Tales From the View Sytem

Michael S. Engber
Apple Computer, PIE Technical Support
Copyright © 1993 - Michael S. Engber

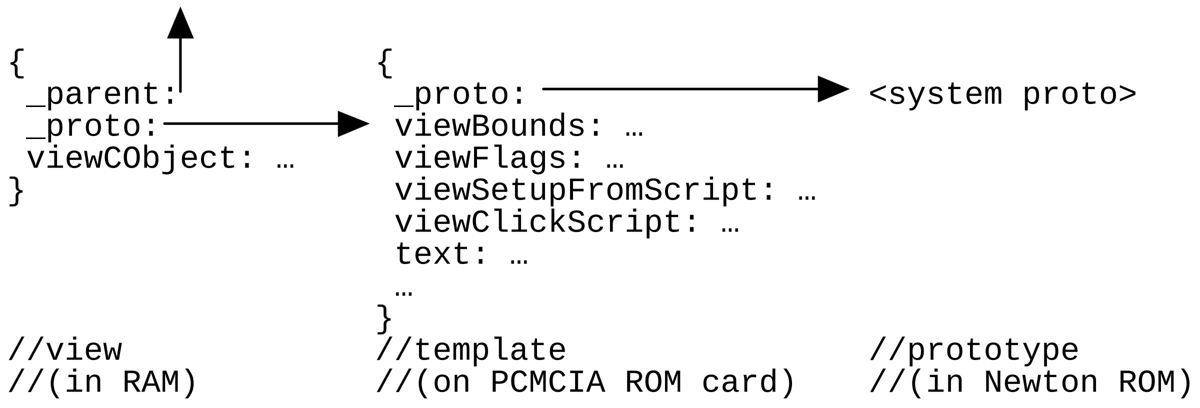
This article was published in the November 1993 issue of PIE Developers magazine. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@APPLELINK.APPLE.COM or 415.621.4252.

This article presents details about the Newton's view system that are often overlooked or misun-derstood. You can write significant Newton applications without knowing all these details (largely due to NTK's ease of use), but ultimately, you need to understand many of the topics presented here.

This is not an introductory article. It's designed to be read on your second or third pass through understanding the Newton's view system. Don't expect to understand everything it contains the first time you read it. This article assumes that the reader knows the basics of NewtonScript (especially proto and parent inheritance) and has some experience using NTK to write Newton applications.

Templates vs. Views

Views are not created by NTK. In NTK you lay out templates. Views are frames created in RAM by the Newton's view system. Views use templates as their prototypes. The difference between views and templates seems obvious enough, but in practice, it's easy to get confused. Figure 1 illustrates the relationship between a typical view, template, and prototype.



```
{                   {
 _parent:            _proto:            <system proto>
 _proto:             viewBounds: …
 viewCObject: …      viewFlags: …
}                    viewSetupFromScript: …
                     viewClickScript: …
                     text: …
                     …
                    }
//view              //template                //prototype
//(in RAM)          //(on PCMCIA ROM card)    //(in Newton ROM)
```

Figure 1 - A typical view / template / proto relationship.

Note several things about Figure 1:

- The view has very few slots, just _parent, _proto, and viewCObject. Views inherit most of the slots and methods they need. This lets views remain small, which is important since they take up space in RAM. Note: if the view has declared children, it would contain a slot for each declared child. Details regarding these slots are discussed later in this article.
- Figure 1 shows why the slot assignment rules work the way they do. Recall, if you change the value of a slot inherited from the _proto chain, a new slot is created in the frame to hold the new value (rather than modifying the inherited slot). If we look at the figure, we can see that only the view is in RAM. Modifying the inherited slot is not even possible. So, even though a view frame starts out very small, it grows over time as slots get modified. This growth is usually very small, since in practice, views seldom ever modify more than one or two slots.

- The template does not have a _parent slot.This might be surprising since NTK displays templates using a parent-child hierarchy. NTK actually uses the hierarchy which exists among the view objects created from the templates.

