To appear in Proceedings of the 1994 IEEE Computer Conference, San Francisco. Copyright © 1994 IEEE

The Newton Application Architecture

Walter R. Smith
Apple Computer, Inc.
Cupertino, California 95014
wrs@apple.com

Abstract

The application architecture of Newton, a technology for supporting Personal Digital Assistants (PDAs), is described. It combines a dynamic, object-oriented language called NewtonScript with a hierarchical view system and a persistent object store. It also contains extensible subsystems for communications and user-input recognition. In addition to PDA devices, the portable implementation has been used to build supporting tools for desktop machines.

1. Introduction

Newton™ technology was designed to support a new type of computing device, the "personal digital assistant". PDAs are intended to help people capture, organize, and communicate information more effectively. They are very personal machines, adapting to the needs of individual users, focused on the functionality required for an individual's tasks. Physically, they are small, light, and inexpensive, and their limited computing resources must be carefully conserved.

This paper focuses on the higher levels of Newton software, collectively known as the application architecture. These components of the system are the most visible to users and to application programmers, and as a group are largely independent of the underlying operating system.

2. Overview

The bulk of the application architecture is found in four interrelated components. The *object system* implements the data representation model used by the other pieces. *NewtonScript* is a language defined and implemented using the object system. The *view system* handles screen display and user input, using the object system and NewtonScript to represent views and define application behavior. Finally, the *object store* is used by applications to store and retrieve objects.

The other parts of the architecture are more independent. There is a flexible and extensible communications system. The recognition architecture, also extensible, interprets user input. Add-on software is delivered in the form of *packages* that combine multiple pieces of software into a single unit.

Most of the implementation is highly portable. We have re-used parts of the architecture to construct a set of tools that run on computers using the Microsoft Windows and Macintosh operating environments. These tools are used by end users, programmers, and content developers.

3. Object system

The object system is used by other parts of the system to represent data. The data model is similar to that found in many dynamic language runtime systems [1]. The basic element is a 32-bit *value*, which may be an "immediate" object such as an integer or character, or a pointer to a heap object. A few low-order bits are reserved for tags that distinguish between the different kinds of values.

The primitive object types include integers, characters, floating-point numbers, symbols (as found in Lisp and Smalltalk), and Unicode strings [5]. A "binary" object type is used for sounds, pictures, and other large pointerless objects. More complicated structures can be created using heap objects called *arrays* and *frames*. An array is an ordered list of slots, numbered consecutively from zero, each holding a value. A frame is a list of *tags* (which are symbols) and associated values. Combinations of these fundamental object types can represent a wide variety of data structures. (See Fig. 1.)

Objects are self-describing. That is, given a value, it is possible to determine all of its characteristics at runtime. In addition to the obvious advantages for debugging, this feature allows more powerful programming. For example,

Newton and MessagePad are trademarks of Apple Computer, Inc.

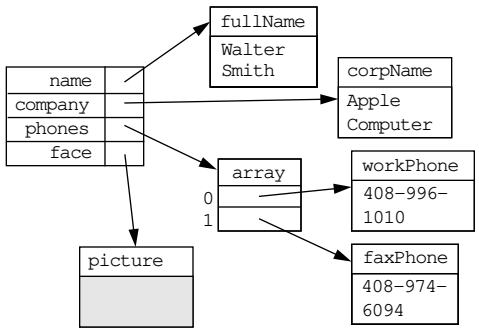

Figure 1. An example of the object model, showing one frame, one array, and five binary objects.

```
func (soup) begin
  local max := nil;
  local cursor :=
    Query(soup, '{type: index});
  while cursor:Entry() do begin
    local e := cursor:Entry();
    if not max or e.price > max then
      max := e.price;
    cursor:Next()
  end;
  max
end
```

Figure 2. A tiny NewtonScript example. This function finds the highest value from the **`price`** *slots of all the entries in the given soup.*

a function may take a sound or an array of sounds, in the latter case playing all the sounds in the array.

Binary objects and arrays have a special "type" slot that must contain a symbol reference. This slot is used to attach type information to these objects. For example, a picture in Macintosh PICT format is represented as a binary object whose type slot contains the symbol **`picture`**. This slot is needed to make such objects self-describing; without such identifying information, they would be identifiable only by context. Type symbols are arranged in a simple hierarchy.

