

# AppleScript

William R. Cook

University of Texas at Austin

wcook@cs.utexas.edu

With contributions from Warren Harris, Kurt Piersol, Dave Curbow, Donn Denman, Edmund Lai, Ron Lichty, Larry Tesler, Donald Olson, Mitchell Gass and Eric House

## Abstract

AppleScript is a scripting language and environment for the Mac OS. Originally conceived in 1989, AppleScript allows end users to *automate* complex tasks and *customize* Mac OS applications. To automate tasks, AppleScript provides standard programming language features (control flow, variables, data structures) and sends Apple Events to invoke application behavior. Apple Events are a variation on standard remote procedure calls in which messages can identify their arguments by queries that are interpreted by the remote application. This approach avoids the need for remote object pointers or proxies, and reduces the number of communication round trips, which are expensive in high latency environments like the early Macintosh OS. To customize an application that uses AppleScript's Open Scripting Architecture, users attach scripts to application objects; these scripts can then intercept and modify application behavior.

AppleScript was designed for casual users: AppleScript syntax resembles natural language, and scripts can be created easily by recording manual operations on a graphical interface. AppleScript also supported internationalization in allowing script to be presented in multiple dialects, including English, Japanese, or French. Although the naturalistic syntax is easy to read, it can make scripts much more difficult to write.

Early adoption was hindered by the difficulty of modifying applications to support Apple Events and the Open Scripting Architecture. Yet AppleScript is now widely used and is an essential differentiator of the Mac OS. AppleScript's communication model is a precursor to web services, and the idea of embedded scripting has been widely adopted.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, New York, NY 11201-0701, USA, fax:+1(212) 869-0481, [permissions@acm.org](mailto:permissions@acm.org)

©2007 ACM 978-1-59593-766-7/2007/06-ART1 \$5.00

DOI 10.1145/1238844.1238845

<http://doi.acm.org/10.1145/1238844.1238845>

*Categories and Subject Descriptors* D.3 [Programming Languages]

*General Terms* Languages, Design, Human Factors

*Keywords* AppleScript, Scripting, History

## 1. Introduction

The development of AppleScript was a long and complex process that spanned multiple teams, experienced several false starts and changes of plan, and required coordination between different projects and companies. It is difficult for any one person, or even a small group of people, to present a comprehensive history of such a project, especially without official support from the company for which the work was done. The email record of the team's communications have been lost, and the author no longer has access to internal specifications and product plans.

Nevertheless, I believe that the development of AppleScript is a story worth telling, and I have been encouraged to attempt it despite the inherent difficulty of the task. I can offer only my own subjective views on the project, as someone who was intimately involved with all its aspects. I apologize in advance for errors and distortions that I will inevitably introduce into the story, in spite of my best efforts to be accurate.

I first heard the idea of AppleScript over lunch with Kurt Piersol in February of 1991. The meeting was arranged by our mutual friend James Redfern. I knew James from Brown, where he was finishing his undergraduate degree after some time off, and I was working on my PhD. James and I both moved to California at about the same time, in 1988. We spent a lot of time together and I had heard a little about what he was up to, but he claimed it was secret. James arranged the meeting because Kurt was looking for someone to lead the AppleScript effort, and I was looking for something new to do.

For the previous two and a half years I had been working at HP Labs. I was a member of the Abel group, which included Walt Hill, Warren Harris, Peter Canning, and Walter Olthoff. John Mitchell consulted with us from Stanford. The group was managed by Alan Snyder, whose formaliza-

tion of object concepts [35] was one basis for the development of CORBA. At HP Labs I finished writing my PhD thesis, *A Denotational Semantics of Inheritance* [14, 19], which the Abel group used as the foundation for a number of papers. We published papers on inheritance and subtyping [18], object-oriented abstraction [7, 16], mixins [5], F-Bounded polymorphism [6], and a fundamental flaw in the Eiffel type system [15]. I continued some of the work, analyzing the Smalltalk collection hierarchy [17], after I left HP.

Near the end of 1990 HP Labs was undergoing a reorganization and I was not sure I would fit into the new plan. In addition, I was interested in making a change. I had developed several fairly large systems as an undergraduate, including a text editor and a graphical software design tool [39, 20] that were used by hundreds of other students for many years. As a graduate student I had focused on theory, but I wanted to learn the process of commercial software development. Apple seemed like a reasonable place to try, so I agreed to talk with Kurt.

### 1.1 AppleScript Vision—Over Lunch

Kurt and I hit it off immediately at lunch. Kurt Piersol is a large, friendly, eloquent man who was looking for people to work on the AppleScript project. Kurt graduated with a B.S. from the University of Louisville’s Speed Scientific School. He then worked at Xerox, building systems in Smalltalk-80 and productizing research systems from Xerox PARC. Kurt was hired to work on AppleScript in 1989. He was originally asked to work on a development environment for the new language, but eventually took on the role of steering the project.

Kurt and I discussed the advantages or disadvantages of command-line versus graphical user interfaces. With command-line interfaces, commonly used on Unix, users frequently write scripts that automate repeated sequences of program executions. The ability to pipe the output of one program into another program is a simple but powerful form of inter-application communication that allows small programs to be integrated to perform larger tasks. For example, one can write a script that sends a customized version of a text file to a list of users. The sed stream editor can create the customized text file, which is then piped into the mail command for delivery. This new script can be saved as a mail-merge command, so that it is available for manual execution or invocation from other scripts. One appealing aspect of this model is its compositionality: users can create new commands that are invoked in the same way as built-in commands. This approach works well when atomic commands all operate on a common data structure, in this case text streams. It was not obvious that it would work for more complex structured data, like images, databases, or office documents, or for long-running programs that interact with users.

With a graphical user interface (GUI) important functions, including the mail-merge command described above,

are usually built into a larger product, e.g. a word processor. A GUI application offers pre-packaged integrated functionality, so users need not combine basic commands to perform common tasks. Although careful design of graphical interfaces eliminates the need for automation for basic use, there are still many tasks that users perform repeatedly within a graphical interface. The designers of the graphical interface cannot include commands to cover all these situations — if they did, then some users would execute these new commands repeatedly. No finite set of commands can ever satisfy all situations.

Most users are happy with GUI applications and do not need a command-line interface or a scripting language. But there are clearly some limitations to the GUI approach on which Macintosh OS was based. Power users and system integrators were frustrated by the inability to build custom solutions by assembling multiple applications and specializing them to particular tasks. Allowing users to automate the tasks that are relevant to them relieves pressure on the graphical user interface to include more and more specialized features.

The vision for AppleScript was to provide a kind of command-line interface to augment the power of GUI applications and to bring this power to casual users.

### 1.2 Automation and Customization

Kurt and I talked about two ways in which scripts and GUI applications could interact: for automation and for customization. *Automation* means that a script directs an application to perform a sequence of actions—the actions are performed “automatically” rather than manually. With automation, the script is in control and one or more applications respond to script requests. *Customization* occurs when a script is invoked from within an application—the script can perform “custom” actions that replace or augment the normal application behavior. With customization, the application manages and invokes scripts that users have attached to application objects. Automation is useful even without customization, but customization requires automation to be useful.

We discussed whether there was sufficient benefit in providing a standard platform for scripting, when custom solutions for each application might be better. Some applications already had their own macro capability or a proprietary scripting language. However, this approach requires users to learn a different scripting language to automate each application. These languages typically include some variation on the basic elements of any programming language, including variables, loops, conditionals, data types, procedures, and exceptions. In addition, they may include special forms or constructs specific to the application in question. For example, a spreadsheet language can refer to cells, sheets, equations and evaluation.

One benefit of a standard scripting platform is that applications can then be *integrated* with each other. This capa-

bility is important because users typically work with multiple applications at the same time. In 1990, the user's options for integrating applications on the Macintosh OS were limited to shared files or copy/paste with the clipboard. If a repeated task involves multiple applications, there is little hope that one application will implement a single command to perform the action. Integrating graphical applications can be done at several levels: visual integration, behavioral integration, or data integration. *Visual integration* involves embedding one graphical component inside another; examples include a running Java applet inside a web page, or a specialized organization chart component displayed inside a word-processing document. *Behavioral integration* occurs when two components communicate with each other; examples include workflow or invocation of a spell-check component from within a word processor. *Data integration* occurs whenever one application reads a file (or clipboard data) written by another application. A given system can include aspects of all three forms of integration.

I agreed with Kurt that the most important need at that time was for behavioral integration. To compete with custom application-specific scripting languages, AppleScript would have to allow application-specific behaviors to be incorporated into the language in as natural a way as possible, while preserving the benefits of a common underlying language. The core of the language should support the standard features of a programming language, including variables, procedures, and basic data types. An application then provides a vocabulary of specific terminology that apply to the domain: a photo-processing application would manipulate images, pixels and colors, while a spreadsheet application would manipulate cells, sheets, and graphs. The idea of AppleScript was to implement the "computer science boilerplate" once, while seamlessly integrating the vocabulary of the application domain so that users of the language can manipulate domain objects naturally. We discussed the vision of AppleScript as a pervasive architecture for inter-application communication, so that it is easy to integrate multiple applications with a script, or invoke the functionality of one application from a script in another application. We hoped that scripting would create a "network effect", by which each new scriptable application improves the value of scripting for all other applications.

