch {
                                        dup self eq {pop} if
                                } forall] put
                        } {pop pop} ifelse
                } if
                dup null eq {
                        pop /@parent unpromote
                } {
                        soften /@parent exch promote
                        @parent /Members
                        2 copy known {2 copy get} {//nullarray} ifelse
                        [self soften] append put
                        /invalidatecontainer self send
                } ifelse
        } ifelse
} /installmethod ClassDrawable send

/container {    % -- container
        @parent
} /installmethod ClassDrawable send

/members {      % -- memberslist
        Members
} /installmethod ClassDrawable send

/invalidatecontainer {  % --
        valid? {
                /invalidate self send
                /members self send {/invalidatecontainer exch send} forall
        } if
} /installmethod ClassDrawable send

/destroy {      % --
        null /setcontainer self send
        /Menu unpromote
        /destroy super send
} /installmethod ClassDrawable send

%
% set the containers for the framebuffers
%

/children globalroot send {
        globalroot /setcontainer 3 -1 roll send
} forall

%
% Make sure that /reparent in ClassDrawable calls /setcontainer.
%
```

```
/reparent {     % parent --
        dup /setcontainer self send
        soften /Parent exch promote
} /installmethod ClassDrawable send

/reparent {     % parent --
        Mapped exch
        /reparent super send
        /Mapped exch promote
} /installmethod ClassCanvas send

/reparent {     % parent --
        /reparent super send
} /installmethod ClassRegion send

%
% newcanvas implicitly does a /reparent so we must execute
% an additional /setcontainer when a canvas is created
%

ClassCanvas /ClassCanvas-NewInit known not {
        ClassCanvas /ClassCanvas-NewInit
        1 index /NewInit get put
} if

/NewInit {      % --
        /ClassCanvas-NewInit self send
        Parent /setcontainer self send
} /installmethod ClassCanvas send

%
% adding subwindows changes the container hierarchy
%

ClassWindow /ClassWindow-addsubwindow known not {
        ClassWindow /ClassWindow-addsubwindow
        1 index /addsubwindow get put
} if

ClassWindow /ClassWindow-removesubwindow known not {
        ClassWindow /ClassWindow-removesubwindow
        1 index /removesubwindow get put
} if

/addsubwindow {         % subwin --
        self /setcontainer 2 index send
        /ClassWindow-addsubwindow self send
} /installmethod ClassWindow send

/removesubwindow {              % oldsubwin --
        {Parent setcontainer} 1 index send
        /ClassWindow-removesubwindow self send
} /installmethod ClassWindow send

%
% setting the basewindow of a notice changes the container hierarchy
%

ClassNotice /ClassNotice-setbasewindow known not {
        ClassNotice /ClassNotice-setbasewindow
        1 index /setbasewindow get put
} if
```

```
/setbasewindow {        % basewindow | null --
        dup null eq {Parent} {dup} ifelse
        Tail null ne {dup /setcontainer Tail send} if
        /setcontainer self send
        /ClassNotice-setbasewindow self send
} /installmethod ClassNotice send

%
% The invoker of a menu changes
%

ClassMenu /ClassMenu-setinvoker known not {
        ClassMenu /ClassMenu-setinvoker
        1 index /setinvoker get put
} if

/setinvoker {   % object|null --
        /ClassMenu-setinvoker self send
        Invoker null ne {Invoker} {Parent} ifelse
        /setcontainer self send
} /installmethod ClassMenu send

%
% Fix a bootstrapping problem with invalidate.
%

/invalidate {
        Parent null ne {/invalidate Parent send} if
        /invalidate super send
} /installmethod ClassMenuHeader send

/invalidate {
        Parent null ne {/invalidate Parent send} if
        /invalidate super send
} /installmethod ClassCommandMenu send

/invalidate {
        Parent null ne {/invalidate Parent send} if
        /invalidate super send
} /installmethod ClassSettingsMenu send

/invalidate {
        /invalidate super send
        /Center /client self send {
                /invalidate exch send
        } if
} /installmethod ClassPinnedMenu send
```

B.2 Scoped variable management

```
%
% Implement lookup of variables in linked lists
%
systemdict begin

/linkwhere {    % dict linkkey key -- dict true | false
        3 1 roll exch {
                % key linkkey dict
                dup 3 index known {
                        3 1 roll pop pop true exit
                } if
                1 index 2 copy known not {
                        pop pop pop pop false exit
                } if
                get dup null eq {
                        pop pop pop false exit
                } if
        } loop
} def

/linkget {      % dict linkkey key -- value
        dup 4 1 roll linkwhere {exch get} {cvx exec} ifelse
} def

/linkknown {    % dict linkkey key -- boolean
        linkwhere {pop true} {false} ifelse
} def

end %systemdict
```

B.3 Scoped variables

```
%
% Change existing TNT variables into scoped variables
%

/makescoped {   % var-name scoped-var-name --
        dup /understands? self send {pop pop} {
                2 copy exch self send def
                [/self cvx /@parent 4 -1 roll /linkget cvx] cvx
                /installmethod self send
        } ifelse
} /installmethod ClassDrawable send

/BG0            /&BG0                   /makescoped ClassDrawable send
/BG             /&BG                    /makescoped ClassDrawable send
/BG2            /&BG2                   /makescoped ClassDrawable send
/BG3            /&BG3                   /makescoped ClassDrawable send
/FG             /&FG                    /makescoped ClassDrawable send
/2DBG           /&2DBG                  /makescoped ClassDrawable send
/2DFG           /&2DFG                  /makescoped ClassDrawable send

/3D?            /&3D?                   /makescoped ClassDrawable send

/TextFont       /&TextFont              /makescoped ClassDrawable send
/FooterFont     /&FooterFont            /makescoped ClassWindow send
/IconFont       /&IconFont              /makescoped ClassBaseWindow send
/DragFrame?     /&DragFrame?            /makescoped ClassWindow send

%
% Promote colors in the right way
%

/setallcolors { % 2dfg 2dbg fg bg0 bg bg2 bg3 => -
        /&BG3 exch promote
        /&BG2 exch promote
        /&BG  exch promote
        /&BG0 exch promote
        /&FG  exch promote
        /&2DBG exch promote
        /&2DFG exch promote
        self isinstance? {
                /invalidatecontainer self send
        } if
} /installmethod ClassDrawable send