Because objects are uniform and self-describing, the system can locate all object references, which allows it to perform automatic memory management. At various times, such as when an object allocation would fail for lack of memory, the system performs a garbage collection that frees memory occupied by objects that cannot be legitimately accessed, due to the lack of references to them. The layout of the objects in memory is kept simple, to make algorithms that must scan objects for pointers, such as garbage collection, simple and fast.

4. NewtonScript

All Newton applications are written in a dynamic object-oriented language called NewtonScript. The syntax of NewtonScript is similar to Pascal. It uses infix notation for built-in math and comparison operators, **`begin`** and **`end`** for compound statements, and has the standard Pascal control structures (such as **`if`**/**`then`**, **`while`**, **`repeat`**, and **`for`**). (See Fig. 2.) However, it is a dynamic language whose semantics are similar to Scheme [7]: functions are first-class objects, all statements have a return value, variables are untyped, and it has upward closures (but not continuations). In addition, it has features for object-oriented programming.

The language is strongly tied to the Newton object system. It includes operators and syntax features that make it easy to create and manipulate Newton objects. More significantly, frames are the basis for inheritance and message passing.

NewtonScript's object-oriented features are in some ways a simplified version of Self [2], with frames taking the role of Self objects. A frame is the only kind of object that can respond to messages and serve as the context of a method execution. Like Self, inheritance is prototype-based: there are no classes, and objects inherit directly from other objects. However, variable references are distinct from message sends, and inheritance is limited to two paths with fixed priority.

The inheritance system looks for two slot tags, **`_proto`** and **`_parent`**, which define the inheritance relationship between frames. The names are chosen to reflect the convention that **`_parent`** frames are containers, frames holding shared data, while **`_proto`** frames are prototypes, frames to be refined by other frames. When a message is sent to a frame, or a variable reference is resolved, the interpreter looks for a matching tag in the receiver frame. If it is not found, the lookup tries the frames in the receiver's **`_proto`** chain, then goes up to the **`_parent`** frame and tries its **`_proto`** chain, and so on.

A slightly different rule applies to variable assignment. The same lookup occurs to locate the frame containing the variable, but the assignment always occurs in the **`_parent`** chain, not in a **`_proto`** chain, creating a new slot if necessary. This assignment rule makes it difficult to accidentally modify a prototype frame, which might be shared by many frames. More importantly, it provides a form of "copy-on-write": the prototypes may provide an initial value for a slot, which is overridden in a particular frame by the first assignment. This delays the allocation of the slot as long as possible, saving space. (See Fig. 3.)

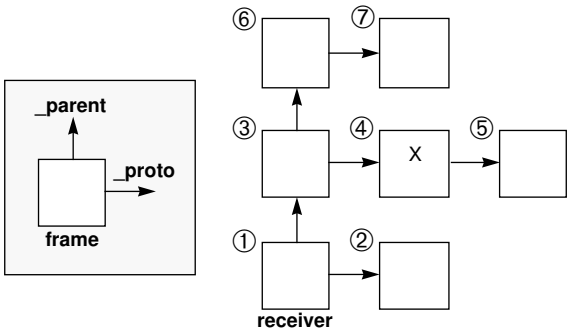

Figure 3. NewtonScript inheritance. Lookup proceeds from the receiver in the order shown. Assignment to the variable X will create a slot X in frame ③.

NewtonScript is compiled into bytecodes, which are interpreted at runtime. This has two advantages: the bytecodes are significantly smaller than native code, and they make Newton applications inherently portable to any processor. Because the Newton technology is widely licensed, and expected to migrate to a variety of devices, these are important properties. However, some applications cannot tolerate the speed penalty of bytecode interpretation. We are working on a native-code compiler for NewtonScript that will allow the programmer to selectively pay the space penalty for the parts of an application for which speed is critical. Bytecodes will still be generated, so the code will retain portability.

5. View system

The view system is used to define the appearance and behavior of application software. It is similar in many respects to the view systems in Smalltalk-80 [3] and MacApp [4]. Briefly, an application's user interface is decomposed into *views*, which are rectangular areas mapped into screen space. Each view displays itself on the screen and reacts to user input in its space, under the management of the view system. A view can contain other views, so they form a hierarchy.

In Newton's view system, each view is defined by a frame. The characteristics of the view are defined by the slots of the frame. For example, the bounds of the view are determined by its **`viewBounds`** slot. The view system translates events into NewtonScript messages to the view frames.

NewtonScript's inheritance system is used to advantage in the view system. When a view and its subviews are activated, the **`_parent`** slot of each view is set to its enclosing view. This allows variables and view characteristics to be inherited from enclosing views through the **`_parent`** chain.