### 1.3 AppleScript Begins

Soon after, Kurt offered me a job and I accepted quickly. This event illustrates one of the recurring characteristics of AppleScript: the basic idea is so compelling that it is enthusiastically embraced by almost every software developer who is exposed to it.

What was not immediately obvious was how difficult the vision was to achieve—not for strictly technical reasons, but because AppleScript required a fundamental refactoring, or at least augmentation, of almost the entire Macintosh code base. The demonstrable benefits of AppleScript's vision has

led developers to persevere in this massive task for the last twenty years; yet the work is truly Sisyphean, in that the slow incremental progress has been interrupted by major steps backward, first when the hardware was changed from the Motorola 68000 chip to the IBM PowerPC, and again when the operating system was reimplemented for Mac OS X.

At this point it is impossible to identify one individual as the originator of the AppleScript vision. The basic idea is simple and has probably been independently discovered many times. The AppleScript team was successful in elaborating the original vision into a practical system used by millions of people around the world.

## 2. Background

When I started working at Apple in April 1991 I had never used a Macintosh computer. My first task was to understand the background and context in which AppleScript had to be built.

The main influences I studied were the Macintosh operating system, HyperCard, and Apple Events. HyperCard was a good source of inspiration because it was a flexible application development environment with a scripting language embedded within it. A previous team had designed and implemented Apple Events to serve as the underlying mechanism for inter-application communication. The Apple Events Manager had to be shipped early so that it could be included in the Macintosh System 7 OS planned for summer 1991. When I started at Apple, the Apple Event Manager was in final beta testing. The fact that AppleScript and Apple Events were not designed together proved to be a continuing source of difficulties.

Macintosh systems at that time had 4 to 8 megabytes of random-access memory (RAM) and a 40- to 60-megabyte hard drive. They had 25-50 MHz Motorola 68000 series processors. The entire company was internally testing System 7.0, a major revision of the Macintosh OS.

Applications on the Macintosh OS were designed around a main event processing loop, which handled lower-level keyboard and mouse events from the operating system [12]. The OS allowed an application to post a low-level event to another application, providing a simple form of inter-application communication. In this way one application could drive another application, by sending synthetic mouse and keyboard events that select menus or data and enter text into an application. This technique was used in two utility applications, MacroMaker and QuicKeys, which recorded and played back low-level user interface events. It was also used in the Macintosh help system, which could post low-level events to show the user how to use the system. Scripts that send low-level events are fragile, because they can fail if the position or size of user interface elements changes between the time the script is recorded and when it is run. They are also limited in the actions they can perform; low-

level events can be used to change the format of the current cell in a spreadsheet, but cannot read the contents of a cell.

In the following section I describe these systems as they were described to me at the start of the AppleScript project, in April 1991.

## 2.1 HyperCard

HyperCard [27, 30], originally released in 1987, was the most direct influence on AppleScript. HyperCard is a combination of a simple database, a collection of user interface widgets, and an English-like scripting language. These elements are organized around the metaphor of information on a collection of index cards. A collection of such cards is called a *stack*. A card could contain text, pictures, buttons, and other graphical objects. Each object has many *properties*, including location on the card, size, font, style, etc. Cards with similar structure could use a common *background*; a background defines the structure, but not the content, of multiple cards. For example, a stack for household information might contain recipe cards and contact cards. The recipe cards use a recipe background that includes text fields for the recipe title, ingredients, and steps. The contact cards use a contact background with appropriate fields, including a photo.

HyperCard scripts are written in HyperTalk, an English-like scripting language [28]. The language is for the most part a standard structured, imperative programming language. However, it introduced a unique approach to data structures: the stack, cards, objects and properties are used to store data. These structures are organized in a containment hierarchy: stacks contain cards, cards contain objects, and properties exist on stacks, cards, and objects. This predefined structure makes it easy to build simple stacks, but more difficult to create custom data structures.

Scripts in a stack can refer to the objects, collections of objects, and their properties by using *chunk expressions*. A chunk expression is best understood as a kind of query. For example, the following chunk expression refers to the text style property of a word element in a field of the current card:

```
the textStyle of word 2  
of card field "Description"
```

A chunk expression can refer to properties and elements of objects. A property is a single-valued attribute, for example `textStyle`. Elements are collections of objects identified by a type, for example `word` and `card field`. Element access may be followed by a name, index or range to select element(s) from the collection. Properties access distributes over collections; the following expression represents a collection of 10 text style properties:

```
the textStyle of character 1 to 10  
of card field "Description"
```

HyperCard has a built-in set of property and collection names.

Each object has a *script* containing procedures defined for that object. If the procedure name is an *event* name, then the procedure is a *handler* for that event—it is called when the event occurs for that object. For example, a button script may have handlers for `mouseDown`, `mouseUp` and `mouseMove` events. The following handler shows the next card when a button is released.

```
on mouseUp  
go to next card  
end mouseUp
```

Actions can be performed on chunk expressions to modify the stack, its cards, the objects, or their properties. For example, clicking a button may run a script that moves to the next card, adds/removes cards, or modifies the contents of one or more cards. HyperCard has a set of predefined actions, including **set**, **go**, **add**, **close**, etc. For example, the text style can be updated to a predefined constant bold:

```
set the textStyle of character 1 to 10  
of card field "Description" to bold
```

HyperCard 2.0 was released in 1990. HyperCard was very influential and widely used. Developers could easily create some applications in HyperCard, but to create more complex applications, they had to switch to more difficult general-purpose development tools. The need for unification of these approaches was recognized early at Apple, leading to the formation of a research and development project to build a new development platform for the Mac, discussed in the next section. Looking forward, the rapid development capabilities pioneered by HyperCard were added to more sophisticated general-purpose development environments. This gradually reduced the need for systems like HyperCard, which was discontinued in 2004.

## 2.2 Family Farm

Many of the ideas that eventually emerged in AppleScript were initially explored as part of a research project code-named Family Farm, which was undertaken in the Advanced Technology Group (ATG) at Apple, starting in 1989. The research team was led by Larry Tesler and included Mike Farr, Mitchell Gass, Mike Gough, Jed Harris, Al Hoffman, Ruben Kleiman, Edmund Lai, and Frank Ludolph. Larry received a B.S. in Mathematics from Stanford University in 1965. In 1963, he founded and ran IPC, one of the first software development firms in Palo Alto, CA. From 1968-73, he did research at the Stanford A.I. Lab on cognitive modeling and natural language understanding, where he designed and developed PUB, one of the first markup languages with embedded tags and scripting. From 1973-80, he was a researcher at Xerox PARC, working on object-oriented languages, user interfaces, and desktop publishing. He joined Apple in 1980

to work on the Lisa. In 1986, he was named director of Advanced Development, and in 1987, the first VP of Advanced Technology, a new group focused on research.

The original goal of Family Farm was to create a new integrated development environment for the Macintosh OS. Family Farm included work on a new system-level programming language, an interprocess communication model, a user-level scripting language, and object component models, in addition to other areas.

The first AppleScript specification, which was the foundation for later development, was written by the Family Farm team. This document defined the basic approach to generalizing HyperTalk chunk expressions to create AppleScript object-specifiers and Apple Events (described in detail below). I believe credit for these ideas must be shared equally by the Family Farm team, which generated many ideas, and the teams which turned these ideas into usable systems.

As a research project that grew quickly, the organization put in place for Family Farm turned out not to be sufficient to build and deliver commercial products. The team was in the research group, not the system software group; No changes to the Macintosh system could be shipped to customers without approval of the system software group. And if Family Farm did get approval, they would have to follow the strict software development, scheduling, and quality control processes enforced by the system software group. Over time it became clear that the Family Farm team was also too small to achieve its vision.

After about a year and a half of work, the Family Farm project was disbanded, and new teams were created to design and implement some of the concepts investigated by Family Farm.

One of the main themes to emerge from Family Farm was a focus on techniques for *integrating* applications. As mentioned in Section 1.2, integrating graphical applications can be done at several levels: visual embedding, behavioral coordination, or data exchange. The spin-off projects were Apple Events, AppleScript, and OpenDoc. The Apple Events project, formed in mid-1990 from a subset of the Family Farm team, developed the underlying communication model on which later projects were based. Later projects involved larger teams that pulled key members from outside Family Farm. AppleScript was next, then OpenDoc, both within the Developer Tools Group at Apple. AppleScript focused on data and behavioral integration. The OpenDoc project, which is not discussed in detail here, focused on visual integration by embedding components. Family Farm's transition from research to product development was a difficult one; in the end the primary product transferred from Family Farm to its descendants was an inspiring vision.

### 2.3 Apple Event Manager

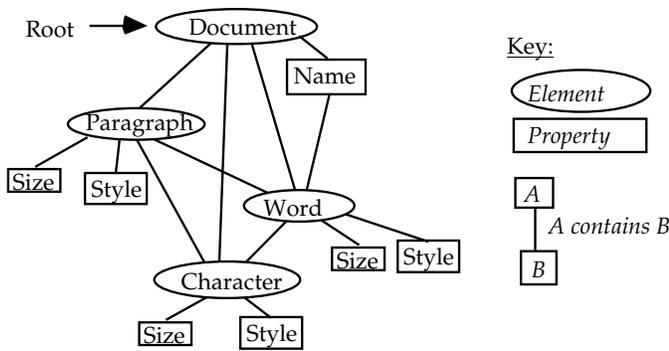
The Apple Event Manager provides an inter-application communication platform for the Macintosh. It was designed

with scripting in mind—however, the design was completed before development of AppleScript began. When I started at Apple in April 1991, my first job was to do a complete code review of Apple Events, which was nearing the end of its beta testing period. I sat in a conference room with Ed Lai for over a week reading the code line by line and also jumping around to check assumptions and review interrelationships. Ed was the primary developer of Apple Events code. The system was written in Pascal, as was most of the Macintosh system software of that era. The Apple Events team was part of the Developer Tools group and was originally managed by Larry Tesler (who was also still VP in ATG), but was later taken over by Kurt Piersol.

