Using a Prototype-based Language for User Interface: The Newton Project's Experience

Walter R. Smith

Apple Computer, Inc.
1 Infinite Loop, MS 305-2B
Cupertino, CA 95014
wrs@apple.com

Abstract

Object-oriented user interface frameworks are usually implemented in a class-based language. We chose instead to develop a prototype-based language, NewtonScript, for this purpose. We found that prototype inheritance has compelling advantages over classes in the domain of user interface programming, and can help overcome the memory constraints of a small machine.

1. Introduction

The benefits of an object-oriented approach to UI programming are well established. Object-oriented graphical user interface frameworks, like Smalltalk-80's Model-View-Controller system [8], Apple's MacApp [4], Metrowerks' PowerPlant [10], and the Microsoft Foundation Classes [11], have become quite common. A user interface toolkit is generally expected to have an accompanying object-oriented framework, either integrated to begin with or added later.

Virtually all of these frameworks are implemented in a class-based language—not surprising, since the most popular object-oriented languages are class-based. However, we have found a prototype-based language to be much better suited to user interface programming. This paper reports our experience using the prototype-based NewtonScript™ language to implement the application framework for Newton® [14].

A chronology of the project would be much too large for this paper, and we don't have room to present all the useful things we learned. We have focused exclusively on the choice of a prototype-based language and the benefits and disadvantages it brings.

2. Newton language history

The Newton software platform was created to support "Personal Digital Assistants" (PDAs): very small, portable personal computers. At one point, we intended Newton devices to run simple form-based applications using a built-in database. The programmability would be limited mostly to designing screen forms and specifying the links between database fields and visible fields. Applications would be static form descriptions rather than programs. Given the simplicity of the requirements and the small size of the machine, a low-level language like C++ was the obvious choice. We did eventually write much of the core Newton software in C++.

However, as we refined the design, it became apparent that more flexibility would be needed from the application framework, and we found this difficult to achieve with C++. A two-language architecture evolved as the project progressed. We continued using C++ for efficiency in low-level components, but developed a new language called NewtonScript for the higher-level software, including much of the user interface toolkit. Newton application developers work almost exclusively in NewtonScript.

We justify the use of a relatively slow interpreted language for user interface programming by recognizing that most of an application's work has to do with coordinating the user interface, not high-performance data structure crunching. That is, in a lot of

To appear in the Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications.

application code, logical structure and ease of construction is more important than raw speed. For this sort of code, a bytecode interpreter is fast enough most of the time.

When greater speed is necessary, programs move to a low-level language (C++) to do the operation, then return to NewtonScript. For common operations like searching and sorting, we provide a library of fast C++ functions that NewtonScript programs can use when necessary. Effective use of these predefined functions greatly improves application performance.

To give application developers more flexibility, we have produced a native-code NewtonScript compiler, and are in the process of providing C++ tools that developers can use. We anticipate, however, that interpreted NewtonScript will remain the core application language.

The semantics of NewtonScript evolved in parallel with the rest of the Newton software architecture [14]. In particular, the language had to integrate well with the persistent object store and the view system. Although we were initially skeptical of prototype-based languages, we eventually discovered that the prototype-based model was natural for what we were trying to do.

3. Prototype-based inheritance

The distinguishing characteristic of inheritance in NewtonScript and other prototype-based languages [7, 12, 16] is the lack of classes. In these languages, objects inherit attributes directly from other objects.

A class-based system makes a distinction between *instances* and *classes*. An instance belongs to a class, which determines the instance's structure and behavior. A class acts as a template for its instances, defining the set of attributes each instance will have, and as a repository of instance behavior.

A class may be a *subclass* of another. The subclass inherits the instance template and behaviors of the superclass, with some modifications. Although the values contained in an instance may be changed, the only way to alter the structure or behavior of an instance is to create a subclass and use it.

In a prototype-based system, there is no need for a class/instance distinction; there are only "objects". Objects themselves can contain behaviors, and each object can have a unique structure, not limited to a predefined template.

Objects can inherit state and behavior from other objects directly. Each object may specify another object (or more than one, depending on the language) from which it inherits attributes.

Prototype-based inheritance has two big advantages: simplicity and concreteness. It is simpler because the concept of "class" is eliminated, and rather than two inheritance relations, subclass and instance, there is only one (usually just called "inheritance"). Eliminating this "extra color of chalk" reduces the cognitive load on the programmer. It is more concrete because programming consists of modifying objects directly, rather than modifying classes to produce an indirect effect on objects. The process is like assembling a bicycle instead of writing the assembly manual for a bicycle.

4. The programming dichotomy

Programs with graphical user interfaces usually contain two very different components: the "model" of the data being manipulated, and the user interface that manipulates it. Often, these components are best implemented using different styles of programming.