Table 1 on the next page is a quiz to test your understanding of the distinction between views and templates.

		template	view
1)	created by NTK		
2)	only reside in RAM		
3)	handles `:Hide()`, `:Open(), and :Close()`messages		
4)	reside in stepChildren and viewChildren arrays		
5)	code modifies their slots at runtime - preferably, using `SetValue`		
6)	use system protos in their proto slots		
7)	passed to AddStepView		
8)	usually contains few slots (inherits, most of its slots)		
9)	usually reside in ROM		
10)	returned by `:ChildViewFrames()`		
11)	returned by Debug()		
12)	is a frame		
13)	has a parent		

Table 1 - A template vs. view quiz.

Templates vs. Prototypes
The distinction between templates and prototypes is a bit subtle. I could even argue that, technically, there is no difference. However, it’s useful to draw the distinction, especially since NTK enforces this distinction. The things you drag out in NTK are templates. The NTK palette that you make selections from contains prototypes (well, viewClasses too, but we won’t go into that now). In general, templates contain prototypes in their _proto slot.

viewChildren and stepChildren
The viewChildren and stepChildren slots both contain template arrays. These arrays are used to create a view’s children when the parent view is first created. In general, you use stepChildren for children you define, and the system uses viewChildren for its children. For example, if you create a template based on ProtoFloatNGo, the close box is provided by the system and its template is in the viewChildren array. The templates you add are put into the stepChildren array by NTK. Normally, you need not concern yourself with these details, unless you’re explicitly defining a template as a NewtonScript frame rather than dragging it out graphically from the palette.

Always remember: “viewChildren and stepChildren contain templates, not views.” Dereferencing some unsuspecting element of your stepChildren or viewChildren array and sending it a Hide message does absolutely no good. It generates an error. If you need access to the views that are created from your viewChildren and stepChildren templates, use the ChildViewFrames message. It returns an array of all the views created from *both* the viewChildren and stepChildren. The viewChildren views are first, followed by the stepChildren views in sibling order (the order they show up in NTK browser). In the FloatNGo example described earlier, ChildViewFrames returns an array whose first element is a close-box view, followed by all the views for the items you define.

Why viewChildren and stepChildren?
If there is only one slot, say the viewChildren slot, then your template’s viewChildren slot overrides the viewChildren slot of your template’s proto. Returning to the ProtoFloatNGo example, the close box would be missing. When the view system instantiates the view, it only sees the items you define and in this case misses

the close box.

This problem can be resolved by searching the entire proto chain for viewChildren slots when creating a view. But this is inefficient. Instead, a parallel slot is used. It’s unfortunate that the term stepChildren makes the stepChildren slot sound inferior in some way. The only difference between stepChildren and viewChildren is that the views created from viewChildren are first in the ChildViewFrames array.

Note: At first glance this may appear to be a problem for user protos–you might expect the stepChildren slot in your template to override the stepChildren slot in the userProto it’s based on. This isn’t a problem because NTK combines the stepChildren of the entire proto chain when it compiles your template. This process doesn’t occur until you build your package, so you’re normally unaware of it. If you look at your view’s stepChildren slot in the inspector, however, you will see all your template’s children plus any children defined in its proto.

It may seem possible to combine all the stepChildren and viewChildren at compile time and use one slot to reference all of them. This is not possible–the viewChildren are defined in templates in the Newton ROM and aren’t available at compile time.

Declaring Views

Declaring a view allows you to access it symbolically. For example, if you declare a view SomeDeclaredView then you can write code like `SomeDeclaredView:Open()` to display it. In NTK you declare views using the Template InfoÉ command. Both the view being declared and the view to which it’s being declared must be named. You select the view you want to declare and execute the Template InfoÉ command (Special menu). Figure 2 shows the dialog which appears. You name the view using the edit text field, and use the Allow Access From checkbox and popup to declare the view. Only named parent views appear in the pop up.