In designing Apple Events, Kurt, Ed and the team had to confront a serious limitation of the Macintosh OS: in 1990, the Macintosh OS could switch processes no more than 60 times a second. If a process performed only a few instructions before requesting a switch, the machine would idle until 1/60th of a second had elapsed. A fine-grained communication model, at the level of individual procedure or method calls between remote objects, would be far too slow: while a script within a single application could easily call thousands of methods in a fraction of a second, it would take several seconds to perform the same script if every method call required a remote message and process switch. As a result of this limitation of the OS, traditional remote procedure calls (RPC) could not be used. Fine-grained RPC was used in CORBA and COM, which were being developed at the same time.

The Macintosh OS needed a communication model that allowed objects in remote applications to be manipulated without requiring many process round-trips. The Apple Events communication model uses a generalization of HyperCard's chunk expressions. Just as a HyperCard command contains a verb and one or more chunk expressions in its predefined internal language, an Apple Event contains a verb and a set of chunk expressions that refer to objects and properties in the target application. The generalized chunk expressions are called *object specifiers*. Apple Events address the process-switching bottleneck by making it natural to pack more behavior into a single message, thereby reducing the need for communication round-trips between processes. An Apple Event is in effect a small query/update program that is formulated in one application and then sent to another application for interpretation.

Kurt Piersol and Mike Farr debated whether there should be few commands that could operate on many objects, or a large number of specific commands as in traditional command-line interfaces. For example, on Unix there are different commands to delete print jobs (`lprm`), directories (`rmdir`), and processes (`kill`). The analogy with verbs and nouns in English helped Kurt win the argument for few commands (verbs) that operate on general objects (nouns). For example, in AppleScript there is a single delete com-



**Figure 1.** The properties and elements in a simple object model.

mand that can delete paragraphs or characters from a word-processing document, or files from the file system. Having a small number of generic verbs, including set, copy, delete, insert, open and close, proved to be more extensible.

### 2.3.1 Object Specifiers

Object specifiers are symbolic references to objects in an application. One application can create object specifiers that refer to objects in another application, called the *target*. The specifiers can then be included in an Apple Event and sent to the target application. These symbolic references are interpreted by the target application to locate the actual remote objects. The target application then performs the action specified by the verb in the Apple Event upon the objects.

For example, a single Apple Event can be used to copy a paragraph from one location to another in a document; the source and target location are included in the event as object specifiers. The content of the paragraph remains local in the target application. Traditional RPC models would require the client to retrieve the paragraph contents, and then send it back to the target as a separate insertion operation.

Object specifiers provide a view of application data that includes *elements* and *properties*. An element name represents a collection of values; a property has a single value. The value of a property may be either a basic value, for example an integer or a string, or another object. Elements are always objects.

A simple object model is illustrated in Figure 1. A document has multiple paragraph elements and a name property. A paragraph has style and size properties and contains word and character elements.

The distinction between properties and elements is related to the concept of cardinality in entity-relationship modeling [9] and UML class diagrams [32]. Cardinality indicates the maximum number of objects that may be involved in a relationship. The most important distinction is between single-valued (cardinality of 1) relationships and multi-valued (cardinality greater than 1). The entity-relationship model also includes attributes, which identify scalar, prim-

itive data values. An AppleScript property is used for both attributes and single-valued relationships. Elements are used to describe multivalued relationships.

The name identifying a set of elements is called a *class* name, identifying a specific kind of contained object and/or its role. For example, a Family object might have elements parents and children, which are elements that refer to sets of Person objects. Object specifiers allow application-specific names for elements and properties, which generalize the fixed set of predefined names available in HyperCard.

Object specifiers also generalize HyperCard chunk expressions in other ways. One extension was the addition of *conditions* to select elements of a collection based on their properties. This extension made object specifiers a form of query language with significant expressive power.

Supporting Apple Events was frequently quite difficult. To create a new scriptable application, the software architect must design a scripting interface in addition to the traditional GUI interface. If the existing application architecture separates views (graphical presentations) from models (underlying information representation) [34], then adding scripting is usually possible. In this case the internal methods on the model may be exposed to scripts. There are two reasons why direct access to an existing object model may not be sufficient:

1. Users often want to control the user interface of an application, not just internal data of the application. Thus the scripting interface should provide access to both the view objects and the model objects.
2. The interfaces of the internal object model may not be suitable as an external scripting interface. In this case the scripting interface is usually implemented to provide another abstract view of the internal model.

Even with a good architecture, it can be difficult to retrofit an existing GUI application to include a second interface for scripting. If the application does not use a model-view architecture, then adding scripting is much more difficult.

The Apple Events team created a support library to assist application writers in interpreting object specifiers. It interprets nesting and conditions in the object specifiers, while using application callbacks to perform primitive property and element operations.

### 2.3.2 Apple Events Implementation

Object specifiers are represented in Apple Events as nested record structures, called *descriptors* [11]. Descriptors use a self-describing, tagged tree data structure designed to be easily transported or stored in a flattened binary format. Descriptors can either contain primitive data, a list of descriptors, or a labeled product of descriptors. Primitive data types include numbers (small and large integers and floats), pictures, styled and unstyled text, process IDs, files, and aliases. All the structures, types, and record fields are identified by

four-byte type codes. These codes are chosen to be human-readable to facilitate low-level debugging.

Each kind of object specifier is a record with fields for the class, property name, index, and container. The container is either another object specifier record or null, representing the default or root container of the application. Events may be sent synchronously or asynchronously. The default behavior of Apple Events is stateless—the server does not maintain state for each client session. However, Apple Events supports a simple form of transaction: multiple events can be tagged with a transaction ID, which requires an application to perform the events atomically or else signal an error.

Apple Events was first released with Macintosh System 7 in 1991. The entire Apple Events system was designed, implemented, and shipped in the Macintosh OS before any products using it were built. It was a complex problem: applications could not be built until the infrastructure existed, but the infrastructure could not be validated until many applications had used it. In the end, the Apple Events team adopted a “build it and they will come” approach. They designed the system as well as they could to meet predicted needs. Only a few small sample applications were developed to validate the model. In addition, the operating system team defined four standard events with a single simple parameter: open application, open documents, print documents, and quit. These first basic events did not use object specifiers; the open and print events used a vector of path names as arguments.

Later projects, including AppleScript, had to work around many of the Apple Events design choices that were made essentially within a vacuum. For example, Apple Events included some complex optimized message that were never used because they were too unwieldy. For example, if an array of values all has a common prefix, this prefix can be defined once and omitted in each element of the array. This was originally motivated by a desire to omit repetitive type information. This optimization is not used by AppleScript because it is difficult to detect when it could be used, and the reduction in message length provided by the optimization does not significantly affect performance.

### 3. The AppleScript Language

My primary task was to lead the design and implementation of the AppleScript language. After I decided to join Apple I mentioned the opportunity to Warren Harris. I enjoyed working with Warren at HP and thought he would be a great addition to the AppleScript effort. Warren has a BS and MS in EE from the University of Kansas. At HP Warren was still working on his “Abel Project Posthumous Report”, which contained all the ideas we had discussed, but had not time to complete, while working together at HP Labs [25]. Warren talked to Kurt and eventually decided to join the AppleScript team as a software engineer. He quickly became the co-architect and primary implementor of the language.

### 3.1 Requirements

AppleScript is intended to be used by all users of the Macintosh OS. This does not imply that all users would use AppleScript to the same degree or in the same way—there is clearly a wide range of sophistication in users of the Macintosh OS, and many of them have no interest in, or need for, learning even the simplest form of programming language. However, these users could *invoke* scripts created by other users, so there were important issues of packaging of scripts, in addition to developing them. More sophisticated users might be able to *record* or *modify* a script even if they could not write it. Finally, it should be possible for non-computer specialists to *write* scripts after some study.

The language was primarily aimed at *casual* programmers, a group consisting of programmers from all experience levels. What distinguishes casual programming is that it is infrequent and in service of some larger goal—the programmer is trying to get something else done, not create a program. Even an experienced software developer can be a casual programmer in AppleScript.

The team recognized that scripting is a form of programming and requires more study and thought than using a graphical interface. The team felt that the language should be easy enough to learn and use to make it accessible to non-programmers, and that such users would learn enough programming as they went along to do the things that they needed to do.

Programs were planned to be a few hundred lines long at most, written by one programmer and maintained by a series of programmers over widely spaced intervals. Scripts embedded inside an application were also expected to be small. In this case, the application provides the structure to interconnect scripts; one script may send messages to an application object that contains another script. Because scripts are small, compilation speed was not a significant concern. Readability was important because it was one way to check that scripts did not contain malicious code.