/setcolors { % fg bg => -
        3D? {
                % REMIND : These are not correct.
                dup +.80 ChangeBrightness /&BG0 exch promote
                dup -.15 ChangeBrightness /&BG2 exch promote
                dup -.50 ChangeBrightness /&BG3 exch promote
                /&BG exch promote
                /&FG exch promote
        }{
                /&2DBG exch promote
                /&2DFG exch promote
        } ifelse

        self isinstance? {
                /invalidatecontainer self send
        } if
} /installmethod ClassDrawable send
```

```
%
% Set the text font properly
%

ClassNotice /TextFont undef
ClassLabel /TextFont undef
ClassWindow /TextFont undef
ClassSettings /TextFont undef
ClassMenuHeader /TextFont undef
ClassCommandMenu /TextFont undef

/settextfont { % font => -
        /ModifyFont self send
        /&TextFont exch promote
        self isinstance? {
                /invalidatecontainer self send
        } if
} /installmethod ClassDrawable send

%
% Promote the /&3D? variable properly
%

/set3D {        % boolean|proc --
        /&3D? exch promote
        self isinstance? {
                /invalidatecontainer self send
        } if
} /installmethod ClassDrawable send
```

B.4 Naming objects

```
%
% Make sure that all UI components have names which
% are unique within exch container and which are
% known in the scope of the parent container.
%

/setname {      % name --
        dup dup length string cvs 0 get 64 ne {
                (@%) sprintf cvn
        } if
        @parent null eq {
                /setname super send
        } {
                @parent 1 index known {pop} {
                        /named? self send {
                                @parent /name self send undef
                        } if
                        dup /setname super send
                        @parent exch self soften put
                } ifelse
        } ifelse
} /installmethod ClassDrawable send

% overwrite the one from 'contain.psh'
/setcontainer { % container --
        @parent 1 index eq {pop} {
                @parent null ne {
                        /named? self send {
                                @parent /name self send undef
                        } if
                        @parent /Members 2 copy known {
                                2 copy get [exch {
                                        dup self eq {pop} if
                                } forall] put
                        } {pop pop} ifelse
                } if
                dup null eq {
                        pop /@parent unpromote
                } {
                        soften /@parent exch promote
                        /named? self send {
                                @parent /name self send known
                        } {true} ifelse {
                                1 1 10000 {
                                        /newnamestr self send sprintf cvn
                                        @parent 1 index known
                                        not {exit} if pop
                                } for
                                /setname super send
                        } if
                        @parent /name self send self soften put
                        @parent /Members
                        2 copy known {2 copy get} {//nullarray} ifelse
                        [self soften] append put
                        /invalidatecontainer self send
                } ifelse
        } ifelse
} /installmethod ClassDrawable send

/destroy {      % --
        @parent null ne {
                /named? self send {
```

```
                        @parent /name self send undef
                } if
                null /setcontainer self send
        } if
        /Members unpromote
        /Menu unpromote
        /destroy super send
} /installmethod ClassDrawable send

%
% These are the name templates used to generate unique names
%

ClassObject /newnamestr (@object%) put
ClassCanvas /newnamestr (@canvas%) put
ClassPanel /newnamestr (@panel%) put
ClassButtons /newnamestr (@buttons%) put
ClassSettings /newnamestr (@settings%) put
ClassHGauge /newnamestr (@gauge%) put
ClassLabel /newnamestr (@label%) put
ClassVScrollbar /newnamestr (@scrollbar%) put
ClassHSlider /newnamestr (@slider%) put
ClassScrollList /newnamestr (@list%) put
ClassFramebuffer /newnamestr (@framebuffer%) put
ClassWindow /newnamestr (@window%) put
ClassMenu /newnamestr (@menu%) put
ClassTextCanvas /newnamestr (@text%) put

%
% assign names to the frame buffers
%

globalroot /@parent null put
/@root /setname globalroot send
1 /children globalroot send {
        dup /@parent globalroot soften put
        1 index (@framebuffer%) sprintf cvn
        /setname 3 -1 roll send
        1 add
} forall pop
```

B.5 Addressing objects

```
%
% Find an object given an address. An address is either
% an object name or a path of object names.
%

/findobject {	% null|instance|name|path -- instance|null
	dup type /arraytype eq {
		dup 0 get /findobject self send dup null ne {
			exch 1 1 index length 1 sub getinterval {
				2 copy known not {
					pop pop null exit
				} if
				get
			} forall
		} {exch pop} ifelse
	} {
		dup null ne {
			dup type /nametype eq {
				self /@parent 2 index linkwhere {
					exch get
				} {pop null} ifelse
			} {
				dup isinstance? not {pop null} if
			} ifelse
		} if
	} ifelse
} /installmethod ClassDrawable send

/address {	% -- path
	@parent null ne {
		/address @parent send [/name self send] append
	} {
		[/name self send]
	} ifelse
} /installmethod ClassDrawable send

%
% Install @canvas generic name in ClassCanvas
%

ClassCanvas /ClassCanvas-destroy known not {
	ClassCanvas /ClassCanvas-destroy 1 index /destroy get put
} if
ClassCanvas /ClassCanvas-NewInit known not {
	ClassCanvas /ClassCanvas-NewInit 1 index /NewInit get put
} if

/NewInit {	% --
	/ClassCanvas-NewInit self send
	Parent /setcontainer self send
	/@canvas self soften promote
} /installmethod ClassCanvas send

/destroy {	% --
	/@canvas unpromote
	/ClassCanvas-destroy self send
} /installmethod ClassCanvas send

%
% Install @window generic name in ClassCanvas
%
```

```
ClassWindow /ClassWindow-destroy known not {
        ClassWindow /ClassWindow-destroy 1 index /destroy get put
} if
ClassWindow /ClassWindow-NewInit known not {
        ClassWindow /ClassWindow-NewInit 1 index /NewInit get put
} if

/NewInit {      % --
        /ClassWindow-NewInit self send
        Parent /setcontainer self send
        /@window self soften promote
} /installmethod ClassWindow send

/destroy {      % --
        /@window unpromote
        /ClassWindow-destroy self send
} /installmethod ClassWindow send

%
% Modify the /target method of ClassControl so that
% a target can be an adress.
%