Class-based programming is intended to make it easy to share structure and behavior among many instances. This is ideal for the "model" part of a program, which usually involves manipulating a complex but fairly repetitive data structure such as a database, spreadsheet, or word processing document. Factoring out all behavior for a group of instances makes it easier to reason about the entire group at once. In fact, the benefits of classes in this sort of programming are sufficiently well established that we will not present further justification.

The needs of the user-interface side of the program are different. In contrast to the model, the user interface usually consists of relatively few objects, most of which appear only once in a given context, and most of which are unique in small but significant ways.

For example, consider the dialog box in Fig. 1, which contains two pop-up menus and two buttons. The purpose of this dialog box is to set the font and size of something. Clearly, it has unique behavior.

Figure 1. Dialog box example

However, there is at most one such dialog box open at any given moment. Similarly, each pop-up menu has unique behavior: one sets the font, the other sets the size. Again, however, there is only one of each in existence.

Of course, all pop-up menus have something in common, and so do all buttons. These commonalities should be factored out somehow. Generally, however, the programmer of the *toolkit* does this factoring; most user interface programming effort is spent on unique objects.

Creating a class is one of the most complicated operations in a programming environment. It is expensive in time, complexity, and cognitive load on the programmer. Normally, these costs are amortized over many instances. But in user interface programming, there is no such leverage—most of the objects are "one-offs". Such objects do not fit well into the class-based programming model; in fact, "object-oriented programming" texts commonly advise that if there is only one instance of a class, something is probably wrong with the design.

Because of the relative difficulty of subclassing and the injunctions against "one-offs", programmers try to avoid defining subclasses for unique objects. Class-based user interface code will often substitute attributes for behaviors, and move behaviors to inappropriate places, just to get enough functionality concentrated in one place to justify creating a new class.

We have found that prototype-based programming is ideal for user interfaces. The situation is reversed: creating unique objects is the *most natural* operation in the prototype-based programming environment. It is easier to reason about and control the interactions of *individual* objects—the usual requirement for UI programming—when the objects themselves are being programmed directly.

Prototypes do allow factoring when needed, and in a particularly natural manner. A "prototype" is a functional (or nearly so) example of a certain kind of object, rather than a partial description of an object's structure and behavior.[*] Programming an object to be shared is only slightly more complicated than programming an individual object; more importantly, the same tools and concepts are used to do both.

The concreteness of prototypes shines in the UI programming domain. The class-based programmer is constantly shifting between abstraction levels, from thinking about instance templates (classes) to thinking about instances. Worse, the programmer has to alter instances by "remote control"—by editing code in a class that will result in a change to an instance later. The programmer in a prototype-based system, on the other hand, is always editing objects.

As mentioned above, the wonders of the prototype-based model do not eliminate the need for a more structured approach to some parts of an application. Class-based programming may be done in NewtonScript as a *style* of prototype-based programming, similar to the use of "traits" objects in Self [15]. This style is explained in more detail elsewhere [13]; we describe it briefly in section 9 below.

5. Visual tools

Object-oriented UI frameworks are frequently accompanied by "visual" UI construction environments (examples include MacApp's ViewEdit, PowerPlant Constructor, and Visual C++ [11]). Such an environment allows the views that make up a user interface implementation to be specified graphically: an interface can be largely "drawn" rather than coded. In addition to specifying the location and hierarchical placement of views, attributes specific to certain types of views may be specified.

These visual envionments make it easy to create instances and set their attributes, but not to define new subclasses; that can only be done by leaving the visual environment and using a text editor. This is a major limitation, since an instance may have a unique

[*] A bicycle without paint, perhaps, rather than a bicycle assembly manual that omits the paint chapter.

value for an attribute, but cannot have a unique behavior, since only classes can define behavior. The visual editor is generally limited to connecting instances together and editing certain of their attributes, though it may produce placeholders in the source code for more complex attributes.

The ease of use of the visual environment is actually rather subversive, in that it makes the programmer even *less* likely to want to return to the text editor and create a new subclass.

With a prototype-based UI toolkit, a visual editor can be both more natural and more powerful than its class-based counterpart. Visual environments are ideal for editing characteristics of individual instances. In the prototype-based model, *all* programming takes this form. Editing a method is the same as editing a property. It is easy to integrate the code editor into the visual editor, and then there is no need to switch to a text editor to change the code (unless you want to).

At least one prototype-based visual environment [6] has taken the extreme approach of giving every object in the system a visual representation. The Newton development environment, Newton Toolkit [3], takes a more moderate approach, showing the programmer two simultaneous views of the user interface, one in graphical form and the other as a hierarchical object browser. Only view objects appear in the graphical view. (See Fig. 2.)

The view objects are shown both graphically, as they will be interpreted visually by the view system, and as collections of slots. These two editors show all the attributes and behaviors of the objects; no other environment for editing view behavior is necessary.