In the early '90s computer memory was still expensive enough that code size was a significant issue, so the AppleScript compiler and execution engine needed to be as small as possible. Portability was not an issue, since the language was specifically designed for the Macintosh OS. Performance of scripts within an application was not a significant concern, because complex operations like image filtering or database lookup would be performed by applications running native code, not by scripts. Due to the high latency of process switches needed for communication, optimization of communication costs was the priority.

### 3.2 Application Terminologies

An *application terminology* is a dictionary that defines the names of all the events, properties, and elements supported by an application. A terminology can define names as plural

```

reference ::=
  propertyName
  | 'beginning'
  | 'end'
  | 'before' term
  | 'after' term
  | 'some' singularClass
  | 'first' singularClass
  | 'last' singularClass
  | term ('st' | 'nd' | 'rd' | 'th') anyClass
  | 'middle' anyClass
  | plural ['from' term toOrThrough term]
  | anyClass term [toOrThrough term]
  | singularClass 'before' term
  | singularClass 'after' term
  | term ('of' | 'in' | 's') term
  | term ('whose' | 'where' | 'that') term
plural ::= pluralClass | 'every' anyClass
toOrThrough ::= 'to' | 'thru' | 'through'

call ::= message '(' expr* ')'
  | message ['in' | 'of'] [term] arguments
  | term name arguments
message ::= name | terminologyMessage
arguments ::= (preposition expression | flag | record)*
flag ::= ('with' | 'without') [name]+
record ::= 'given' (name ':' expr)*
preposition ::=
  'to' | 'from' | 'thru' | 'through'
  | 'by' | 'on' | 'into' | terminologyPreposition

```

**Figure 2.** AppleScript grammar for object references and message sends.

or masculine/feminine, and this information can be used by the custom parser for a dialect.

One of the main functions of AppleScript is to send and receive Apple Events (defined in Section 2.3). Sending an Apple Event is analogous to an object-oriented message send, while handling an Apple Event is done by defining a method whose name is an Apple Event. One view of this arrangement is that each application is an object that responds to a variety of messages containing complex argument types. AppleScript encourages a higher-level view in which each application manages a set of objects to which messages may be sent.

AppleScript also provides special syntax for manipulating Apple Event object specifiers, which are called “object references” in AppleScript documentation. When an operation is performed on an object reference, an Apple Event is created containing the appropriate verb and object specifier. Thus AppleScript needed to provide a concise syntax for expressing Apple Events. These AppleScript expressions create object specifiers:

```

the first word of paragraph 22
name of every figure of document “taxes”
the modification date of every file whose size > 1024

```

The first example is a reference to a particular word of a paragraph. The second refers to the collection of names associated with all figures in a document named “taxes”. The last one refers to the modification dates of large files.

For example, the object reference name **of** window 1 identifies the name of the first window in an application. Object references are like first-class pointers that can be dereferenced or updated. An object reference is automatically dereferenced when a primitive value is needed:

```
print the name of window 1
```

The primitive value associated with an object reference can be updated:

```
set the name of window 1 to “Taxes”
```

These examples illustrate that object specifiers can act as both l-values and r-values.

Figure 2 includes the part of the AppleScript grammar relating to object specifiers. The grammar makes use of four nonterminals that represent symbols from the application terminology: for **property**, **singularClass**, **pluralClass**, and **anyClass**. As mentioned in Section 2.3.1, a terminology has properties and elements, which are identified by a class name. Property names are identified by lexical analysis and passed to the parser. For class names the terminology can include both plural and singular forms or a generic “any” form. For example, name is a property, window is a singular class, and windows is a plural class. The grammar then accepts windows **from 1 to 10** and **every** window **from 1 to 10**,

Figure 2 also summarizes the syntax of messages. Arguments can be given by position or name after the **given** keyword. In addition, a set of standard prepositions can be used as argument labels. This allows messages of the form:

```
copy paragraph 1 to end of document
```

The first parameter is paragraph 1, and the second argument is a prepositional argument named **to** with value **end of document**.

One of the most difficult aspects of the language design arose from the fundamental ambiguity of object references: an object reference is itself a first-class value, but it also denotes a particular object (or set of objects) within an application. In a sense an object reference is like a symbolic pointer, which can be dereferenced to obtain its value; the referenced value can also be updated or assigned through the object reference. The difficulties arose because of a desire to hide the distinction between a reference and its value. The solution to this problem was to dereference them automatically when necessary, and require special syntax to create an object reference instead of accessing its value. The expression a reference **to** o creates a first-class reference to an object

described by `o`. Although the automatic dereferencing handles most cases, a script can explicitly dereference `r` using the expression `value of r`. The examples above can be expressed using a reference value:

```
set x to a reference to the name of window 1
```

The variable `x` can then be used in place of the original reference. The following statements illustrate the effect of operations involving `x`:

```
print the value of x
print x
set the value of x to "Taxes"
set x to "Taxes"
```

The first and second statements both print the name of the window. In the first statement the dereference is explicit, but in the second it happens implicitly because `print` expects a string, not a reference. The third statement changes the name of the window, while the last one changes the values of the variable `x` but leaves the window unchanged.

An object reference can be used as a base for further object specifications.

```
set x to a reference to window 1
print the name of x
```

Figure 3 and 4 describe the custom terminology dictionary for iChat, a chat, or instant messaging, program for the Macintosh. It illustrates the use of classes, elements, properties, and custom events. Terminologies for large applications are quite extensive: the Microsoft Word 2004 AppleScript Reference is 529 pages long, and the Adobe Photoshop CS2 AppleScript Scripting Reference is 251 pages. They each have dozens of classes and hundreds of properties.

In addition to the terminology interpreted by individual applications, AppleScript has its own terminology to identify applications on multiple machines. The expression `application "name"` identifies an application. The expression `application "appName" of machine "machineName"` refers to an application running on a specific machine. A block of commands can be targeted at an application using the `tell` statement:

```
tell application "Excel" on machine x
  put 3.14 into cell 1 of row 2 of window 1
end
```

This is an example of a static `tell` command, because the name of the target application is known statically, at compile time. The target of the `tell` statement can also be a dynamic value rather than an application literal. In this case the terminology is not loaded from the application. To communicate with a dynamic application using a statically specified terminology, a dynamic `tell` can be nested inside a static `tell`; the outer one sets the static terminology, while the inner one defines the dynamic target. This brings up the possibility that applications may receive messages that they

### Class application: iChat application

*Plural form:* applications  
*Elements:* account, service, window, document  
*Properties:*

- idle time integer  
*Time in seconds that I have been idle.*
- image picture  
*My image as it appears in all services.*
- status message string  
*My status message, visible to other people while I am online.*
- status string  
*My status on all services: away/offline/available.*

### Class service: An instant-messaging service

*Plural form:* services  
*Elements:* account  
*Properties:*

- status string  
*The status of the service: disconnecting/connected/connecting/disconnected.*
- id string  
*The unique id of the service.*
- name string  
*The name of the service.*
- image picture  
*The image for the service.*

### Class account: An account on a service

*Plural form:* accounts  
*Properties:*

- status string  
*away/offline/available/idle/unknown.*
- id string  
*The account's service and handle. For example: AIM:JohnDoe007.*
- handle string  
*The account's online name.*
- name string  
*The account's name as it appears in the buddy list.*
- status message  
*The account's status message.*
- capabilities list  
*The account's messaging capabilities.*
- image picture  
*The account's custom image.*
- idle time integer  
*The time in seconds the account has been idle.*

**Figure 3.** iChat Suite: Classes in the iChat scripting terminology [13].

## Events

### **log in** service

*Log in a service with an account. If the account password is not in the keychain the user will be prompted to enter one.*

### **log out** service

*Logs out of a service, or all services if none is specified.*

### **send message to** account

*Send account a text message or video invitation.*

---

**Figure 4.** iChat Suite: Events in the iChat scripting terminology [13].

do not understand. In such situations, the application should return an error.

Integration of multiple applications opens the possibility that a single command may involve object references from multiple applications. The target of a message is determined by examining the arguments of the message. If all the arguments are references to the same application, then that application is the target. But if the arguments contain references to different applications, one of the applications must be chosen as the target. Since applications can interpret only their own object specifiers, the other object references must be evaluated to primitive values, which can then be sent to the target application.

```
copy the name of the first window  
  of application "Excel"  
to the end of the first paragraph  
  of app "Scriptable Text Editor"
```

This example illustrates copying between applications without using the global clipboard. AppleScript picks the target for an event by examining the first object reference in the argument list. If the argument list contains references to other applications, the values of the references are retrieved and passed in place of the references in the argument list.

Standard events and reference structures are defined in the *Apple Event Registry*. The Registry is divided into suites that apply to domains of application. Suites contain specifications for classes and their properties, and events. Currently there are suites for core operations, text manipulation, databases, graphics, collaboration, and word services (spell-checking, etc.).

Jon Pugh, with a BSCS from Western Washington University in 1983, was in charge of the Apple Events registry. He also helped out with quality assurance and evangelism. Since then he has worked on numerous scriptable applications, and created "Jon's Commands," a shareware library of AppleScript extensions.

Terminologies also provide natural-language names for the four-letter codes used within an Apple Event. This metadata is stored in an application as a resource. As discussed in

Section 3.4 below, the terminology resources in an application are used when parsing scripts targeting that application.