/target {       % -- object
        Target findobject
} /installmethod ClassControl send
```

B.6 Sending messages

```
%
% Robust message sending. /findobject is used to
% establish a target, /deliver is used to deliver
% the message to the reciever.
% The /SendCTX field in the currentprocess dict is the
% operand stack of the previous context and contains
% the sender information.
%

/SendCTXget {   % n -- null|thing
        currentprocess /SendCTX 2 copy known {
                get dup length 3 -1 roll sub get
        } {pop pop pop null} ifelse
} /installmethod ClassDrawable send

/SendReciever   {2 SendCTXget} /installmethod ClassDrawable send
/SendSender     {3 SendCTXget} /installmethod ClassDrawable send
/SendTarget     {4 SendCTXget} /installmethod ClassDrawable send
/SendMessage    {5 SendCTXget} /installmethod ClassDrawable send
/SendArgs       {6 SendCTXget} /installmethod ClassDrawable send

/Send {         % args message target --
        /findobject self send dup null eq {pop pop pop} {
                self soften null
                currentprocess /SendCTX 2 copy known {get} {
                        pop pop 4 array
                } ifelse

                currentprocess /OperandStack get
                currentprocess /SendCTX 3 -1 roll put

                clear SendArgs aload pop SendMessage
                /deliver SendTarget send

                clear currentprocess /SendCTX get aload pop
                currentprocess /SendCTX 3 -1 roll put
                pop pop pop pop pop
        } ifelse
} /installmethod ClassDrawable send
```

B.7 Delivering messages

```
%
% Deliver a message to the first object in the
% container hierarchy which understands it.
%

/deliverself {  % args.. message --
        currentprocess /ErrorDetailLevel 1 put
        currentprocess /SendCTX 2 copy known {
                get dup length 2 sub self soften put
        } {pop pop} ifelse

        /log self send
        {cvx exec} stopped {
                /error self send
        } if
} /installmethod ClassDrawable send

/deliverparent {        % args.. message --
        @parent null ne {
                /deliver @parent send
        } {
                /log self send
        } ifelse
} /installmethod ClassDrawable send

/deliver {      % args... message --
        dup /understands? self send {
                /deliverself self send
        } {
                /deliverparent self send
        } ifelse
} /installmethod ClassDrawable send
```

B.8 Error handling

```
%
% Error handler
%

ClassDrawable /&reporterrors? true put
/reporterrors? {        % -- boolean
        self /@parent /&reporterrors? linkget
} /installmethod ClassDrawable send

/error {        % --
        reporterrors? {
                currentprocess /errordict known {
                        currentprocess /$error get /newerror get {
                                currentprocess /$error get begin
                                console (*** error ***\n) writestring
                                console message writestring
                                console flushfile
                                end
                        } if
                } if
        } if
} /installmethod ClassDrawable send
```

B.9 Message logging

```
%
% log message to the console
%

ClassDrawable /&log? true put
/log? { % -- boolean
	self /@parent /&log? linkget
} /installmethod ClassDrawable send

/logstr {	% -- str
	SendTarget SendReciever eq {
		SendSender SendReciever eq {
			SendReciever dup null ne {/namestr exch send} if
		} {
			(% to %) [
				SendSender dup null ne {/namestr exch send} if
				SendReciever dup null ne {/namestr exch send} if
			] sprintf
		} ifelse
	} {
		(% via % to %) [
			SendSender dup null ne {/namestr exch send} if
			SendTarget dup null ne {/namestr exch send} if
			SendReciever dup null ne {/namestr exch send} if
		] sprintf
	} ifelse
	(-- %\() [SendMessage] sprintf append
	0 1 SendArgs length 1 sub {
		dup 0 gt {exch (,) append exch} if
		SendArgs exch get 200 string cvs append
	} for
	(\)) append
} /installmethod ClassDrawable send

/log {		% --
	log? {
		console /logstr self send writestring
		console (\n) writestring
		console flushfile
	} if
} /installmethod ClassDrawable send
```

B.10 Control actions

```
%
% Change ExecuteNotifier in ClassControl so that it uses 'Send'
%

ClassControl /Notifier /Action put

ClassControl /ClassControl-ExecuteNotifier known not {
        ClassControl /ClassControl-ExecuteNotifier
        1 index /ExecuteNotifier get put
} if

/ExecuteNotifier {      % value notifier --
        dup null ne 1 index xcheck not and /target self send null eq and {
                exch [exch NotificationObject] exch  self /Send self
                % [value] notifier self /Send self
                5 array astore
                /sendmanager /eventmgr self send send
        } {
                /ClassControl-ExecuteNotifier self send
        } ifelse
} /installmethod ClassControl send
```

B.11 Resource manager

```
%
% A brain dead resource manager.
%
systemdict begin

/ClassResourceMgr ClassObject [] classbegin

/paths [
        (HOME) getenv (/rsrc) append
        (HOME) getenv (/cool/rsrc) append
        (COOLHOME) (/usr/local/cool) ?getenv (/rsrc) append
] def

/find { % rsrcname -- path file true | rsrcname false
        dup length string cvs paths (r) filepathopen
} def

classend def %ClassResourceMgr

/resourcemgr /new ClassResourceMgr send def

end %systemdict
```

B.12 Auto dictionaries

```
%
% A dictionary class that automatically undefs every
% key when the value becomes obsolete. Note that the
% dictionary also contains the /ParentDictArray key!
%
systemdict begin

/ClassAutoDict ClassObject [] classbegin
/HandleObsoleteTarget { % val --
        self {
                2 index eq {
                        self exch undef exit
                } if pop
        } forall pop
} def

/addkey {       % key val --
        soften
        dup self /addclient ObsoleteService send
        promote
} def
classend def %ClassAutoDict

end %systemdict
```

B.13 Plug-in classes

```
%
% A ResourceClass manager. The FIX to classbegin ensures that
% super class names can be used instead of super class instances.
%
systemdict begin

ClassObject /classfile null put

/ClassResourceClassMgr ClassObject [] classbegin
/classdict nulldict def
/rsrcmgr resourcemgr def

/FIX dictbegin
/classbegin {   % supers instancevars -- supers'  instancevars
        exch dup isarray? not {[exch]} if
        [exch {
                dup type /nametype eq {
                        /find classmgr send
                } if
        } forall]
        exch systemdict /classbegin get exec
} def
dictend def