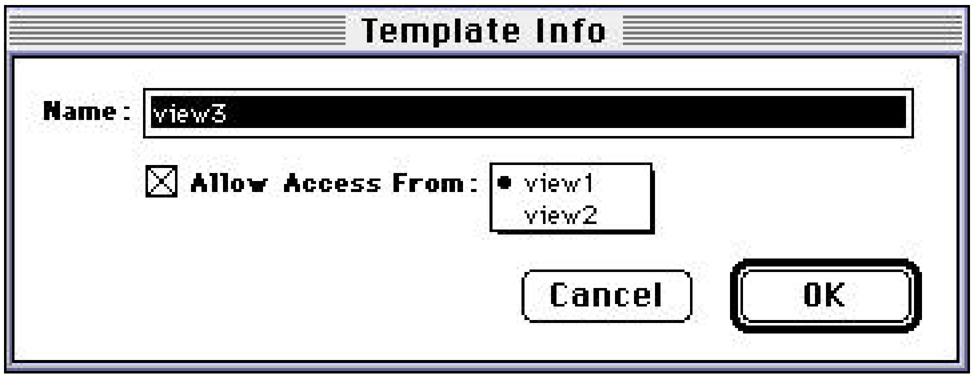


Figure 2 - The NTK’s Template Info dialog.

What Declaring Really Does

Declaring a view does several things. First, it creates a slot in the parent view. The slot’s name is the name of the declared view and the slot’s value points to the new view. Because of this it’s a bad idea to pick view names like _proto or viewClickScript. Deciding where to declare a view amounts to deciding where you want that view’s slot to go.

Remember, NTK only deals with templates. Views are declared. NTK sets up the templates with the correct information (like their names) needed for the declarations. The view system actually takes care of the work when the views are created. The details of this process are discussed in more detail later in this article.

Where to Declare a View

Consider the view hierarchy shown in Figure 3. If we declare view3 in view2, then scripts in view2, view3, and view4 have access to view3 (since the slot is in view2). View3 and view4 get access through parent inheritance (scripts in view3 could also use self)–view1 is left out in the cold. If we declare view3 to view1, instead, everyone has access. Alternately, if view2 is declared to view1 then view1 can access view3 using `view2.view3`.

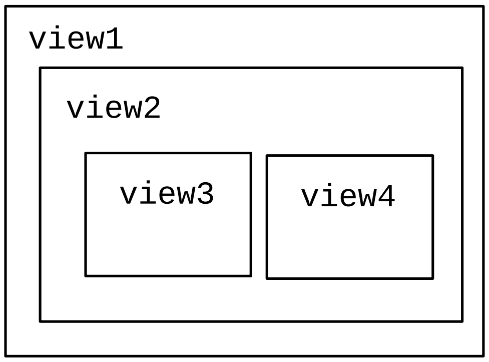


Figure 3 - An example view hierarchy.

You should declare a view to a common ancestor of all the views that need access to it. You can simply declare everyone to your application's base view (everyone's ancestor), but you may not want to do that because of potential name conflicts, efficiency considerations (to be discussed later), and most importantly, because it entirely eliminates the challenge of deciding where to declare a view.

Why is Declaring Necessary
On the surface, declare seems just like a convenient way to create a view slot containing self. Couldn't we achieve the same effect by simply having each view we want declared do something like:

```
:Parent().myName := self;
```

in its viewSetupDoneScript? In addition to allowing symbolic access to a view, declaring a view also ensures that it is preallocated (the view frame consisting of the _parent, _proto, and viewCObject is made ahead of time). This means that declare has some overhead associated with it–not a lot, but you should only declare the views that you need to access.

Note that invisible views are never created unless they are declared. This makes sense if you think about it–if they aren't declared there is nowhere to send the Open (or Show) message. The declare mechanism is necessary to handle the case of initially invisible views like dialog boxes. If the declared view is initially visible then you can write code in its viewSetupDoneScript to achieve the desired effect. The decision is up to you. Most people find it easier to just to use the declare mechanism.