Newton Toolkit is currently somewhat weak at supporting non-UI code. The browser is really just a "frame hierarchy" editor, with few features specific to views, so it can be used to edit non-view frames. Plain text files are also supported as a least common denominator. We plan to provide better support for other kinds of code, particularly class-style code (see Section 9), in future versions of the Toolkit.

6. Container inheritance

Container inheritance is helpful in user interface programming, and easily implemented in a prototype-based object model. NewtonScript's inheritance system was designed to make container inheritance very convenient.

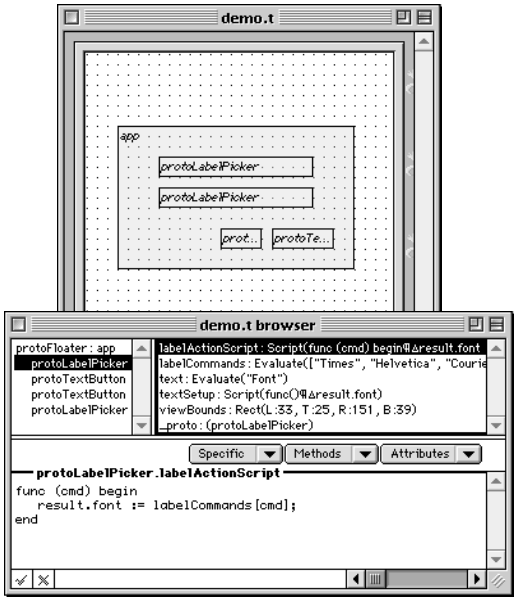

Figure 2. Newton Toolkit example. The top window shows the graphical outline view of the dialog box in Fig. 1. The lower window shows the same object hierarchy in browser form.

Newton's container inheritance was inspired in part by HyperCard, in which inheritance is based on a *part-of* relation, rather than the more typical *is-a* relation. In HyperCard, an application (or "stack") consists of parts arranged in a hierarchy: the stack contains backgrounds, which contain cards; backgrounds and cards contain fields and buttons (see Fig. 3). Unhandled messages travel up the hierachy of containment. For example, if a message is sent to a button, the system will look for a handler in the button, then in its card, then in the card's background, then in the stack.

Newton's inheritance model is more general than

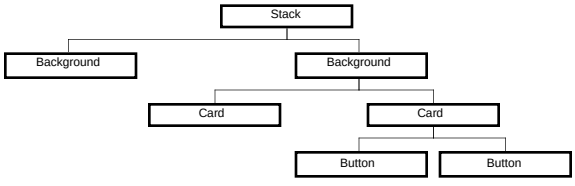

Figure 3. HyperCard container inheritance

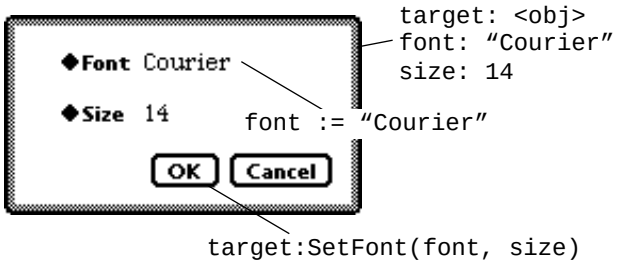

Figure 4. Container inheritance. The dialog box defines slots that can be used as inherited variables by the subviews to manage the user interaction.

HyperCard’s. In HyperCard, the set of object types and their hierarchical arrangement is fixed—it is impossible to create a new type of object, like a radio button cluster. Newton lacks this restriction. Also, in NewtonScript, objects inherit properties (variables) in addition to message handlers.

Naturally, then, Newton’s idea of container inheritance is more general than HyperCard’s. In addition to specifying a prototype, an object can specify a container object to inherit from. The container path has lower priority than the prototype path, so the effect is that container inheritance applies to the “virtual objects” created by prototype inheritance (see Section 7).

General container inheritance is most useful as a form of “visual scoping”. Views in the Newton system are arranged in a hierarchy, and the view objects’ container inheritance paths are connected accordingly. That is, a view inherits from its superviews.

This allows for a natural form of information hiding: a view’s slots are visible to its subviews, but nowhere else. Thus, the graphical view editor is showing not only the placement of objects on the screen, but the “variable scopes” created by container inheritance as well.

In our dialog box example, the box itself might contain a slot referring to the object whose font and size are being set (the “target”), and two others holding the size and font currently selected (see Fig. 4). These slots act as variables that are “in scope” in the menus’ and buttons’ methods, and in the dialog box itself, but nowhere else in the application.