/find {         % classname -- class|null
        classdict 1 index known {
                classdict exch get
        } {
                dup (%.class.ps) sprintf /find rsrcmgr send {
                        % classname path file
                        console (LOAD %\n) [4 index] fprintf
                        //FIX begin 2 index exch cvx exec end exch pop
                        % classname path class
                        dup /classfile 4 -1 roll put
                        dup 3 1 roll
                        /classdict promoted? not {
                                /classdict /new ClassAutoDict send promote
                        } if
                        /addkey /classdict self send send
                } {
                        pop systemdict 1 index known {
                                systemdict exch get exec
                        } {pop null} ifelse
                } ifelse
        } ifelse
} def

currentdict /FIX undef

classend def %ClassResourceClassMgr

/classmgr /new ClassResourceClassMgr send def

end %systemdict
```

B.14 Shared classes

```
%
% Implement script classes.
%

/scriptclass {  % -- class|null
        /script where not {null} if
} /installmethod ClassObject send

/baseclass {    % -- class
        /superclasses self send dup length 1 sub
        /scriptclass self send null ne {1 sub} if
        get
} /installmethod ClassObject send

/getscript {    % -- string|null
        /scriptclass self send dup null ne {/script get} if
} /installmethod ClassObject send

/findscriptclass { % script -- class
        % script
        self dup /scriptclassdict known not {
                dup /scriptclassdict /new ClassAutoDict send put
        } if

        % script baseclass
        dup /scriptclassdict get 2 index known {
                /scriptclassdict get exch get harden
        } {
                /visible? self send {
                        console (SCRIPT CLOSE\n) [] fprintf
                        /OnClose self send
                } if

                % script class
                /classname 1 index send (%-script) sprintf cvn 1 index []

                % script class class-name class []
                classbegin 1 index cvx exec classend exch pop

                % script class script-class
                dup /script 4 index put
                exch 3 copy /scriptclassdict get /addkey exch send
                pop exch pop

                /visible? self send {
                        console (SCRIPT OPEN\n) [] fprintf
                        /OnOpen self send
                } if
        } ifelse
} /installmethod ClassObject send

/setscript {    % script --
        /baseclass self send
        /findscriptclass exch send
        /ParentDictArray exch promote
} /installmethod ClassObject send
```

B.15 Creating default components

```
%
% Create an initialized object from scratch
%

/newdefault {   % parent -- instance
        /new self send
        0 0 100 50 /reshape 5 index send
} /installmethod ClassRegion self send

/newdefault {   % parent -- instance
        (Example Label) exch /new self send
} /installmethod ClassLabel send

/newdefault {   % parent -- instance
        /Spaced exch /new self send
        [(One) (Two) (Three)] /setitemlist 2 index send
        /Exclusive /setchoicemode 2 index send
} /installmethod ClassSettings send

/newdefault {   % parent -- instance
        /Spaced exch /new self send
        [(One) (Two) (Three)] /setitemlist 2 index send
        8 8 /setgaps 3 index send
        0 /setdefault 2 index send
} /installmethod ClassButtons send

/newdefault {   % parent -- instance
        /Spaced exch /new self send
        [[(File) /Grid framebuffer /new ClassMenu send
                [(Save) (Store) (Load)] /setitemlist 2 index send
                0 /setdefault 2 index send]
        [(Edit) /Grid framebuffer /new ClassMenu send
                [(Undo) (Copy) (Paste)] /setitemlist 2 index send
                1 /setdefault 2 index send]
        ] /setitemlist 2 index send
        8 8 /setgaps 3 index send
        1 /Right /setmenudirection 3 index send
} /installmethod ClassMenuButtons send

/newdefault {   % parent -- instance
        /new self send
        37 /setvalue 2 index send
} /installmethod ClassHGauge send

/newdefault {   % parent -- instance
        /new self send
        0 10 100 /setparameters 4 index send
        0 0 15 100 /reshape 5 index send
} /installmethod ClassVScrollbar send

/newdefault {   % parent -- instance
        /new self send
        0 10 100 /setparameters 4 index send
        0 0 100 15 /reshape 5 index send
} /installmethod ClassHScrollbar send

/newdefault {   % parent -- instance
        /new self send
        37 /setvalue 2 index send
        20 /settickmarks 2 index send
} /installmethod ClassHSlider send
```

```
/newdefault {   % parent -- instance
        /Spaced exch /new self send
        /Center 1 index /newdefault ClassButtons send /addclient 3 index send
} /installmethod ClassPanel send

/newdefault {   % parent -- instance
        /new self send
        (Text field) /setvalue 2 index send
        10 /setminimumvisible 2 index send
} /installmethod ClassTextField send

/newdefault {   % parent -- instance
        /new self send
        37 /setvalue 2 index send
        10 /setminimumvisible 2 index send
} /installmethod ClassNumericField send

/newdefault {   % parent -- instance
        /new self send
        /Exclusive /setchoicemode 2 index send
        [(Monday) (Tuesday) (Wednesday) (Thursday)
         (Friday) (Saturday) (Sunday)]
        /setitemlist 2 index send
        0 0 100 125 /reshape 5 index send
} /installmethod ClassScrollList send
```

B.16 Persistent variables

```
%
% Mark persistent variables. It is done explicitly like this
% since it would otherwise require a fix in classbegin
%

% clear existing persistency information
{ % proc class --
        dup /persistent //nullarray put
        /subclasses exch send {2 copy 1 index exec pop} forall pop
} ClassObject 1 index exec

systemdict /systemdict-classbegin known not {
        systemdict /systemdict-classbegin systemdict /classbegin get put
} if
systemdict /classbegin {        % ... - ...
        systemdict-classbegin
        /superclasses self send {
                /persistent exch send {persist} forall
        } forall
} put

/persist {      % name|list --
        dup isarray? not {[exch]} if
        [exch {
                /persistent self send 1 index arraycontains? {pop} if
        } forall] dup length 0 gt {
                /persistent self send 1 index append /persistent exch store
                /subclasses self send {
                        2 copy /persist exch send pop
                } forall pop
        } {pop} ifelse
} /installmethod ClassObject send