## 3.3 Programming Language Features

AppleScript's programming language features include variables and assignment, control flow, and basic data structures. Control flow constructs include conditionals, a variety of looping constructs, subroutines, and exceptions. Subroutines allow positional, prepositional, and keyword parameters. Data structures include records, lists, and objects. Destructuring bind, also known as pattern matching, can be used to break apart basic data structures. Lists and records are mutable. AppleScript also supports objects and a simple transaction mechanism.

AppleScript has a simple object model. A *script object* contains *properties* and *methods*. Methods are dynamically dispatched, so script objects support a simple form of object-oriented programming. The following simple script declaration binds the name Counter to a new script object representing a counter:

```
script Counter  
  property count : 0  
  to increment  
    set count to count + 1  
    return count  
  end increment  
end script
```

A script declaration can be placed inside a method to create multiple instances. Such a method is called a factory method, and is similar to a constructor method in Java. Since script can access any lexically enclosing variables, all the objects created by a factory have access to the state of the object that constructed them. The resulting pattern of object references resembles the class/metaclass system in Smalltalk [23], although in much simpler form.

AppleScript's object model is a prototype model similar to that employed by Self [37], as opposed to the container-based inheritance model of HyperTalk. Script objects support single inheritance by delegating unhandled commands to the value in their parent property [36]. JavaScript later adopted a model similar to AppleScript.

The top level of every script is an implicit object declaration. Top-level properties are *persistent*, in that changes to properties are saved when the application running the script quits. A standalone script can be stored in a script file and executed by opening it in the Finder. Such a script can direct other applications to perform useful functions, but may also call other script files. Thus, script files provide a simple mechanism for modularity.

AppleScript provides no explicit support for threading or synchronization. However, the application hosting a script can invoke scripts from multiple threads: the execution engine was thread-safe when run by the non-preemptive scheduling in the original Macintosh OS. It is not safe when

English	the first character of every word whose style is bold
Japanese	スタイル=ボールドであるすべての単語の最初の文字
French	le premier caractère de tous les mots dont style est gras
Professional	{ words   style == bold }.character[1]

Figure 5. Illustration of dialects.

run on multiple preemptive threads on Mac OS X. Script objects can also be sent to another machine, making mobile code possible.

### 3.4 Parsing and Internationalization

The AppleScript parser integrates the terminology of applications with its built-in language constructs. For example, when targeting the Microsoft Excel™ application, spreadsheet terms are known by the parser—nouns like `cell` and `formula`, and verbs like `recalculate`. The statement `tell application "Excel"` introduces a block in which the Excel terminology is available. The terminology can contain names that are formed from multiple words; this means that the lexical analyzer must be able to recognize multiple words as a single logical identifier. As a result, lexical analysis depends upon the state of the parser: on entering a `tell` block, the lexical analysis tables are modified with new token definitions. The tables are reset when the parser reaches the end of the block. This approach increases flexibility but makes parsing more difficult. I believe the added complexity in lexing/parsing makes it more difficult for users to write scripts.

Apple also required that AppleScript, like most of its other products, support localization, so that scripts could be read and written in languages other than English. Scripts are stored in a language-independent internal representation. A *dialect* defines a presentation for the internal language. Dialects contain lexing and parsing tables, and printing routines. A script can be presented using any dialect—so a script written using the English dialect can be viewed in Japanese. Examples are given in Figure 5. For complete localization, the application terminologies must also include entries for multiple languages. Apple developed dialects for Japanese and French. A “professional” dialect, which resembles Java, was created but not released.

There are numerous difficulties in parsing a programming language that resembles a natural language. For example, Japanese does not have explicit separation between words. This is not a problem for language keywords and names from the terminology, but special conventions were required to recognize user-defined identifiers. Other languages have complex conjugation and agreement rules that are difficult to implement. Nonetheless, the internal representation of AppleScript and the terminology resources contain information to support these features.

The AppleScript parser was created using Yacc [29], a popular LALR parser generator. Poor error messages are a

common problem with LALR parsing [1]. I wrote a tool that produces somewhat better error messages by including a simplified version of the follow set at the point where the error occurred. The follow set was simplified by replacing some common sets of symbols (like binary operators) with a generic name, so that the error message would be “expected binary operator” instead of a list of every binary operator symbol. Despite these improvements, obscure error messages continue to be one of the biggest impediments to using AppleScript.

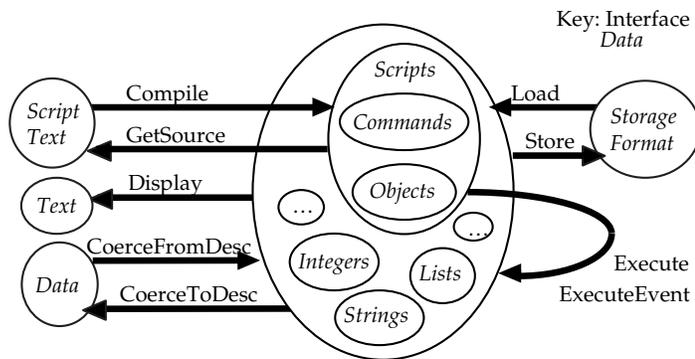
### 3.5 AppleScript Implementation

During the design of AppleScript in mid-1991, we considered building AppleScript on top of an existing language or runtime. We evaluated Macintosh Common Lisp (MCL), Franz Lisp, and Smalltalk systems from ParcPlace and Digitalk. These were all good systems, but were not suitable as a foundation for AppleScript for the same reason: there was not sufficient separation between the development environment and the runtime environment. Separating development from execution is useful because it allows compiled script to be executed in a limited runtime environment with low overhead. The full environment would be needed only when compiling or debugging a script.

Instead, we developed our own runtime and compiler. The runtime includes a garbage collector and byte-code interpreter. The compiler and runtime were loaded separately to minimize memory footprint.

One AppleScript T-shirt had the slogan “We don’t patch out the universe”. Many projects at Apple were implemented by “patching”: installing new functions in place of kernel operating system functions. The operating system had no protection mechanisms, so any function could be patched. Patches typically had to be installed in a particular order, or else they would not function. In addition, a bug in a patch could cause havoc for a wide range of applications.

AppleScript did not patch any operating system functions. Instead the system was carefully packaged as a thread-safe QuickTime component. QuickTime components are a lightweight dynamic library mechanism introduced by the QuickTime team. Only one copy of the AppleScript compiler and runtime was loaded and shared by all applications on a machine. The careful packaging is one of the reasons AppleScript was able to continue running unchanged through numerous operating system upgrades, and even onto the PowerPC.



**Figure 6.** Overview of the Open Scripting API.

The AppleScript runtime is implemented in C++. The entire system, including dialects, is 118K lines of code, including comments and header files. Compiling the entire AppleScript runtime took over an hour on the machines used by the development team. After the alpha milestone, the development team was not allowed to produce official builds for the testing team. Instead, the code had to be checked in and built by a separate group on a clean development environment. This process typically took 8 to 12 hours, because the process was not fully automated, so there was sometimes a significant lag between identifying a bug and delivering a fix to the quality assurance team. This was a significant source of frustration for the overall team.

## 4. Script Management

Open Scripting Architecture (OSA) allows any application to manipulate and execute scripts [11]. The Open Scripting API is centered around the notion of a *script*, as shown in Figure 6. A script is either a data value or a program. Many of the routines in the API are for translating between scripts and various external formats. Compile parses script text and creates a script object, while GetSource translates a script object back into human-readable script text. Display translates a value into a printed representation. When applied to a string, e.g. “Gustav”, GetSource returns a program literal “Gustav”, while Display just returns the text Gustav. CoerceFromDesc and CoerceToDesc convert AppleScript values to and from Apple Event descriptors. Load and Store convert to/from compact binary byte-streams that can be included in a file.

The Execute function runs a script in a context. A context is a script that contains bindings for global variables.

At its simplest, the script management API supports the construction of a basic script editor that can save scripts as stand-alone script applications.

The OSA API does not include support for debugging, although this was frequently discussed by the team. However, other companies have worked around this problem and created effective debugging tools (Section 6.3).

## 4.1 Embedding

The script management API also supports attaching scripts to objects in an existing application. Such scripts can be triggered during normal use of the application. This usage is supported by the ExecuteEvent function, which takes as input a script and an Apple Event. The event is interpreted as a method call to the script. The corresponding method declaration in the script is invoked. In this way an application can pass Apple Events to scripts that are attached to application objects.

Embedded scripts allow default application behavior to be customized, extended, or even replaced. For example, the Finder can run a script whenever a file is added to a folder, or an email package can run a script when new mail is received. Scripts can also be attached to new or existing menu items to add new functionality to an application. By embedding a universal scripting languages, application developers do not need to build proprietary scripting languages, and users do not need to learn multiple languages. Users can also access multiple applications from a single script. AppleScript demonstrated the idea that a single scripting language could be used for all applications, while allowing application-specific behaviors to be incorporated so that the language was specialized for use in each application.

Embedding can also be used to create entire applications. In this case there is no predefined application structure to which scripts are attached. Instead, the user builds the application objects — for data and user interfaces, and then attaches scripts to them. Several application development tools based on AppleScript are described in Section 6.

## 4.2 Multiple Scripting Languages

Halfway through the development of AppleScript, Apple management decided to allow third-party scripting languages to be used in addition to AppleScript. A new API for managing scripts and scripting language runtime engines had to be designed and implemented. These changes contributed to delays in shipping AppleScript. However, they also led to a more robust architecture for embedding.