Each subview can refer to these slots as inherited variables. Thus, the “Font” menu reacts to a menu choice by setting the inherited “font” variable, and

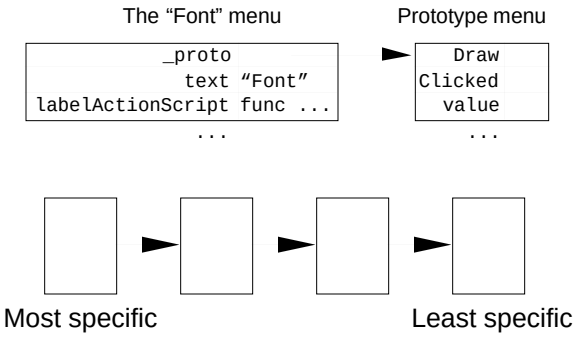

Figure 5. A tiny example of `_proto` *inheritance. The “Font” menu frame inherits menu-like behavior from its* `_proto` *frame, the prototypical menu, and adds its own specific title and behavior when an item is picked. In general, the* `_proto` *chain leads from a specific (more refined) object to progressively less-specific objects.*

the “OK” button sends a message to the inherited “target” containing the inherited “font” and “size”.

7. NewtonScript inheritance

Before delving into an example, we first need to present some details NewtonScript. We will not attempt to explain all of NewtonScript here (more details can be found in [1, 14]). It is only necessary to explain how inheritance is defined.

Objects called *frames* are the basis of object-oriented programming in NewtonScript. A frame is a collection of tagged slots, each of which contains a value. Slots can contain references to other objects, including frames, and can contain function objects that provide behavior.

Inheritance occurs when the special slot tags `_proto` and `_parent` are present in a frame. The `_proto` slot is used for prototype inheritance; `_parent` is used for container inheritance. The two slots are similar in operation, with `_parent` having lower priority, but there are some special rules that make things interesting.

If a `_proto` slot is present in a frame, it refers to another frame, whose slots are inherited. For example, looking up the tag `Draw` in the Font menu frame in Fig. 5 will find the `Draw` slot in the `_proto` frame (the prototype menu). The effect of `_proto` inheritance is to produce a chain of refinement—a series of ever-more-specific objects.

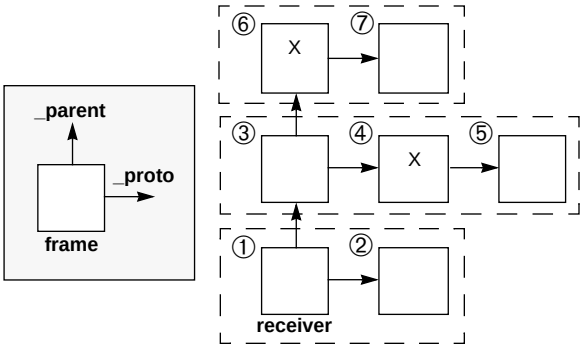

Figure 6. The NewtonScript "comb". Looking up the slot x in the frame marked "receiver" will find the x slot in frame ④. Assigning to x will create a new x slot in frame ③. Dotted lines show the "virtual" objects defined by the `_proto` *relation.*

Assignment of a slot never occurs in a `_proto` frame; an assignment to `value` will create a new `value` slot in the outer frame rather than altering the `_proto` frame. From the point of view of the outer frame, this has the same effect, since the new slot overrides the value in the `_proto`. The rule lets us avoid altering the `_proto` frame, which is almost always the right thing to do, since it may be inherited by many other objects. It is convenient to be able to specify an initial value in the prototype and get this "copy-on-write" behavior on assignment, and it saves memory (see section 10).

The `_parent` slot works similarly, but has a lower priority than the `_proto` slot (that is, the `_proto` slot is searched first), and does not have the assignment restriction. The `_parent` slot is not searched in `_proto` frames; only the `_parent` chain of the receiver is considered. Whereas a general two-path inheritance system would produce a complex inheritance tree, these rules have the effect of creating a simpler comb-like inheritance structure (Fig. 6).

The comb can be viewed as a refinement relation (`_proto`) embedded within a containership relation (`_parent`). Each "tooth" of the comb is like a single "virtual" object; these virtual objects are linked together by the `_parent` slots.

The comb structure evolved to support the features described in this paper. The most important is traditional prototype-based inheritance, which occurs through the `_proto` slot. We also found that the `_proto` mechanism provides a way to reduce system RAM requirements—for PDA devices, an important consideration (see section 10). The `_parent` slot provides container inheritance, which is especially useful in user interface programming, and also enables class-style programming (see section 9).

8. Comparative example

Now we present a simple example of a user interface implementation in both the class-based and prototype-based models. Our example will be the font and size dialog in Fig. 1.

8.1 Class-based implementation

Of course, there are behaviors and properties common to all pop-up menus. It makes sense to share these attributes by putting them in the class PopUpMenu.

Each of the menus in this dialog box, however, has certain unique properties. For our purposes, let's consider them to be a location, a name ("Font"/"Size"), a set of items (the list of available fonts/sizes), and a behavior when an item is picked (setting the current font/size). There are a few different ways of handling these unique attributes in the class-based framework.