%
% Mark the persistent instance variables of the TNT classes
%

[
        /ObjectName /Properties
] /persist ClassObject send

[
        /Transparent /VisualState
        /&BG0 /&BG /&BG2 /&BG3 /&FG /&2DBBG /&2DFG
        /&TextFont /GraphicFont /HelpKeyWord
        /Menu /&3D?
        /minsize /preferredsize
        /TNTLayout /Mapped
] /persist ClassDrawable send

[
        /BorderStroke
        /Freezable? /Frontable? /Helpable? /Keyable?
        /Menuable? /Openable? /Receptible? /Selectable?
        /Trackable?
        /DropPreviewLevel /SaveBehind /Retained /EventsConsumed
        /ForkFix? /SelectableType /@canvas
] /persist ClassCanvas send

[
        /X /Y /Width /Height
] /persist ClassRegion send
```

```
[
        /FocusNoticeable? /FocusForwardable?
        /ClientDict /RegionClientList
] /persist ClassBag send

[
        /NGap /EGap /SGap /WGap /NInset /EInset /SInset /WInset
] /persist ClassBorderBag send

[
        /&DragFrame? /FooterFont /Freezable? /BorderEdge
        /ReshapeSize /BorderStroke
        /HeaderGap /FooterGap /FooterPad /HeaderPad
        /FooterFraction /HeaderHeight /FooterHeight
        /HeaderDeltaY /Close? /Footer? /Label? /Pin? /Opened?
        /Reshape? /Label /LeftFooter /RightFooter
        /@window
] /persist ClassWindow send

[
        /IconLabel /IconImage /IconFont
] /persist ClassBaseWindow send

[
] /persist ClassPopupWindow send

[
        /Placement /LayoutParameters /Columns
        /Rows /VerticalGap /HorizontalGap
] /persist ClassLayout send

[
        /Value /Notifier /Previewer
] /persist ClassControl send

[
        /Contents
] /persist ClassPanel send

[
        /ItemList /FixedItemSize?
] /persist ClassItemGroup send

[
        /ItemWidth /ItemHeight /Abbreviated?
        /Justification /Default /buttonfont
] /persist ClassButtons send

[
] /persist ClassMenuButtons send

[
        /Justification /ChoiceMode /ItemWidth /ItemHeight
] /persist ClassSettings send

[
        /Label
] /persist ClassLabel send

[
        /ChoiceMode /VisualState /BaseMenu? /PinnedLabel
        /Label /Pinnable? /Justification /LayoutArgCount
] /persist ClassMenu send
```

```
[
] /persist ClassMenuHeader send

[
	/Default /LayoutByRows? /Columns /Rows /CellSize
] /persist ClassCommandMenu send

[
	/LayoutByRows? /Columns /Rows /CellSize
] /persist ClassSettingsMenu send

[
] /persist ClassPinnedMenu send

[
	/Top /Bottom /Width /Height
	/ScrollMinimum /ItemList /ItemWidth /ItemHeight
	/Value /ChoiceMode
] /persist ClassScrollList send

[
	/Low /High /Max /ScrollDelay /ScrollThresh
] /persist ClassVScrollbar send

[
	/Value /Min /Max /EndBoxes? /TickMarks /Delta
	/Normalize /SliderDelay /SliderThresh
] /persist ClassHSlider send

[
	/Value /Min /Max /TickMarks
] /persist ClassHGauge send

[
	/ReadOnly? /MinimumVisible /ScrollThresh /ScrollDelay
	/Value /CaretDelay
] /persist ClassTextField send

[
	/Maximum /Minimum /Delta /Normalizer
] /persist ClassNumericField send
```

B.17 Cloning components

```
%
% Create a clone of an object
%
/REFDICT growabledict def

/CLONEDICT dictbegin
countedtypes {{pop null} def} forall
uncountedtypes {{} def} forall
/stringtype {   % string -- 'string
        dup length string copy
} def
/arraytype {    % array -- 'array
        [ 1 index {clonething} forall] exch xcheck {cvx} if
} def
/packedarraytype /arraytype load def
/dicttype {     % dict -- 'dict
        //REFDICT 1 index known {
                //REFDICT 1 index get
                exch soft {soften} {harden} ifelse
        } {
                dup isobject? {
                        dup isinstance? {
                                growabledict
                                //REFDICT 2 index 2 index put
                                /cloneinstance 2 index send
                                exch soft {soften} if
                        } {pop null} ifelse
                } {
                        dup maxlength dict
                        //REFDICT 2 index 2 index put
                        1 index {
                                % dict 'dict key val --
                                exch clonething exch clonething
                                3 copy put pop pop
                        } forall
                        exch soft {soften} if
                } ifelse
        } ifelse
} def
/canvastype {   % canvas -- 'canvas
        //REFDICT 1 index known {
                //REFDICT 1 index get
                exch soft {soften} {harden} ifelse
        } {
                dup isinstance? {
                        nullcanvas newcanvas
                        //REFDICT 2 index 2 index put
                        /cloneinstance 2 index send
                        exch soft {soften} if
                } {
                        dup /Retained get not
                        1 index /Parent get null ne or {
                                pop null
                        } if
                } ifelse
        } ifelse
} def
dictend def     %CLONEDICT

/cloneinstance { % clone -- 'clone
        dup /ParentDictArray ParentDictArray put
        /persistent self send {
```

```
                % 'self key
                dup promoted? {
                        2 copy self 1 index get /clonething self send put
                } if
                pop
        } forall
} /installmethod ClassObject send

/cloneinstance {        % canvas -- 'canvas
        /cloneinstance super send
        dup /BBOX [/bbox self send] cvx put
} /installmethod ClassCanvas send

/cloneinstance {        % clone -- 'clone
        /cloneinstance super send
        dup /children
        /children self send /clonething self send put
} /installmethod ClassBag send

/clonething {   % any -- 'any
        //CLONEDICT 1 index truetype get cvx exec
} /installmethod ClassObject send

/clone {        % -- clone
        self harden /clonething self send
        //REFDICT cleanoutdict
} /installmethod ClassObject send

currentdict /CLONEDICT undef
currentdict /REFDICT undef
```

B.18 Renewing components

```
%
% Renew a component from data.
%
/renew {	% parent --
	pop InstanceVars {
		1 index promoted? not {promote} {pop pop} ifelse
	} forall
} /installmethod ClassObject send