Preallocation
It's important to understand the difference between preallocating and opening a view. A preallocated view has its _parent and _proto slots set up and its viewCObject slot set to nil. This actually uses very little RAM. When the view is opened, a more substantial object is allocated in RAM and viewCObject is set to reference it. You can check if a view is opened by checking if its viewCObject slot is non-nil.

When a preallocated context is initially created, its _parent slot is set to reference the view to which it's declared. This may or may not be its true parent (it may be a grandparent). When it's time to use the preallocated context to open the declared view, the _parent slot is changed to the true parent. Since this happens before the declared view's viewSetupFormScript is executed, you're normally unaware of all this. There is one case, however, where this can affect you. If the declared view is initially invisible it doesn't get opened automatically–you have to explicitly open it. Its _parent slot doesn't get updated automatically. This problem rarely occurs in practice. Most views are either declared to their parent or are initially visible.

Declare Implementation Details
In order to accomplish pre-allocation, declaring a view creates some extra slots in both the template being declared and the template to which the view is declared. In the template being declared, a preallocatedContext slot is created which contains the name you gave it in NTK. Before a view is created from a template, the view system first checks for a preallocatedContext slot in the template. If the slot exists, then this view has been declared (and allocated)–this preallocated view is used instead of creating a new one. The symbol in the preallocatedContext slot is used to find the preallocated view–the parent chain is searched for a slot with this name.

A template with declared views has a slot for each declared view. These slots' names are the names of the views being declared. They are initialized to nil.

A template with declared views also has a stepAllocateContext slot. This slot contains an array with two entries per declared view. The two entries are the declared view's name and a reference to the declared view's template. In system-defined templates and protos there is a similar allocateContext slot. The stepAllocateContext slot is for declared stepChildren, the allocateContext slot is for declared viewChildren.

The stepAllocateContext and allocateContext slots are used when a view is opened, in order to pre-allocate all the declared subviews. For each pair of entries in these arrays, there is a slot in the view. The slot's name (the subview's declared name) is the first element of the pair. The slot contains a preallocated view.

To try and help you visualize all these details, consider the simple case of two views, view1 and view2, with view2 declared to view1. Figure 4a shows the visual hierarchy; Figure 4b shows some of the underlying data structures.These details are rarely of much practical use, but if you're crawling around in the Inspector and see these weird slots in your templates, you'll know they're part of the declare mechanism.

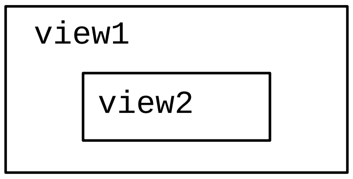
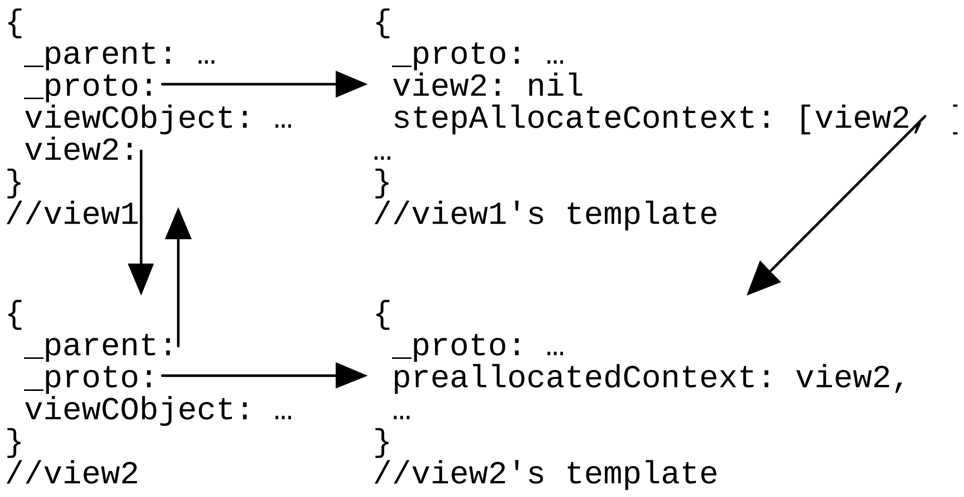