In February of 1992, just before the first AppleScript alpha release, Dave Winer convinced Apple management that having one scripting language would not be good for the Macintosh. At that time, Dave Winer was an experienced Macintosh developer, having created one of the first outliner applications, ThinkTank. In the early 1990s, Dave created an alternative scripting system, called Frontier. Before I joined the project, Apple had discussed the possibility of buying Frontier and using it instead of creating its own language. For some reason the deal fell through, but Dave continued developing Frontier. Apple does not like to take business away from developers, so when Dave complained that the impending release of AppleScript was interfering with his product, Apple decided the AppleScript should be opened up to multiple scripting languages. The AppleScript team mod-

ified the OSA APIs so that they could be implemented by multiple scripting systems, not just AppleScript. As a result, OSA is a generic interface between clients of scripting services and scripting systems that support a scripting language. Each script is tagged with the scripting system that created it, so clients can handle multiple kinds of script without knowing which scripting system they belong to.

Dave Winer's Frontier is a complete scripting and application development environment that eventually became available as an Open Scripting component. Dave went on to participate in the design of web services and SOAP [4]. Tcl, JavaScript, Python and Perl have also been packaged as Open Scripting components.

### 4.3 Recording Events as Scripts

The AppleScript infrastructure supports recording of events in *recordable* applications, which publish events in response to user actions. Donn Denman, a senior software engineer on the AppleScript team with a BS in Computer Science and Math from Antioch College, designed and implemented much of the infrastructure for recording. At Apple he worked on Basic interpreters. He was involved in some of the early discussions of AppleScript, worked on Apple Events and application terminologies in AppleScript. In addition, Donn created MacroMaker, a low-level event record and playback system for Macintosh OS System 5. Working on MacroMaker gave Donn a lot of experience in how recording should work.

Recording allows automatic generation of scripts for repeated playback of what would otherwise be repetitive tasks. Recorded scripts can be subsequently generalized by users for more flexibility. This approach to scripting alleviates the "staring at a blank page" syndrome that can be so crippling to new scripters. Recording is also useful for learning the terminology of basic operations of an application, because it helps users to connect actions in the graphical interface with their symbolic expression in script.

Recording high-level events is different from recording low-level events of the graphical user interface. Low-level events include mouse and keyboard events. Low-level events can also express user interface actions, e.g. "perform Open menu item in the File menu", although the response to this event is usually to display a dialog box, not to open a particular file. Additional low-level events are required to manipulate dialog boxes by clicking on interface elements and buttons. Low-level events do not necessarily have the same effect if played back on a different machine, or when different windows are open. High-level events are more robust because they express the intent of an action more directly. For example, a high-level event can indicate which file to open.

Recording is supported by a special mode in the Apple Event manager, based on the idea that a user's actions in manipulating a GUI interface can be described by a corresponding Apple Event. For example, if the user selects the

File Open menu, then finds and selects a file named "Résumé" in a folder named "Personal", the corresponding Apple Event would be a FileOpen event containing the path "Personal:Résumé". To be recordable, an application must post Apple Events that describe the actions a user performs with the GUI.

Recordable applications can be difficult to build, since they must post an Apple Event describing each operation performed by a user. The AppleScript team promoted an architecture that turned this difficulty into a feature. We advocated that applications should be factored into two parts, a GUI and a back end, where the only communication from the GUI to the back end is via Apple Events. With this architecture, all the core functionality of the application must be exposed via Apple Events, so the application is inherently scriptable. The GUI's job becomes one of translating low-level user input events (keystrokes and mouse movements) into high-level Apple Events. An application built in this way is inherently recordable; the Apple Event manager simply records the Apple Events that pass from the GUI to the back end. If an application is already scriptable, it can be made recordable by arranging for the user interface to communicate with the application model only through Apple Events.

The reality of recording is more complex, however. If there is a type Apple Event to add characters into a document, the GUI must forward each character immediately to the back end so that the user will see the result of typing. During recording, if the user types "Hello" the actions will record an undesirable script:

```
type "H"
type "e"
type "l"
type "l"
type "o"
```

It would be better to record type "Hello". To get this effect, the GUI developer could buffer the typing and send a single event. But then the user will not see the effect of typing immediately. AppleEvents has the ability to specify certain events as *record-only*, meaning that it is a summary of a user's actions and should not be executed. Creating such summaries makes developing a recordable application quite difficult.

In 2006 twenty-five recordable applications were listed on Apple's website and in the AppleScript Sourcebook [8], one of several repositories of information about AppleScript. Some, but fewer than half, of the major productivity applications are recordable. Recordable applications include Microsoft Word and Excel, Netscape Navigator, Quark Express (via a plugin) and CorelDRAW.

One of the inherent difficulties of recording is the ambiguity of object specification. As the language of events becomes more rich, there may be many ways to describe a given user action. Each version might be appropriate for a

given situation, but the system cannot pick the correct action without knowing the intent of the user. For example, when closing a window, is the user closing the front window or the window specifically named “Example”? This is a well-known problem in research on programming by example, where multiple examples of a given action can be used to disambiguate the user’s intent. Allen Cypher did fundamental research on this problem while at Apple. He built a prototype system called Eager that anticipated user actions by watching previous actions [21, 22]. AppleScript does not have built-in support for analyzing multiple examples. There are also ambiguities when recording and embedding are combined: if a recorded event causes a script to execute, should the original event or the events generated by the script be recorded? Application designers must decide what is most appropriate for a given situation. Cypher worked with Dave Curbow in writing guidelines to help developers make these difficult choices [26].

Recording can also be used for other purposes. For example, a help system can include step-by-step instructions defined by a script. The steps can be played as normal scripts, or the user can be given the option of performing the steps manually under the supervision of the help system. By recording the user’s actions, the help system can provide feedback or warnings when the user’s actions do not correspond to the script.

## 5. Development Process

Unlike most programming languages, AppleScript was designed within a commercial software development project. The team members are listed in Figure 7. AppleScript was designed by neither an individual nor a committee; the team used a collaborative design process, with significant user testing and evaluation. The project leaders guided the process and made final decisions: there was lively debate within the group about how things should work. The extended team included project management, user interface design and testing, documentation, and product marketing.

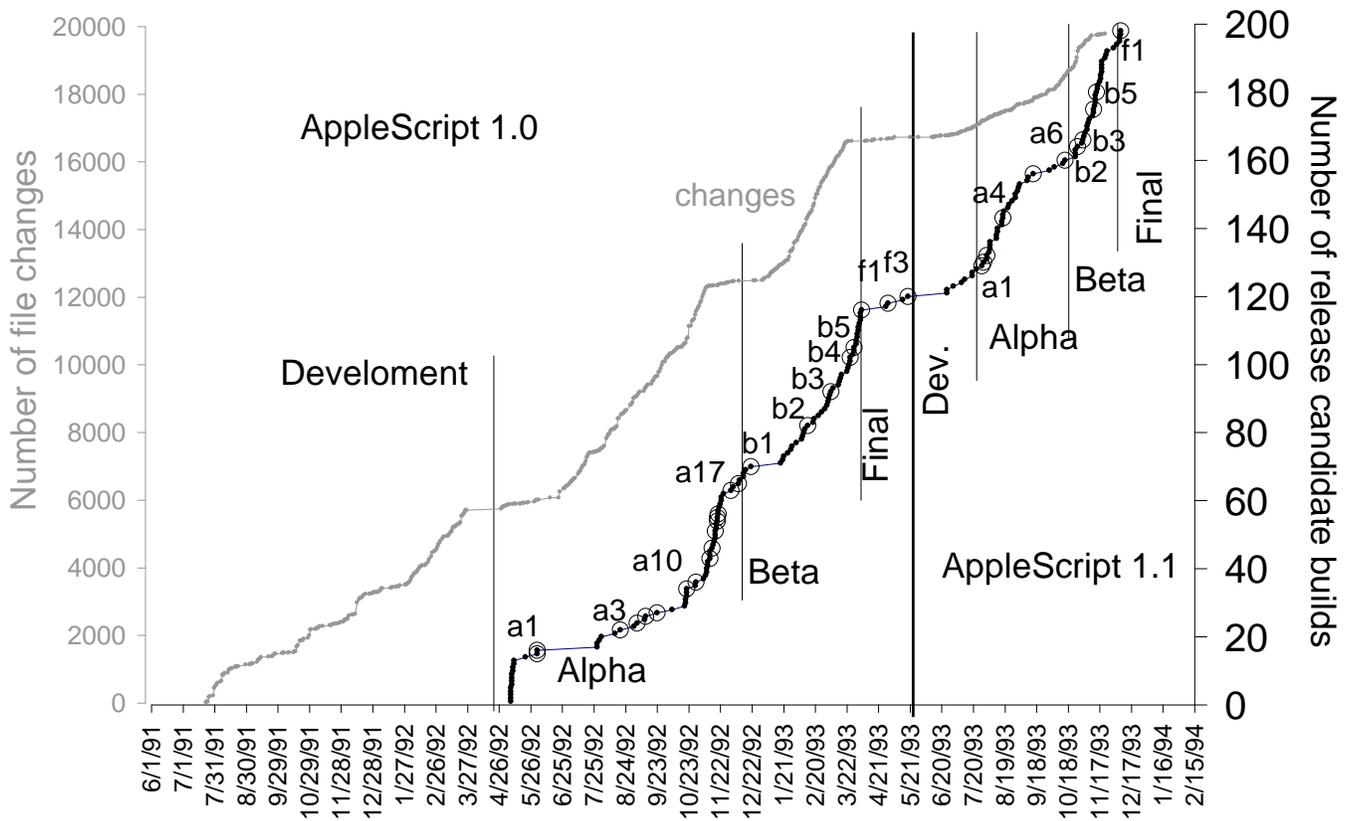
The AppleScript project had a strong quality assurance (QA) team. They created a large test suite which was run against nightly builds. From a management viewpoint, the QA group also had significant control over the project, because they were required to give final approval of a release.