/renew {	% parent --
	dup /renew super send
	/reparent self send
	/Menu promoted? {
		self /renew Menu send
	} if
} /installmethod ClassDrawable send

/renew {	% parent --
	dup /Mapped get Mapped and {
		/Mapped false promote
	} if
	/renew super send
	gsave Parent setcanvas BBOX /reshape self send grestore
	/BBOX unpromote
	Selectable? {Holder null eq {/Holder self soften promote} if} if
} /installmethod ClassCanvas send

/renew {	% parent --
	/renew super send
	/regionclientlist self send {
		self /renew 3 -1 roll send
	} forall
	/children self send {
		self /renew 3 -1 roll send
	} forall
	/children unpromote
} /installmethod ClassBag send

/renew {	% parent --
	pop framebuffer /renew super send
} /installmethod ClassMenu send

/renew {	% parent --
	/renew super send
	0 1 /itemcount self send 1 sub {
		/menu self send
		self /renew 3 -1 roll send
	} for
} /installmethod ClassMenuButtons send

/renew {	% parent --
	/renew super send
	0 1 /itemcount self send 1 sub {
		/Item self send dup /SubMenu known {
			/SubMenu get self /renew 3 -1 roll send
		} {pop} ifelse
	} for
} /installmethod ClassCommandMenu send

/renew {	% parent --
	/renew super send
	0 1 /itemcount self send 1 sub {
```

```
                /Item self send dup /SubMenu known {
                        /SubMenu get self /renew 3 -1 roll send
                } {pop} ifelse
        } for
} /installmethod ClassSettingsMenu send

/renewvalue {
        Value /setvalue self send
} /installmethod ClassTextField send

/renew {
        /renew super send
        /Caret /new ClassForCaret send promote
        /CheckModifiable self send
        /PendingDelete? dictbegin
                InterestingRankDict {pop false def} forall
        dictend def
        /renewvalue self send
} /installmethod ClassTextField send

/renewvalue {
        /value self send /setvalue self send
        /invalidate self send
} /installmethod ClassNumericField send
```

B.19 Saving components

```
%
% Save an object to a file
%
/REFDICT growabledict def

/SAVEDICT dictbegin
uncountedtypes {{writeobject} def} forall
countedtypes {{pop null writeobject} def} forall
/stringtype {   % file string --
        2 copy writeobject
        length 32 gt {(\n) writestring} {pop} ifelse
} def
/realtype {     % file real --
        dup dup cvi eq {cvi} if writeobject
} def
/fixedtype //realtype def
/marktype {pop ([) writestring} def
/colortype {    % file color --
        % file r g b --
        colorrgb
        3 index 4 -1 roll savething 2 index 32 write
        2 index 3 -1 roll savething 1 index 32 write
        1 index exch savething dup 32 write
        ( rgbcolor) writestring
} def
/operatortype { % file op --
        128 string cvs 1 1 index length 2 sub getinterval writestring
} def
/fonttype {     % file font --
        2 copy /FontName get writeobject
        1 index ( findfont ) writestring
        2 copy /FontMatrix get savething
        1 index ( makefont ) writestring
        2 copy /PrinterMatched get writeobject
        pop ( printermatchfont) writestring
} def
/arraytype {    % file array --
        2 copy xcheck {({)} {([)} ifelse writestring
        0 1 2 index length 1 sub {
                dup 0 ne {
                        dup 6 mod 0 eq {
                                2 index (\n) writestring
                        } {
                                2 index 32 write
                        } ifelse
                } if
                1 index exch get 2 index exch savething
        } for
        xcheck {(})} {(])} ifelse writestring
} def
/packedarraytype //arraytype def
/dicttype {     % file dict --
        //REFDICT 1 index known {
                2 copy //REFDICT exch get writeobject
                1 index ( GETREF) writestring
                soft {( soften) writestring} {pop} ifelse
        } {
                dup isclass? {
                        dup /script known {
                                2 copy /superclasses exch send
                                dup length 1 sub get savething
                                //REFDICT 2 copy /refcounter get put
```

```
                                //REFDICT /refcounter 2 copy get 1 add put

                                1 index 32 write
                                /script get 1 index exch writeobject
                                ( SCRIPT) writestring
                        } {
                                /classname exch send
                                1 index exch writeobject
                                ( CLASS) writestring
                        } ifelse
                } {
                        dup isinstance? {
                                2 copy /class exch send savething
                                //REFDICT 2 copy /refcounter get put
                                //REFDICT /refcounter 2 copy get 1 add put
                                1 index ( BEGININSTANCE\n) writestring
                                2 copy /saveinstance exch send
                                1 index (ENDINSTANCE) writestring
                        } {
                                //REFDICT 2 copy /refcounter get put
                                //REFDICT /refcounter 2 copy get 1 add put
                                2 copy maxlength writeobject
                                1 index ( BEGINDICT\n) writestring
                                2 copy {
                                        % file key val
                                        3 copy pop savething exch pop
                                        1 index 32 write
                                        1 index exch savething
                                        dup ( def\n) writestring
                                } forall
                                (ENDDICT) writestring
                        } ifelse
                        soft {( soften) writestring} {pop} ifelse
                } ifelse
        } ifelse
} def
/canvastype {   % file canvas --
        //REFDICT 1 index known {
                2 copy //REFDICT exch get writeobject
                1 index ( GETREF) writestring
                soft {( soften) writestring} {pop} ifelse
        } {
                dup isinstance? {
                        2 copy /class exch send savething

                        //REFDICT 2 copy /refcounter get put
                        //REFDICT /refcounter 2 copy get 1 add put

                        1 index ( BEGINCANVAS\n) writestring
                        2 copy /saveinstance exch send
                        1 index (ENDCANVAS) writestring
                        soft {( soften) writestring} {pop} ifelse
                } {
                        pop null writeobject
                } ifelse
        } ifelse
} def
dictend def     %SAVEDICT

/savething {    % file val --
        //SAVEDICT 1 index truetype get exec
} /installmethod ClassDrawable send

/saveobject {           % file --
```

```
        dup (0.01 BEGINSAVE\n) writestring
        //REFDICT /refcounter 0 put
        dup self harden /savething self send
        //REFDICT cleanoutdict
        (\nENDSAVE\n) writestring
} /installmethod ClassDrawable send