Figure 4a - A simple declaration.



```
{                    {
 _parent: …           _proto: …
 _proto:              view2: nil
 viewCObject: …       stepAllocateContext: [view2,
 view2:              …
}                    }
//view1              //view1's template

{                    {
 _parent:             _proto: …
 _proto:              preallocatedContext: view2,
 viewCObject: …       …
}                    }
//view2              //view2's template
```

Figure 4b - The data structures underlying Figure 4a.

View Opening and Closing
When a view is opened, all of its declared subviews are preallocated. When a view is closed, all those slots containing the preallocated views are set to nil. Declare keeps pre-allocation exactly one step ahead of you. At any point in time, all the open views have their declared children preallocated, ready to be opened. Closed views can't do anything anyway, so setting their pre-allocation slots to nil allows garbage collection.

When a view is closed, the slots containing the preallocated contexts are set to nil instead of being deleted. For your application's base view, this has an interesting implication. The first time your base view is opened, a slot for each declared view is created. When your base view is closed, those slots are not deleted. If you're looking for memory leaks by comparing free memory before you open and after you close your application, you'll see a slight discrepancy the first time you open and close it due to these new slots. This shouldn't be a problem on subsequent opens and closes.

Pre-allocation happens early in the view opening process, before the viewSetupFormScript is called. When it comes time to instantiate subviews, if the subview has a preallocated context, that preallocated context is used, otherwise a new one is created. Because pre-allocation occurs so early in the view instantiation process, there is a subtle bug that can arise in your program. If you declare a view (let's call it X), and send it an Open message in your viewSetupFormScript, it already has its visible flag set–two viewCObjects get created. One is created when you send the open message in your viewSetupFormScript; the other as a result of the normal view instantiation process. The symptom you see in your program is that you close X and it's still there. You can even look at X in the Listener and see that its viewCObject is nil, but there is an extra viewCObject floating around, still being drawn. Unfortunately, the view system doesn't check for this anomaly during view instantiation and handle it more gracefully.

There are two pieces of advice that you should follow which avoid this problem entirely. First, if a view can be opened naturally, let it be opened. Don't explicitly call Open. Second, don't send Open messages, or any other messages, to your children until your viewSetup-DoneScript executes. Your children are not created until your viewSetupDoneScript executes.

Declaring Your Base View
Conceptually, your base view is declared to the root view automatically after your package is downloaded and your InstallScript runs. It's name is your AppSymbol. That's why `GetRoot().AppSymbol` returns your base view. In NTK, you set your application's AppSymbol in the SettingsÉ dialog.

The preceding comment is qualified with "conceptually" because your base view isn't installed in the root view by the declare mechanism I've already described. The end result is the same–your base view is preallocated in the root view–but the process isn't the same (you'll note your base view's template doesn't have a preallocatedContext slot).

Declaring Linked Subviews
The linked subviews feature of NTK adds a twist to declaring views. The best way to think of a linked subview is as a place holder and a linked-layout. You drag the place holder (the LinkedSubview icon from the proto palette) to the parent you want your view to actually have. Then you use the Link LayoutÉ command to link it to a layout. The place holder is just that, a place holder. The linked layout is substituted for the place holder when your project is compiled. All traces of the place holder are lost, exceptÉ

If you want to declare a linked subview, you should name and declare just the place holder. You do not need to name or declare the linked-layout. In fact, you are unable to declare the linked-layout because its Allow Access popup menu is empty–it appears to have no parents. You may want to name the linked-layout so that its children can be declared to it. There is no need to give it the same name you gave the link, but it's a good idea. Figure 5 illustrates these points.