The project was code-named “Gustav” after Donn’s massive Rottweiler dog. The dog slimed everything it came in contact with, and was the impetus behind a T-shirt that read “Script Happens”. The project tag line was “Pure Guava” because Gary Bond was designing a t-shirt that said “AppleScript: Pure Gold” and Warren Harris got him to change it to Pure Guava after the Ween album he was in love with at the time.

AppleScript and the associated tools were designed and implemented between 1990 and 1993. Figure 8 gives a timeline of the development process. The line labeled “changes”

<b>Jens Alfke</b> Developer	<b>Ron Karr</b> QA, Apple Events Developer
<b>Greg Anderson</b> Developer, Scriptable Finder	<b>Edmund Lai</b> Developer, Apple Events
<b>Mike Askins</b> Engineering Project Manager	<b>Ron Lichty</b> Manager, Finder
<b>Gary Bond</b> QA	<b>Bennet Marks</b> Developer
<b>Scott Bongiorno</b> QA, User Testing	<b>Mark Minshull</b> Manager
<b>B. Bruce Brinson</b> Developer	<b>Kazuhisa Ohta</b> Developer, Dialects
<b>Kevin Calhoun</b> Manager	<b>Donald Olson</b> QA, Manager
<b>Jennifer Chaffee</b> User Interface Design	<b>Chuck Piercey</b> Marketing
<b>Dan Clifford</b> Developer	<b>Kurt Piersol</b> Architect
<b>William Cook</b> Architect, Developer, Manager	<b>James Redfern</b> QA, Developer
<b>Sean Cotter</b> Documentation	<b>Brett Sher</b> Developer, QA
<b>Dave Curbow</b> User Interface Design	<b>Laile Di Silvestro</b> QA, Developer
<b>Donn Denman</b> Developer	<b>Sal Soghoian</b> Product Manager
<b>Sue Dumont</b> Developer, QA	<b>Francis Stanbach</b> Developer, Scriptable Finder
<b>Mike Farr</b> Marketing	<b>Kazuhiko Tateda</b> Japanese Dialect
<b>Mitch Gass</b> Documentation	<b>Larry Tesler</b> Manager, VP
<b>Laura Clark Hamersley</b> Marketing	<b>Mark Thomas</b> Evangelist
<b>Warren Harris</b> Architect, Developer	<b>Susan Watkins</b> Marketing
<b>Eric House</b> QA, Developer	

**Figure 7.** AppleScript and related project team members.



**Figure 8.** Development statistics: number of file changes and candidate builds.

shows the cumulative number of files changed during development (the scale is on the left). The second line shows the cumulative number of candidate release builds. The final candidate builds were created by Apple source control group from a clean set of sources and then given to the testing team. By placing source code control between development and testing, Apple ensured that each build could be recreated on a clean development environment with archived sources. Note that the number of file changes in the alpha or beta phases starts off slowly, then increases until just before the next milestone, when changes are no longer allowed unless absolutely necessary.

The AppleScript Beta was delivered in September 1992. In April 1993 the AppleScript 1.0 Developer's Toolkit shipped, including interface declaration files (header files), sample code, sample applications and the Scripting Language Guide.

The first end-user version, AppleScript 1.1, was released in September 1993 and included with System 7 Pro. In December 1993, the 1.1 Developer's Toolkit and Scripting Kit versions both released. In 1994, AppleScript was included as part of System 7.5.

In January 1993, Apple management decided that the next version of AppleScript had to have more features than AppleScript 1.1, but that the development must be done with half the number of people. Since this was not likely

to lead to a successful development process, Warren and I decided to leave Apple. Without leadership, the AppleScript group was disbanded. Many of the team members, including Jens Alfke, Donn Denman, and Donald Olson, joined Kurt Piersol on the OpenDoc team, which was working on visual integration of applications. AppleScript was integrated into the OpenDoc framework.

### 5.1 Documentation

Internal documentation was ad hoc. The team made extensive use of an early collaborative document management/writing tool called Instant Update, that was used in a wiki-like fashion, a living document constantly updated with the current design. Instant Update provides a shared space of multiple documents that were viewed and edited simultaneously by any number of users. Each user's text was color-coded and time-stamped. I have not been able to recover a copy of this collection of documents.

No formal semantics was created for the language, despite the fact that my PhD research and work at HP Labs was focused entirely on formal semantics of programming languages. One reason was that only one person on the team was familiar with formal semantics techniques, so writing a formal semantics would not be an effective means of communication. In addition, there wasn't much point in developing a formal semantics for the well-known features (objects,

inheritance, lexical closures, etc.), because the goal for this aspect of the language was to apply well-known constructs, not define new ones. There was no standard formal semantic framework for the novel aspects of AppleScript, especially the notion of references for access to external objects residing in an application. The project did not have the luxury of undertaking extensive research into semantic foundations; its charter was to develop and ship a practical language in a short amount of time. Sketches of a formal semantics were developed, but the primary guidance for language design came from solving practical problems and from user studies, rather than from a priori formal analysis.

The public documentation was developed by professional writers who worked closely with the team throughout the project. The primary document is *Inside Macintosh: Inter-application Communication*, which includes details on the Apple Event Manager and Scripting Components [11]. The AppleScript language is also thoroughly documented [2], and numerous third-party books have been written about it, for examples see [31, 24]. Mitch Gass and Sean Cotter documented Apple Events and AppleScript for external use. Mitch has a bachelor's degrees in comparative literature and computer science, and worked at Tandem and Amiga before joining Apple. Mitch worked during the entire project to provide that documentation, and in the process managed to be a significant communication point for the entire team.

## 5.2 User Testing

Following Apple's standard practice, we user-tested the language in a variety of ways. We identified novice users and asked them, "What do you think this script does?" The following questions illustrate the kinds of questions asked during user testing.

**Part I.** Please answer the following multiple choice questions about AppleScript.

3. Given the handler:

```
on doit from x to y with z
    return (x * y) + z
end doit
```

What does the following statement evaluate to?

```
doit with 3 from 8 to 5
```

- a) 29
- b) 43
- c) error
- d) other:

**Part II.** Please state which of the following AppleScript statements you prefer.

- 8. a) `put "a", {"b", "c"} into x`  
b) `put {"a", {"b", "c"}} into x`
- 9. a) window named "fred"  
b) window "fred"
- 10. a) window 1  
b) window #1
- 11. a) word 34  
b) word #34
- 12. a) "apple" < "betty"  
b) "apple" comes **before** "betty"

**Part III.** This section shows sequences of commands and then asks questions about various variables after they are executed.

15. Given the commands:

```
put {1, 2, 3} into x
put x into y
put 4 into item 1 of x
```

**What is x?**

- a) {1, 2, 3}
- b) {4, 2, 3}
- c) error
- d) other:

**What is y?**

- a) {1, 2, 3}
- b) {4, 2, 3}
- c) error
- d) other:

**Part IV.** In this section, assume that all AppleScript statements refer to window 1, which contains the following text:

```
this is a test
of the emergency broadcast system
```

**18. What does the following statement evaluate to?**

```
count every line of window 1
```

- a) 2
- b) 4, 5
- c) 9
- d) 14, 33
- e) 47
- f) error
- g) other:

### What does the following statement evaluate to?

count each line **of** window 1

- a) 2
- b) 4, 5
- c) 9
- d) 14, 33
- e) 47
- f) error
- g) other:

### 21. What does the following statement evaluate to?

**every** line **of** window 1  
**whose first** character = "x"

- a) {}
- b) error
- c) other:

One result of user testing concerned the choice of verb for assignment commands. The average user thought that after the command **put**  $x$  **into**  $y$  the variable  $x$  no longer retained its old value. The language was changed to use **copy**  $x$  **into**  $y$  instead. We also conducted interviews and a round-table discussion about what kind of functionality users would like to see in the system. In the summer of 1992, Apple briefed 50 key developers and collected reactions. The user interface team conducted controlled experiments of the usability of the language in early 1993, but since these took place during the beta-testing period, they were too late in the product development cycle to have fundamental impact.

## 6. AppleScript Applications and Tools

Much of the practical power of AppleScript comes from the applications and tools that work with scripts and handle events. From the viewpoint of AppleScript, applications are large, well-designed and internally consistent libraries of specialized functionality and algorithms. So, when used with a database application, AppleScript can perform data-oriented operations. When used with a text layout application, AppleScript can automate typesetting processes. When used with a photo editing application, AppleScript can perform complex image manipulation.

Since new libraries can be created to cover any application domain, only the most basic data types were supported in AppleScript directly. For example, string handling was minimal in AppleScript. AppleScript's capabilities were initially limited by the availability of scriptable applications. Success of the project required that many applications and diverse parts of the operating system be updated to support scripting.