One way to implement the "Font" menu would be to implement a new class for it, inheriting from the PopUpMenu class (Fig. 7). There would be attributes for location and name, and methods for generating the set of items and handling the item picked. The general methods in PopUpMenu would send mes-

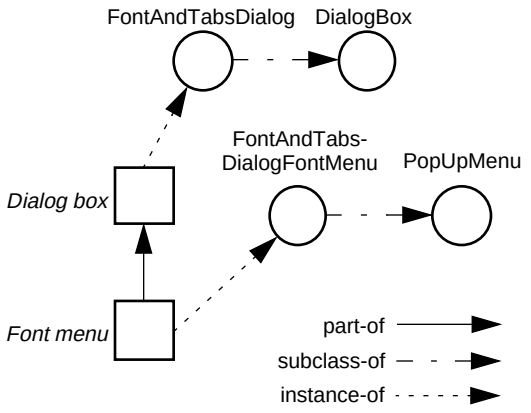

Figure 7. A "proper" class-based implementation

sages to “self” to get this information.

This is a very “proper” class-based implementation, but from a practical point of view, it leaves much to be desired. Implementing a new class is usually an effort: one has to switch editors, engage in a different mental process, endure a relatively slow recompilation, and (perhaps worst of all) generate a unique name for the class (like FontAndTabsDialog-FontMenu). Of course, all of this must be done for the “Size” menu as well.

A more common way to implement this dialog would be to use a basic PopUpMenu class for the menus (Fig. 8). Each menu instance has attributes for location, name, and set of items. Because the set of items must be determined at runtime, there is code in the FontAndTabsDialog class to set the menu’s item list attribute. Because the basic PopUpMenu class doesn’t know what to do with the values picked, menu instances have an attribute for a “controller”, which in this case is set to the dialog box. The PopUpMenu class sends a message to the view in this attribute when an item is picked; in this example, the dialog box reacts to the message by setting the font or size, depending on the sender. Since the sender is not ordinarily part of the message, it is included as an argument.

It doesn’t really make sense to do these menu manipulations in the dialog box class. The only reason the dialog box must set the menu item list and respond to the item-picked message is that it’s the nearest object with enough unique behaviors to justify the effort of creating a new class. The difficulty of giving objects unique behaviors has caused the design to become more coupled and harder to understand, having encouraged the programmer to bend the rules for “convenience” (and the end result isn’t really very convenient after all).

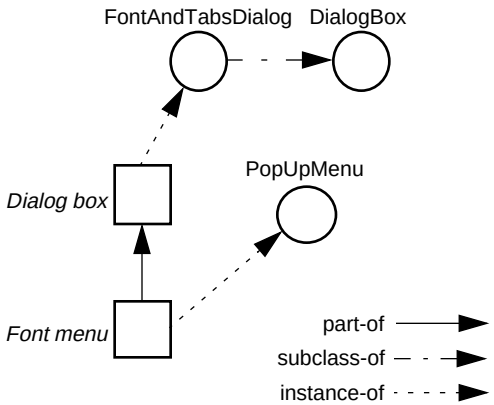

Figure 8. A more typical class-based implementation

8.2 Prototype-based implementation

Now consider this example in a prototype-based language (Fig. 9). There is still a PopUpMenu object containing the shared attributes and behaviors of pop-up menus. However, PopUpMenu in this case is a *prototype* of a menu. It is a functional menu object; it just doesn’t have a particular purpose. To use it, the programmer creates an object that inherits from it and adds the characteristics needed for its specific task; in this case, the four attributes mentioned.

The menu object in the dialog box has its own methods for generating the list of items and reacting when one is chosen. It’s not necessary to define a new class (or think of a name for it). By making the menu object inherit from the prototypical PopUpMenu, the programmer is directly expressing that this menu is the same as the prototype, but with a few specific new attributes.

To show the advantage of the reduced coupling in the prototype-based design, imagine that we want to replace the size menu with an editable text box, so any size can be written or typed in. In the Fig. 8 design, we have to change the dialog box class, removing the menu operations and replacing them with text box operations. In the Fig. 9 design, we

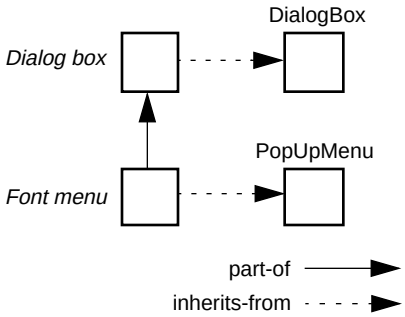

Figure 9. Prototype-based implementation

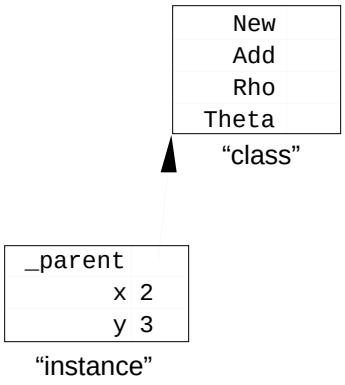

Figure 10. Class-based NewtonScript style.

simply replace the menu object with a text box object, and put the method that sets the "size" variable in the new object.