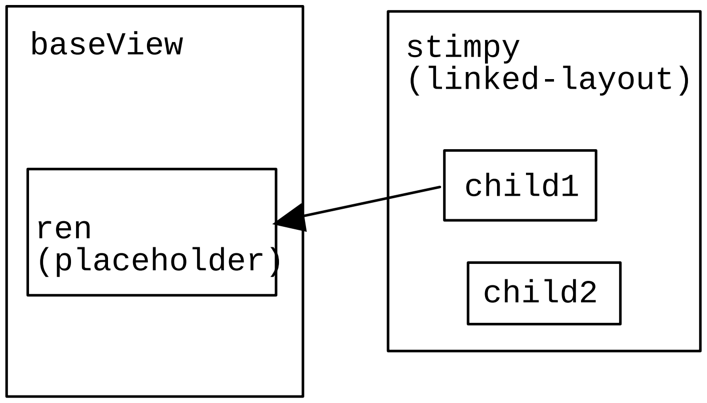


Figure 5 - A place holder and linked-layout relationship.

Declare ren to baseView and declare child1 and child2 to stimpy. You are unable to declare stimpy to anyone because it appears to have no parents. This is not a problem–you really want to declare ren anyway. When you declare child1 and child2, the only place to declare them is stimpy; baseView is not in the popup because stimpy appears to have no parents. This is a limitation of using linked views. If baseView needs to access these children you can use code references like `ren.child1`.

Note that the names chosen for the place holder and the linked-layout (ren and stimpy) are not the same, but they could be. The symbol stimpy is not available at runtime. If child1 or child2 need to refer to their parent, they should use ren. This illustrates why it's a good idea to name your place holder and linked layout identically–the code is clearer. It's kind of strange for child1 and child2 to refer to their parent as ren, a symbol defined outside the linked-layout in which they're defined.

Dynamically Adding Views
Adding views dynamically actually turns out to be a complex issue. I describe several techniques in the following sections. You need to pick the one most appropriate to your situation.

Don't Create Views Dynamically
The canonical case is a dialog box. You should create the dialog ahead of time (probably using the linked view feature of NTK) and declare it to your base view. Then at runtime open and close it as you please.This is the preferred method–it's by far the simplest and least error prone.

Don't Create Views Dynamically–Yourself
Perhaps you need to create a group of subviews, but you don't know the exact number until runtime. For example, your application may need to compute the number of subviews based on screen size (to maintain compatibility with future products). The best way to accomplish this is to compute your stepChildren array in your viewSetupChildrenScript. Just put in the templates you want and the view system handles creating the actual views.

If you're going to adjust the size of your stepChildren array using AddArraySlot it had better be in RAM. You might need to do:

```
if not HasSlot(self, 'stepChildren) then
  self.stepChildren := Clone(self.stepChildren)
```

HasSlot is used to test if the viewChildren slot already exists in RAM in the view itself. If the viewChildren already exist, you don't need to do anything. Using `self.stepchildren` in the condition instead of HasSlot doesn't work because the _proto chain is followed into ROM. All the explicit selfs are needed lest you accidentally get the stepChildren slot of your parent.

If you want to modify the contents of the stepChildren array after your view is created, and create new views, use the RedoChildren method. First make the changes you want to the stepChildren array, then send your view a RedoChildren method. All of the view's current children are closed and removed, the viewSetupChildrenScript is called, and then a new set of children is created from the stepChildren array. Note that reordering stepChildren and then calling RedoChildren is a way to re-order your child views dynamically.

If You Insist, There's AddStepView

If you really want to create a template at runtime and then make a view from it, use the global function AddStepView. It takes two arguments: the parent view you want your view added to and the template to use in making the view. It returns the view that is created. Be sure to save this return value so you can access the view later.

AddStepView doesn't force a redraw. You need to either:

- Send the new view a Dirty message;
- Send the view's parent a Dirty message. This is useful if you're adding multiple views–one Dirty and they all show up at once (if you're adding several views, you should also consider using the RedoChildren technique); or
- If the template has the visible bit cleared, just send the new view a Show message. This is useful if you want the view to appear with some sort of special effect.