The **`_proto`** chain is used to share common view features. A *view template* may be constructed that contains some or all of the slots necessary for a certain kind of view, such as a button. Specific views may then have a **`_proto`** slot referring to the template, thus inheriting its characteristics, and additional slots parameterizing the template, such as a button's title. Some templates may be located in the system ROM, where they can be shared by all applications, and each application may contain templates for its own views to share.

`_Proto` inheritance is also used to reduce RAM usage. View frames are normally located in ROM or in read-only package space (see "Packages" below). Rather than cloning them into RAM so their **`_parent`** slots can be set, the view system creates a small frame in RAM for each view. This frame initially contains only the **`_parent`** slot and a **`_proto`** slot referring to the view

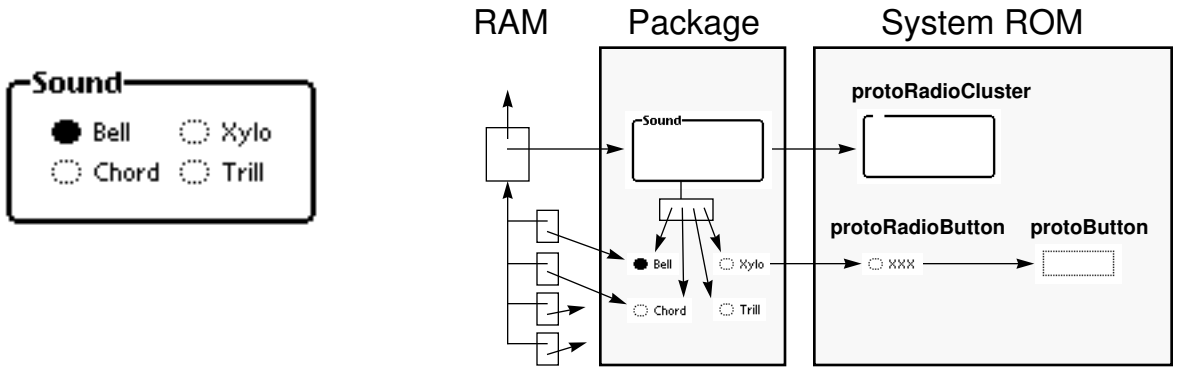

Figure 4. A view containing four subviews, and its underlying Newton objects. The views are represented by the small RAM frames on the left, containing only **`_parent`** *and* **`_proto`** *pointers. The RAM frames'* **`_proto`** *slots reference the view frames in the application package, whose* **`_proto`** *slots reference view templates in the system ROM.*

frame. When view slots are changed by variable assignment, the standard NewtonScript assignment mechanism will create new slots in the RAM frame to hold the changed values, thus delaying the use of RAM space as long as possible. (See Fig. 4.)

6. Object store

Newton devices can have a variety of storage systems. For example, the Apple MessagePad™ reserves a portion of its internal RAM as a permanent, protected user data store. PCMCIA cards can extend that storage with RAM, Flash RAM, or ROM devices. All of these media are accessed through a single high-level interface called the object store.

The object store imposes the uniform Newton object model on persistent storage devices. Each physical device (internal RAM, PCMCIA card, etc.) contains a *store*. Each store contains one or more *soups*, which are collections of related data items called *entries*. Each entry is structured as a frame. Of course, the frame may refer to other objects, which may refer to others, and so on; the entry includes the entire set of objects reachable from the top-level frame.

Each soup has a set of *indexes*, which are defined by a data type (integer, string, etc.) and a path from the top level of an entry to the indexed data item. For example, a soup for inventory items might have an integer index on the "item number" slot of its entries. The indexes can be used in a *query*, which specifies an index and a set of constraints and results in a *cursor*. A cursor is an object that represents a position in the set of entries matched by the query. It can be moved back and forth, and can deliver the current entry.

An entry acts exactly like a regular frame in all respects, except that it has a few more valid operations to find out what soup it came from, write out changes to the soup, and so forth. This transparency makes it easy to manipulate persistent data. To minimize RAM usage, *fault blocks*, small objects containing only enough information to locate an entry in the storage medium, are used to represent entries. The system does not actually read in and decompress an entry until the first access to one of its slots.

The usual behavior of a Newton application is to present a merged view of the data in the internal store with data on a PCMCIA card. The data on a card is regarded as an extension to the internal data. The object store makes this easier by providing *union soups*, which are virtual soups—objects with the same interface as soups—that automatically merge the data from a set of real soups.