In Listing 2, we present the NewtonScript source code for this dialog example. We ignore one aspect, which is where the value of "target" comes from; it's not important to this discussion. We assume the target has GetFont, GetSize, and SetFont methods. Also, keep in mind that `_parent` slots will be created at runtime to link the subviews to the dialog box view.

9. Class-based style

As we noted above, there are situations in which class-based programming is more appropriate than the prototype-based style described in the last section. However, a language with prototype inheritance can be used in a class-based *style*, as described in detail elsewhere [13, 15]. We will cover it briefly here.

To write a class-based program in NewtonScript, we construct objects corresponding to classes and instances. For the purposes of this discussion, we will refer to them by those names, but keep in mind that in NewtonScript, this is merely a convention, not something built into the language.

A class object contains state and behavior shared among all instances, and an operation that creates new instances. It should also be possible to inherit from another class. Each instance object needs some state (as defined by the class), and an inheritance link to its class.

We can translate this description of the class-based style directly into NewtonScript objects, as in Fig. 10. A NewtonScript "class" is an object containing a `New` method, which creates an "instance", and the instance methods. An "instance" is an object containing only data slots, with a `_parent` link back to the class. The `_parent` slot is used rather than `_proto` for two reasons: it allows slots in the class to be used as "class variables" that are shared among instances, which would run afoul of the `_proto` assignment rule (see Section 7) if `_proto` was used, and instances can use `_proto` to point to a "prototypical instance" containing initial values for instance slots.

There is no explicit support in NewtonScript for this style of programming; it requires a certain amount of self-discipline. Because it is so useful, however, we are investigating ways of supporting it more explicitly in the compiler and development environment.

The NewtonScript code for a simple "point" class is shown in Listing 1.

10. Improved RAM usage

PDAs generally cost well under $1000, are handheld, and run on a few standard batteries. This budget results in severe hardware constraints. The greatest hardware expense is in the memory subsystem—in particular, fast system RAM. Thus, the PDA software designer is in the position of trading off ROM and processor time for RAM in order to keep hardware costs down.

In a typical class-based application framework, views are objects in RAM created from templates. For example, MacApp uses disk-based descriptions

```
Point := {
   New: func (x,y) {_parent: Point, x: x, y: y},
   Add: func (pt)
           begin
               x := x + pt.x;
               y := y + pt.y;
           end,
   Rho: func () ...,
   Theta: func () ...,
};

p := Point:New(1,1);
q := Point:New(2,3);
p:Add(q);
r := p:Rho();
```

Listing 1. Simple class-based example.

```
fontSizeDialogExample :=                    This is the main dialog box view.
    {_proto: ROM_protoFloater,              Its prototype is the generic "floating window".
     viewBounds: {left: -4, top: 62, right: 168, bottom: 162},
     declareSelf: 'base,                    At runtime, create a slot "base" that holds a reference to the
                                            dialog box view itself.
     font: nil,                             Font and size maintain the current settings; they are visible
     size: nil,                             to this view and its subviews.
     target: nil,                           Target is the object whose font and size are being set.
     viewSetupFormScript:                   The view initialization method, which gets the
       func() begin                         initial values of font and size from the target.
        font := target:GetFont();
        size := target:GetSize();
       end,
     viewChildren: [fontMenu, sizeMenu,     The array of subviews (defined below).
         okButton, cancelButton],
    };

fontMenu :=                                 This is the "Font" popup menu view.
    {_proto: ROM_protoLabelPicker,          Its prototype is the generic labeled menu of values.
     viewBounds: {left: 33, top: 25, right: 151, bottom: 39},
     labelCommands: nil,                    Placeholder for the array of menu commands (calculated later)
     text: "Font",                          The menu's title.
     viewSetupFormScript:                   The view initialization method gets the set of
       func()                               available fonts and uses it as the labelCommands array.
         labelCommands := self:GetFontList(),
     textSetup:                             The menu's initial value is the current font setting, inherited
       func() font,                         from the dialog box (the menu's superview).
     labelActionScript:                     The method called by the protoLabelPicker when an item is
       func (cmd)                           chosen from the menu. This sets the inherited "font" variable
         font := labelCommands[cmd],        to the chosen font.
    };

sizeMenu :=                                 This is the "Size" popup menu view..
    {_proto: ROM_protoLabelPicker,          The prototype is once again the labelPicker.
     viewBounds: {left: 33, top: 49, right: 151, bottom: 63},
     labelCommands: ["9", "10", "12",       In this menu, the options are fixed.
         "14", "18"],
     text: "Size",
     textSetup:                             Concatenating the size integer with the empty string is just an
       func() size & "",                    way to convert it to a string.
     labelActionScript:                     When a menu item is picked, convert the item to an integer
       func (cmd)                           and set the inherited "size" variable to the chosen size.
         size := floor(StringToNumber(labelCommands[cmd])),
    };

okButton :=                                 The "OK" button view.
    {_proto: ROM_protoTextButton,           Its prototype is the generic button labeled by a text string.
     viewBounds: {left: 82, top: 82, right: 110, bottom: 94},
     text: "OK",                            The button's title.
     buttonClickScript:                     The method called by the protoTextButton when the button
       func() begin                         is clicked. This uses the inherited "font" and "size" variables
        target:SetFont(font, size);         to set the target's font...
        base:Close();                       and closes the dialog box ("base" is the "declareSelf" above).
       end,
    };
cancelButton :=                             The "Cancel" button view. Like the "OK" button, except that it
    {_proto: ROM_protoTextButton,           just closes the dialog box without setting anything.
     viewBounds: {left: 122, top: 82, right: 166, bottom: 94},
     text: "Cancel",
     buttonClickScript:
       func() base:Close(),
    };
```