/saveinstance { % file --
        /persistent self send {
                dup promoted? {
                        2 copy writeobject
                        1 index 32 write
                        self exch get 1 index exch savething
                        dup ( def\n) writestring
                } {pop} ifelse
        } forall pop
} /installmethod ClassDrawable send

/saveinstance { % file --
        dup /saveinstance super send
        dup (/BBOX ) writestring
        dup [gsave Parent setcanvas /bbox self send grestore] cvx savething
        ( def\n) writestring
} /installmethod ClassCanvas send

/saveinstance { % file --
        dup /saveinstance super send
        dup /children writeobject
        dup /children self send savething
        ( def\n) writestring
} /installmethod ClassBag send

currentdict /SAVEDICT undef
currentdict /REFDICT undef
```

B.20 Loading components

```
%
% Load from file
%
/REFDICT growabledict def

/LOADDICT dictbegin
/BEGINSAVE {    % version --
        pop
} def
/ENDSAVE nullproc def

/CLASS {        % name -- class
        /find classmgr send
} def
/SCRIPT {       % class string -- class
        /findscriptclass 3 -1 roll send
        //REFDICT dup /refcounter get currentdict put
        //REFDICT /refcounter 2 copy get 1 add put
} def

/GETREF {       % nr -- thing
        //REFDICT exch get
} def

/BEGINDICT {            % maxlength --
        dict begin
        //REFDICT dup /refcounter get currentdict put
        //REFDICT /refcounter 2 copy get 1 add put
} def
/ENDDICT {              % -- dict
        currentdict end
} def

/BEGININSTANCE {        % class --
        growabledict begin
        //REFDICT dup /refcounter get currentdict put
        //REFDICT /refcounter 2 copy get 1 add put
        /ParentDictArray exch def
} def
/ENDINSTANCE {          % -- instance
        currentdict end
} def

/BEGINCANVAS {          % class --
        nullcanvas newcanvas begin
        //REFDICT dup /refcounter get currentdict put
        //REFDICT /refcounter 2 copy get 1 add put
        /ParentDictArray exch def
} def
/ENDCANVAS {            % -- canvas
        currentdict end
} def
dictend def %LOADDICT

systemdict begin
/loadobject {           % file -- data
        //LOADDICT begin
        //REFDICT /refcounter 0 put
        cvx exec
        //REFDICT cleanoutdict
        end
} def
```

```
end %systemdict

currentdict /REFDICT undef
currentdict /LOADDICT undef
```

C Examples

This section contains some small example programs to illustrate how Cool can be used.

C.1 Scoped variables

The `/set3D` method is used to change the scoped variable that controls whether an interface is 3D or 2D. Setting this variable in the window changes the whole UI.

```
%
% Demonstrate parent scope
%
(cool/src/init.psh) run

/win null framebuffer /new ClassBaseWindow send def
/settings /Spaced win /new ClassSettings send def

% create the settings
/Exclusive /setchoicemode settings send
[ (Black and white) (Three Dimensional) ] /setitemlist settings send
[/3D? win send {1} {0} ifelse] /setvalue settings send
8 8 /setgaps settings send

% set the script of the window
(/Action {      % item --
        pop 0 get 1 eq set3D /paint self send
} def) /setscript win send

% Add the settings to the window
/Center settings /addclient win send
(Scoped variables Demo) /setlabel win send
20 10 20 10 /setgaps win send
/place win send
/new ClassEventMgr send /activate win send
/map win send

newprocessgroup currentfile closefile
```

C.2 Scripts

The `/setscript` method can be used to change the functionality of an instance.

```
(cool/src/init.psh) run

/demo ClassButtons send
/b exch def
/w exch def

w /sillyfooter undef
(/sillyfooter { % index buttons => -
        /item exch send
        (The % is extremely intresting...) sprintf ()
        /setfooter self send
} def) /setscript w send

newprocessgroup currentfile closefile
```

C.3 Send

Demonstrate how the `Send` primitive can be used to communicate between components. Two windows are created. The first contains a scrollbar which controls a gauge in the second.

```
(cool/src/init.psh) run
%
% Demonstrate the use of 'Send'
%

% Create window 1
/win1 null framebuffer /new /ClassBaseWindow /find classmgr send send def
(Send demo) /setlabel win1 send
20 10 20 10 /setgaps win1 send
(/Toggle {      % item --
        pop [] /ToggleOpened /@play Send
} def
/Action {       % value --
        pop [exch cvi] /setvalue [/@play /@temp] Send
} def) /setscript win1 send

% Create scrollbar
/sb win1 /new /ClassHScrollbar /find classmgr send send def
0 10 100 /setparameters sb send
/Action /setpreviewer sb send
/South sb /addclient win1 send

% create the buttons
/buttons /Spaced win1 /new /ClassButtons /find classmgr send send def
[[(toggle) /Toggle] [(Quit) /QuitFromUser]]
/setitemlist buttons send
8 8 /setgaps buttons send
/Center buttons /addclient win1 send

% Create window 2
/win2 null framebuffer /new /ClassBaseWindow /find classmgr send send def
(win2) /setlabel win2 send
20 20 20 20 /setgaps win2 send
/@play /setname win2 send

% create gauge
/g framebuffer /new /ClassHGauge /find classmgr send send def
/@temp /setname g send
0 100 /setrange g send
0 /setvalue g send
/Center g /addclient win2 send
0 0 40 40 /reshape g send

% Initialize the windows
/place win1 send
/location win1 send
0 -105 xyadd 300 100 /reshape win2 send
/new ClassEventMgr send
dup /activate win1 send
/activate win2 send
/map win1 send
/map win2 send

newprocessgroup currentfile closefile
```

C.4 Buttons demo

Implement the demo method of `ClassButtons` using the techniques provided by Cool. Makes use of the class manager and script classes.