In addition to creating the view, AddStepView also adds its template to the stepChildren array. This means that the stepChildren array needs to be in RAM so it can be expanded or AddStepView will fail. See the previous comments on ensuring your stepChildren array is in RAM. Do not use AddStepView in your viewSetupFormScript or your viewSetupChildrenScript–it doesn't work. Instead, append your template to the stepChildren array and let the view system do the rest of the work. There is another function, AddView, which works like AddStepView, except it adds the template to the viewChildren array. You should use AddStepView, since the viewChildren slot is reserved for system use.

There is also a pair of functions, RemoveStepView and RemoveView, that are designed to undo the effect of AddStepView and AddView. These functions take one argument, the view (not the template), and remove that view. Unfortunately, they do not remove the corresponding template from the stepChildren or viewChildren slot. You have to do this yourself. If you do not take care of this, your stepChildren slot builds up all these old templates which waste RAM and which may surprise you if something forces a RedoChildren (all the "removed" views' ghosts suddenly come back to life).

BuildContext

Another function that is occasionally useful is BuildContext. It takes one argument, a template. It makes a view and returns it. The view's parent is the root view. The template is not added to any viewChildren or stepChildren array. Basically, you get a free-agent view.

Normally, you shouldn't need BuildContext. It's useful when you need to create a view from code that isn't part of an application (when there's no base view to use as a parent). For instance, if your InstallScript or RemoveScript needs to prompt the user with a dialog, you need to use BuildContext to create the dialog.

Creating Templates

All but the first of the techniques presented require you to create templates. You can do this using NewtonScript to define a frame. You have to remember whether the oneLineOnly bit gets set in the viewFlags slot or in the viewJustify slot. It's easy to mess up.

I recommend creating a user proto in NTK and then using it as a template. That allows you to take advantage of NTK's slot editors. At runtime you can access the user proto as PT_<filname>. If there are slots whose value you can't compute ahead of time, leave them out of the user proto, and at runtime create a frame with those slots set properly and proto off the user proto.

A typical example might be computing the bounds on the fly. If you define all the static slots in a user proto in a file called dynoTemplate, you can create the template you need using code like:

```
template := { viewBounds: RelBounds (
          x, y, wid, hgt),
             _proto: PT_dynoTemplate }
```

This really shows off the advantage of a prototype-based object system. You create a small object on the fly and use inheritance to get the rest of the values you need. Your template is only a two slot view in RAM. The user proto sits on the card with the rest of your application. The conventional RAM-wasting alternative is :

```
template := Clone (PT_dynoTemplate);
template.viewBounds := RelBounds (x, y, wid, hgt);
```

You should also note that for creating views arranged in a table, there is a function called LayoutTable which calculates all the bounds. It returns an array of templates. See the *Newton Programmer's Guide* for details.

Summary of Dynamic View Advice

- Avoid adding views dynamically; use declare and invisible views instead. Barring that, add your templates to the stepChildren array in your viewSetUpChildrenScript.
- Do not send messages to or set slots in templates. This is most commonly done in code that uses the stepChildren array instead of :ChildViewFrames(). It also happens when people use the template passed to AddStepView instead of the view it returns.
- The stepChildren array must be in RAM before you can modify it. Remember, "never try to write to ROM–it wastes your time and annoys the ROM."
- Do not anger AddView.
- Use NTK to define as much of your template as possible.

p

		template	view
1)	created by NTK	X	
2)	only reside in RAM		X
3)	handles `:Hide()`, `:Open(), and :Close()`messages		X
4)	reside in stepChildren and viewChildren arrays	X	
5)	code modifies their slots at runtime - preferably, using `SetValue`		X
6)	use system protos in their proto slots	X	
7)	passed to AddStepView	X	
8)	usually contains few slots (inherits, most of its slots)		X
9)	usually reside in ROM	X	
10)	returned by `:ChildViewFrames()`		X
11)	returned by Debug()		X
12)	is a frame	X	X
13)	has a parent		X

Table 2 - The template vs. view quiz answers.