The object store has many advantages over a more traditional filesystem. Applications do not need to implement their own methods of translating objects between runtime form and persistent form, so they are smaller and easier to write. It is very easy for Newton applications to share data, since they all get the same high-level interface to it; we encourage developers to share their persistent data structures to foster the creation of highly cooperative applications.

The Newton object store was inspired by a number of research projects, notably PS-Algol [8] and Persistent Smalltalk [9].

7. Communications

The ability to communicate with other devices is critical to many PDA applications. Newton's communication framework is designed to provide uniform access to communication services, and to allow new protocols and services to be installed and removed dynamically.

Communications services are accessed through objects called *endpoints*. When an endpoint is created, all information necessary to construct and connect it is specified through a set of *options*. For example, the options might specify an ADSP (AppleTalk Data Stream Protocol) connection to a certain address and socket number, or a modem connection at 9600 baud using MNP 5 to a certain phone number. Once the connection is made, data can be sent and received over the endpoint using a set of operations common to all endpoints. Separating the protocol-dependent connection process from the largely independent data-moving operations simplifies the interface as well as the client applications.

Application developers use endpoints through a NewtonScript interface. This interface is somewhat similar to the view system, in that endpoints are represented by frames. An endpoint frame specifies the options for the connection, as well as actions to be performed when certain events occur to the endpoint. The inheritance system allows common features to be shared between endpoint frames.

Protocol developers use a low-level interface to add new kinds of endpoints. New protocols can be installed and removed at runtime, and are looked up dynamically when an endpoint is created. This ability is useful for PCMCIA cards that contain communications hardware. For example, the Apple Messaging Card is a PCMCIA card that contains an alphanumeric pager and a ROM with a Newton communication driver and application. The pager has its own power supply and processor and can receive and store messages independently. When the card is inserted into a Newton, a protocol on the card is installed that allows access to the pager interface from applications.

8. Recognition

User input is managed by a powerful recognition architecture that can arbitrate between several recognizers operating simultaneously. For example, the MessagePad contains a text recognizer that can handle printed, cursive, or mixed handwriting; a graphics recognizer that looks for lines, curves, and symmetries; and a recognizer for gestures such as scrubs and carets. All of these recognizers can be examining the input from the tablet at the same time. The match with the highest confidence is dispatched to the appropriate view. Applications can activate and parameterize recognition on a view-by-view basis. Also, the recognition architecture is extensible: new recognizers can be added and existing ones replaced.

9. Application structure

A Newton application usually consists of at least one top-level view, with many subviews. When the application is installed, the top-level view is made a subview of the "root view", which is the view that contains everything that appears on the screen. Opening the application is done simply by showing the application's top-level view.

The definition of NewtonScript does not allow the possibility of certain bugs endemic to low-level languages like C—dangling pointers, pointer aliasing, multiple deallocation, out-of-bounds array references—that can result in arbitrary corruption of the heap or stack. Thus, Newton does not need the usual solution to such bugs: walling off applications into their own heaps, stacks, or even entire address spaces.

All Newton applications run in the same address space. In fact, they are part of the same "meta-application", since they are just subviews of the root view. This simplifies the system structure considerably. There are no "windows", just views that look like windows, so there is no need for a separate "window manager". Applications can communicate simply by sending NewtonScript messages to each other, or to objects whose pointers they put in a public place, so there is no need for an inter-application communication system. Code, data, and view templates can be freely shared between applications just by passing pointers around.

Newton software is delivered in the form of a *package*. Each package contains one or more *parts*, which are independent pieces that are dispatched to the appropriate part of the system when the package is installed. For example, a package for a communication application might contain three parts: the application itself, a font used by the application, and a communication driver.

Packages are stored in the object store, but the system uses them as if they were resident in memory. This allows the use of direct pointers to objects embedded in the package, from objects in RAM as well as other objects in the package. Each package is assigned a region of virtual memory and is paged in as necessary. Because storage is very limited, the pages are usually compressed in the store and decompressed on demand.