Listing 2. The dialog box example in NewtonScript. For pedagogical reasons, this is not quite a valid NewtonScript program. Most importantly, in real life, we souldn't give names to all these views, and we wouldn't write this as a text file.

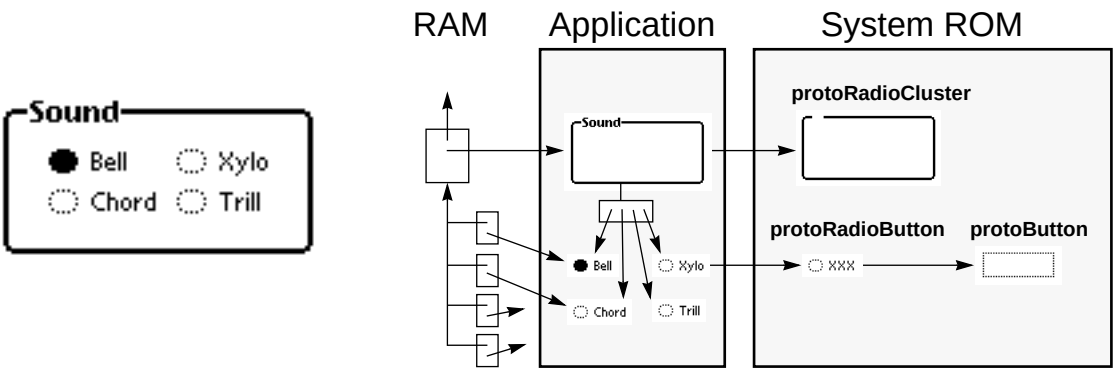

Figure 11. A view containing four subviews, and its underlying Newton objects. The views are represented by the small RAM frames on the left, containing only `_parent` *and* `_proto` *pointers. The RAM frames'* `_proto` *slots reference the view frames in the application package, whose* `_proto` *slots reference view templates in the system ROM.*

of views. The system follows the descriptions to create view objects with the appropriate properties at runtime. This technique works, and allows declarative view specifications (perhaps in a visual environment), but the resulting view objects store all of their attributes in RAM, even those that don't change during execution.

Newton takes advantage of `_proto` inheritance to keep most view information out of RAM. (See Fig. 11.) Rather than descriptions to be parsed, Newton applications contain "view templates", which are actual frames that will be mapped into read-only virtual memory and used as prototypes for the runtime views. View objects are created in RAM, but each one contains only a few slots, mainly a `_proto` slot pointing to its view template in the application and a `_parent` slot pointing to its enclosing view for container inheritance.

Thus, a Newton view starts out taking minimal RAM. As slots are set, the "copy-on-write" assignment behavior of NewtonScript `_proto` inheritance causes the RAM-based frame to expand as necessary. The end result is that RAM is used only for the information that changes; the rest of the view object stays out of RAM.

11. Problems

Of course, there is a price to pay for the benefits of prototype-based inheritance. Although we are generally happy with the system, there are some problems.

11.1 Performance

The most significant disadvantage we have found is the inherent difficulty of making the search for a slot fast in this system.

In a class-based system, optimizing slot lookups is fairly easy. Because all the instances have the same structure, the location of each slot can be predetermined by examining the classes as methods are compiled. Method lookups can be effectively cached according to class.