```
%
% The COOL ClassButtons demo
%
(cool/src/init.psh) run

/win null framebuffer /new /ClassBaseWindow /find classmgr send send def
/buttons /Spaced win /new /ClassButtons /find classmgr send send def

% Initialize the buttons
[ (Weather) (Sport) (Gossip)
  [(NeWS) /NeWSAction]
  (Entertainment) (Finance) (Happy Talk)
] /setitemlist buttons send
3 /setdefault buttons send
8 8 /setgaps buttons send
/preferredsize {278 50} /installmethod buttons send
/minsize {194 76} /installmethod buttons send

% Define the actions in the window's script
(/Action {      % index --
        /item exch send
        (The % is extremely boring...) sprintf ()
        /setfooter self send
} def
/NeWSAction {   % index --
        pop {5 {beep .0005 sleep} repeat} fork pop
} def) /setscript win send

% Add the buttons to the window and Go!
/Center buttons /addclient win send
(ClassButtons Demo) /setlabel win send
() (This uses the /Spaced layout style -- reshape it!) /setfooter win send
20 10 20 10 /setgaps win send
/place win send
/new /ClassEventMgr /find classmgr send send /activate win send
/map win send

% clean exit
newprocessgroup currentfile closefile
```

C.5 Scrollbar demo

Implement the demo method of `ClassVScrollbar`. The original demo method makes use of a subclass of `ClassCanvas` called `SBDemoCanvas`.

```
executive
(cool/src/init.psh) run
%
% use the class manager to load SBDemoCanvas.class.ps
%
/sb framebuffer /new /ClassVScrollbar /find classmgr send send def
/can framebuffer /new /SBDemoCanvas /find classmgr send send def

/@sbdemocanvas /setname can send

0 10 101 /setparameters sb send

/@sbdemocanvas /settarget sb send
/updatedemo /setnotifier sb send
/updatedemo /setpreviewer sb send

37 dup /setvalue sb send
null /updatedemo can send

/win can framebuffer /new ClassBaseWindow send def
/East sb /addclient win send
/Footer? false /setattribute win send
(ClassVScrollbar) /setlabel win send
/place win send
/new ClassEventMgr send /activate win send
/map win send

% clean exit
newprocessgroup currentfile closefile
```

It uses the resource class `SBDemoCanvas`. This stored in the file *SBDemoCanvas.class.ps*:

```
/ClassCanvas
dictbegin
    /String () def
dictend
classbegin
    /TextFont   /ZapfChancery-MediumItalic findfont
                100 scalefont def
    /FG         0 0 1 rgbcolor def

    /Paint {
        BackgroundColor setcolor /bbox self send rectpath fill
        TextFont setfont
        FG setcolor ShowString
    } def

    /updatedemo { % value sbar => -
        gsave
            pop self setcanvas
            TextFont setfont
            BackgroundColor setcolor ShowString
            cvi 3 string cvs /String exch def
            FG setcolor ShowString
        grestore
    } def

    /ShowString {
        size self send 2 div exch 2 div exch 25 sub moveto
        String cshow
```

```
    } def

    /preferredsize {150 125} def
classend
```

C.6 Newdefault demo

Create a window containing all the default UI components.

```
(cool/src/init.psh) run
%
% Create a window containig all default components
%

/panel /Grid framebuffer /new ClassPanel send def
10 10 /setgaps panel send
[true 4 4] /setlayoutparameters panel send
[
        ClassRegion ClassLabel ClassSettings ClassButtons
        ClassMenuButtons
        ClassPanel
        ClassTextField ClassNumericField
        ClassHSlider ClassHGauge ClassHScrollbar
        ClassVSlider ClassVGauge ClassVScrollbar
        ClassScrollList
] {
        framebuffer /newdefault 3 -1 roll send
        dup /addclient panel send
} forall

/win panel framebuffer /new ClassBaseWindow send def
(newdefault demo) /setlabel win send
20 20 20 20 /setgaps win send
/place win send
/new ClassEventMgr send /activate win send
/map win send

newprocessgroup currentfile closefile
```

C.7 Clone demo

Create a window which can clone itself. The window contains 2 buttons, one to create a clone the other quits the application. The window also contains a default scrolling list.

```
%
% Create a window which can clone itself
%
(cool/src/init.psh) run

/b /Spaced framebuffer /new ClassButtons send def
10 10 /setgaps b send
[
        [(Duplicate) /Duplicate]
        [(Quit) /QuitFromUser]
] /setitemlist b send

/w b framebuffer /new ClassBaseWindow send def
/South w /newdefault ClassScrollList send /addclient w send

(/Duplicate {
        /clone self send
        framebuffer /renew 2 index send
        /eventmgr self send  /activate 2 index send
        /location self send 20 -20 xyadd /move 3 index send
        /map 1 index send
        userdict begin dup def end
        /footer self send pop (cloned) /setfooter self send
} def
/Action {
        exch 0 get /item 3 -1 roll send
         (I hate %s) sprintf () /setfooter self send
} def) /setscript w send

(Clone demo) /setlabel w send
10 10 10 10 /setgaps w send
/place w send
/new ClassEventMgr send /activate w send
/map w send

newprocessgroup currentfile closefile
```

C.8 Save and load demo

Create a window which can save and reload itself. The window contains 3 buttons, one to save a copy of itself in `~/window.rsrc`, one to load a copy from the same file and one to quit the application. The window also contains a settings object to select the window color.

```
%
% Create a window that can save and load from a file.
%
(cool/src/init.psh) run

/b /Spaced framebuffer /new ClassButtons send def
10 10 /setgaps b send
[
        [(Save) /Save]
        [(Load) /Load]
        [(Quit) /QuitFromUser]
] /setitemlist b send

/w b framebuffer /new ClassBaseWindow send def
/South w /newdefault ClassSettings send /addclient w send

(/Save {
        () () /setfooter self send
        (window.rsrc) (w) file /saveobject self send
        (Saved) (window.rsrc) /setfooter self send
} def
/Load {
        () () /setfooter self send
        {(window.rsrc) (r) file} stopped not {
                /loadobject self send
                framebuffer /renew 2 index send
                /eventmgr self send  /activate 2 index send
                /location self send 20 -20 xyadd /move 3 index send
                /map 1 index send
                userdict begin dup def end
                (Loaded) (window.rsrc) /setfooter self send
        } if
} def
/Action {
        () () /setfooter self send
        pop 0 get {
                0 {1 .8 .8 rgbcolor}
                1 {.8 1 .8 rgbcolor}
                /Default {.8 .8 1 rgbcolor}
        } case
        0 0 0 rgbcolor exch /setcolors self send
        /paint self send
} def) /setscript w send

(Save/Load demo) /setlabel w send
10 10 10 10 /setgaps w send
/place w send
/new ClassEventMgr send /activate w send
/map w send

newprocessgroup currentfile closefile
```