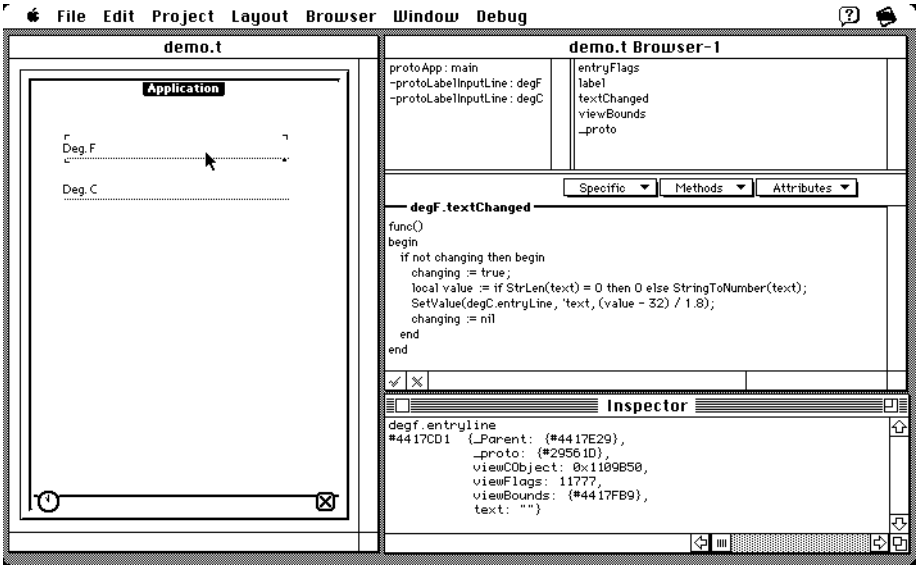

Figure 5. Newton Toolkit. The window on the left shows an application's views in graphic form; they can be moved and resized with the mouse. The upper-right window is a browser on the view hierarchy, currently showing the NewtonScript source for a view method. The "Inspector" window allows NewtonScript expressions to be executed on the Newton device.

10. Tools

The object system, the compiler and interpreter for NewtonScript, and the object store are portable by design, and are currently working on Macintosh and Windows platforms in addition to Newton devices. We have used these subsystems as the basis of several tools.

Newton Toolkit (or "NTK") is the development environment for Newton applications. (See Fig. 5.) It gives the programmer a graphical interface to edit the visual aspects of the view hierarchy, as well as a browser to edit the slots of the view frames. The output of NTK is a package, which is downloaded to a Newton for execution. An "Inspector" window gives access to the NewtonScript interpreter in a Newton device, allowing live debugging and modification of a running application. The editor in NTK is partially written in NewtonScript, and can be extended with new commands written by the user.

Book Maker is an adjunct to NTK that produces digital books. It accepts files in common word processor file formats, and produces a data structure that can be built into a package by NTK and interpreted by a book reader built into the Newton system. The files can contain commands that specify pagination, hypertext-like links, embedded forms, and other features.

Newton Connection extends the reach of the Newton object store to desktop machines. It can maintain duplicate data stores on the desktop, periodically "synchronizing" with the data on a Newton device—changes are propagated in both directions on an entry-by-entry basis. It also contains editors that allow the desktop user to modify the data using the native interface.

11. Conclusion

Newton combines many advanced software technologies. It is the first widely-available platform with a dynamic object-oriented language, view system, and persistent object store built into the basic system software. Nevertheless, the highly integrated nature of the design gives it a surprisingly simple structure—a small number of concepts used in different ways. Our experience so far shows that the system and tools are relatively easy to learn, and that programmer productivity is much higher than with more traditional systems. We hope to see Newton become a standard platform for a new PDA industry.

We would also like to promote a broader software market. In desktop computing, there is a strong trend toward huge, monolithic application programs whose vast functionality is largely ignored by any individual user. It is difficult for the smaller, perhaps more innovative, software maker to compete and survive in such an environment. The Newton architecture and tools make it easier to create and distribute small, focused applications quickly, opening up opportunities for smaller vendors in the PDA software arena.

12. Bibliography

[1] Andrew W. Appel. A runtime system. *Lisp and Symbolic Computation 3*, 1990, 343–380.

[2] *Programmer's Guide to MacApp*. Developer Technical Publications 030-1937-A, Apple Computer, Inc., 1992.

[3] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices 17*, 7 (July 1981).

[4] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[5] Anthony L. Hosking, J. Eliot B. Moss, and Cynthia Bliss. Design of an object faulting persistent Smalltalk. COINS Technical Report 90-45, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, May 1990.

[6] J. Rees and W. Clinger, editors. The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices 21*, 12 (December 1986).

[7] 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.

[8] The Unicode Consortium. *The Unicode Standard: Worldwide Character Encoding, Version 1.0.* Addison-Wesley, 1991.

[9] N. Wirth. The programming language PASCAL. *Acta Informatica 6*, 4, 35–63.