Because each object in a prototype-based system can have a unique structure, these mechanisms don't work, or are harder to implement. The problem is not that good performance is impossible, but rather that much of the accumulated wisdom of class-based languages does not apply. Research is in progress on ways to improve the performance of prototype-based languages [5, 9].

As mentioned in Section 2, we have found that through judicious use of low-level languages, applications can achieve reasonable speed despite these problems. Also, it should eventually be possible to compile the "class-based style" of NewtonScript code more efficiently, using techniques from class-based languages.

11.2 Visibility

NewtonScript has no provision for private slots. All objects can see all slots in all other objects. This has the potential to cause trouble.

The worst problems occur when a prototype object is modified. If a slot is added, it may accidentally conflict with a slot in an object derived from the prototype, which will cause unintended behavior. If a slot is removed, objects that rely on it may stop working (the actual problem being that they shouldn't have been allowed to rely on the slot in the first place). Class-based systems without privacy have essentially the same problems, but because there are more inheritance relationships in a prototype-based system, there are more chances for them to occur. Container inheritance is prone to similar accidents, when a slot that should belong to an object is inadvertently inherited from the object's container instead.

It would be better if the programmer could mark explicitly those slots that should be inherited. We have some ideas to address this problem, but have done no serious work on it yet.

11.3 Structure

NewtonScript does not impose much structure on the programmer. We have sometimes seen a tendency to write programs in an ad hoc manner because it is so easy to do. This is perhaps an advantage when writing a quick test application or prototyping a user interface, but for industrial-strength work, it is not a good idea.

For some programmers, it may be better to enforce program structure through the language or development environment. Now that we have gained enough experience with the system to have some idea of the "correct" way to write programs, it is probably worth exploring a more formal approach to program structure.

12. Conclusion

Using a prototype-based language for Newton was an unusual choice, but we are pleased with the results. The inheritance model is easy to understand and use, and has brought some advantages in memory usage as well.

Surprisingly, most programmers who learn NewtonScript don't seem to miss the traditional class-based paradigm; rather, we receive many comments on how unhappy they are to return to traditional languages. This is at least partially due to the ease of constructing user interfaces (other aspects of the system, such as garbage collection, are also popular). When a class-based approach is most appropriate, NewtonScript can adapt to it.

Many of the ideas inspired by prototype-based languages can be implemented on top of a class-based language. Designers of user-interface toolkits who choose to use class-based languages should consider building some prototype-like features into the toolkits, to take advantage of the simplicity and concreteness this paradigm brings.

We would like to end with a plea for more research on prototype-based languages and programming environments. Judging from our experience, the area is definitely worthy of further exploration.

13. Bibliography

[1] Apple Computer, Inc. *The NewtonScript Programming Language*. Apple Computer, 1993.

[2] Apple Computer, Inc. *Newton Programmer's Guide*. Apple Computer, 1993.

[3] Apple Computer, Inc. *Newton Toolkit User's Guide*. Apple Computer, 1993.

[4] Apple Computer, Inc. *Programmer's Guide to MacApp*. Apple Computer, 1992.

[5] Craig Chambers. The Cecil language: specification and rationale. University of Washington technical report UW CS TR 93-03-05, 1993.

[6] Bay-Wei Chang and David Ungar. Animation: from cartoons to the user interface. *User Interface Software and Technology Conference Proceedings*, Atlanta, GA, November 1993.

[7] Borning, A. H., Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/ IEEE Fall Joint Computer Conference*, pp 36-40, 1994.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[9] Urz Hölzle and David Ungar. A third-generation SELF implementation: reconciling responsiveness with performance. In *OOPSLA '94 Conference Proceedings*, pp. 229–243, Portland Oregon, 1994. Published as *SIGPLAN Notices 29*, 10, October 1994.

[10] Metrowerks Inc. *PowerPlant Manual*. Metrowerks, St. Laurent, Quebec, 1994.

[11] Microsoft Corp. Visual C++ documentation. Microsoft, 1994.

[12] Randall B. Smith, moderator. Prototype-based languages: object lessons from class-free programming (panel). In *OOPSLA '94 Conference Proceedings*, pp. 102–112, Portland Oregon, 1994. Published as *SIGPLAN Notices 29*, 10, October 1994.

[13] Walter Smith. Class-based NewtonScript programming. *PIE Developers*, January 1994. Also available as `ftp://ftp.apple.com/pub/wrs/class-based-NS.ps`.

[14] Walter R. Smith. The Newton application architecture. In *Proceedings of the 39th IEEE Computer Society International Conference*, pp. 156–161, San Francisco, 1994. Also available as `ftp://ftp.apple.com/pie/newton/articles/COMPCON-Arch.ps`.

[15] David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Holzle. Organizing programs without classes. Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June 1991.

[16] David Ungar and Randall B. Smith. Self: the power of simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227–241, Orlando, Florida, 1987. Published as *SIGPLAN Notices 22*, 12, December 1987.