A second benefit of pervasive scripting is that it can be used to provide a uniform interface to the operating system. With Unix, access to information in a machine is idiosyncratic, in the sense that one program was used to list print jobs, another to list users, another for files, and another for hardware configuration. I envisioned a way in which all these different kinds of information could be referenced uniformly.

A *uniform naming model* allows every piece of information anywhere in the system, be it an application or the operating system, to be accessed and updated uniformly. Application-specific terminologies allow applications to be accessed uniformly; an operating system terminology would provide access to printer queues, processor attributes, or network configurations. Thus, the language must support multiple terminologies simultaneously so that a single script can access objects from multiple applications and the operating system at the same time.

### 6.1 Scriptable Finder

Having a scriptable Finder was a critical requirement for AppleScript, since the Finder provides access to most system resources. However, it was difficult to coordinate schedules and priorities because the Finder and AppleScript teams were in different divisions within Apple. The Finder team was also pulled in many directions at once.

As a result, Finder was not fully scriptable when AppleScript shipped in 1992. The Finder team created a separate library, called the "Finder scripting extension", to provide some additional Finder script commands. The Finder had been rewritten in C++ from the ground up for System 7 to be extensible. But extensions relied on internal C++ dispatch tables, so the Finder was not dynamically extensible: it had to be recompiled for each extension. The Finder extension mechanism had been designed so that Finder functionality could grow incrementally. It was the mechanism for adding large quantities of new functionality to support a specific project.

It was not until 1997 that a scriptable Finder was released. A year later the Finder supported embedding, which greatly increased its power. Embedding allowed scripts to be triggered from within the Finder in response to events, for example opening a folder or emptying the trash.

### 6.2 Publishing Workflow

Automation of publishing workflows is a good illustration of AppleScript and scriptable applications. Consider the automation of a catalog publishing system. An office-products company keeps all its product information in a FileMaker Pro™ database that includes descriptions, prices, special offer information, and a product code. The product code identifies a picture of the product in a Kudos™ image database. The final catalog is a QuarkXPress™ document that is ready for printing. Previously, the catalog was produced manually,

a task that took a team of twenty up to a month for a single catalog.

An AppleScript script automates the entire process. The script reads the price and descriptive text from the FileMaker Pro database and inserts it into appropriate QuarkXPress fields. The script applies special formatting: it deletes the decimal point in the prices and superscripts the cents (e.g. 34<sup>99</sup>). To make the text fit precisely in the width of the enclosing box, the script computes a fractional expansion factor for the text by dividing the width of the box by the width of the text (this task was previously done with a calculator). It adjusts colors and sets the first line of the description in boldface type. Finally, it adds special markers like “Buy 2 get 1 free” and “Sale price \$17<sup>99</sup>” where specified by the database.

Once this process is automated, one person can produce the entire catalog in under a day, a tiny fraction of the time taken by the manual process. It also reduced errors during copying and formatting. Of course, creating and maintaining the scripts takes time, but the overall time is significantly reduced over the long run.

### 6.3 Scripting Tools

AppleScript included a simple and elegant script editor created by Jens Alfke, who had graduated from Caltech and worked with Kurt Piersol at Xerox Parc on Smalltalk-80 applications. Jens was one of the key developers on the AppleScript team; he focused on tools, consistency of the APIs and usability of the overall system.

Soon after AppleScript was released, more powerful script development environments were created outside Apple. They addressed one of the major weaknesses of AppleScript: lack of support for debugging. One developer outside Apple who took on this challenge is Cal Simone, who has also been an unofficial evangelist for AppleScript since its inception. Cal created *Scripter*, which allows users to single-step through a script. It works by breaking a script up into individual lines that are compiled and executed separately. The enclosing **tell** statements are preserved around each line as it is executed. Scripter also allows inspection of local variables and execution of immediate commands within the context of the suspended script. *Script Debugger* uses a different technique: it adds a special Apple Event between each line of a script. The Apple Event is caught by the debugger and the processing of the script is suspended. The current values of variables can then be inspected. To continue the script, the debugger simply returns from the event.

AppleScript also enables creation of sophisticated interface builders. The interface elements post messages when a user interacts with them. The user arranges the elements into windows, menus, and dialogs. Scripts may be attached to any object in the interface to intercept the messages being sent by the interface elements and provide sophisticated behavior and linking between the elements. Early application builders included Frontmost<sup>TM</sup>, a window and dialog

builder, and AgentBuilder<sup>TM</sup>, which specialized in communication front-ends. Version 2.2 of HyperCard, released in 1992, included support for OSA, so that AppleScript or any OSA language could be used in place of HyperTalk.

Two major application builders have emerged recently. FaceSpan, originally released in 1994, has grown into a full-featured application development tool. FaceSpan includes an integrated script debugger. Apple released AppleScript Studio in 2002 as part of its XCode development platform. A complete application can be developed with a wide range of standard user interface elements to which scripts can be attached. AppleScript Studio won Macworld Best of Show Awards at the 2001 Seybold Conference in San Francisco.

In 2005 Apple released Automator, a tool for creating sequences of actions that define workflows. Automator sequences are not stored or executed as AppleScripts, but can contain AppleScripts as primitive actions. The most interesting thing about Automator is that each action has an input and an output, much like a command in a Unix pipe. The resulting model is quite intuitive and easy to use for simple automation tasks.

Although Apple Events are normally handled by applications, it is also possible to install *system event handlers*. When an Apple Event is delivered to an application, the application may handle the event or indicate that it was not handled. When an application does not handle an event, the Apple Event manager searches for a system event handler. System event handlers are packaged in *script extensions* (also known as OSAX) and are installed on the system via Scripting Additions that are loaded when the system starts up.

### 6.4 Scriptable Applications

Eventually, a wide range of scriptable applications became available: there are currently 160 scriptable applications listed on the Apple web site and the AppleScript sourcebook [8]. Every kind of application is present, including word processors, databases, file compression utilities, and development tools. Many of the most popular applications are scriptable, including Microsoft Office, Adobe Photoshop, Quark Expression, FileMaker, and Illustrator. In addition, most components of the Mac OS are scriptable, including the Finder, QuickTime Player, Address Book, iTunes, Mail, Safari Browser, AppleWorks, DVD Player, Help Viewer, iCal, iChat, iSync, iPhoto, and control panels.

Other systems also benefitted from the infrastructure created by AppleScript. The Macintosh AV<sup>TM</sup> speech recognition system uses AppleScript, so any scriptable application can be driven using speech.

## 7. Evolution

After version 1.1, the evolution of AppleScript was driven primarily by changes in the Macintosh OS. Since AppleScript was first released, the operating system has undergone two major shifts, first when Apple moved from the Motorola

68000 to the PowerPC chip, and then when it moved from the Classic Macintosh OS to the Unix-based OS X. Few changes were made to the language itself, while scriptable applications and operating system components experienced rapid expansion and evolution. A detailed history with discussion of new features, bugs, and fixes can be found in the AppleScript Sourcebook [8], which we summarize here.

The first upgrade to AppleScript, version 1.1.2, was created for Macintosh OS 8.0, introduced in July 1997. Despite the rigorous source code configuration process (see Section 5), Apple could not figure out how to compile the system and contracted with Warren Harris to help with the job. A number of bugs were fixed and some small enhancements were made to conform to Macintosh OS 8.0 standards. At the same time several system applications and extensions were changed in ways that could break old scripts. The most important improvement was a new scriptable Finder, which eliminated the need for a Finder scripting extension.

In 1997 AppleScript was at the top of the list of features to eliminate in order to save money. Cal Simone, mentioned in Section 6.3, successfully rallied customers to rescue AppleScript.

In October 1998 Apple released AppleScript 1.3 with UNICODE support recompiled as a native PowerPC extension; however, the Apple Events Manager was still emulated as Motorola 68000 code. The dialect feature was no longer supported; English became the single standard dialect. This version came much closer to realizing the vision of uniform access to all system resources from scripts. At least 30 different system components, including File Sharing, Apple Video Player and Users & Groups, were now scriptable. New scriptable applications appeared as well, including Microsoft Internet Explorer and Outlook Express.

The PowerPC version of AppleScript received an Eddy Award from MacWorld as “Technology of the Year” for 1998 and was also demonstrated in Steve Jobs’ Seybold 1998 address. In 2006, MacWorld placed AppleScript as #17 on its list of the 30 most significant Mac products ever. AppleScript was a long-term investment in fundamental infrastructure that took many years to pay dividends.

The most significant language changes involved the **tell** statement. For example, the machine class used to identify remote applications was extended to accept URLs (see Section 3.2), allowing AppleScript control of remote applications via TCP/IP.

When Mac OS X was released in March 2001, it included AppleScript 1.6. In porting applications and system components to OS X, Apple sometimes sacrificed scripting support. As a result, there was a significant reduction in the number of scriptable applications after the release of OS X. Full scriptability is being restored slowly in later releases.

In October 2006, Google reported an estimated 8,570,000 hits for the word “AppleScript”.

## 8. Evaluation

AppleScript was developed by a small group with a short schedule, a tight budget and a big job. There was neither time nor money to fully research design choices.

AppleScript and Apple Events introduced a new approach to remote communication in high-latency environments [33]. Object references are symbolic paths, or queries, that identify one or more objects in an application. When a command is applied to an object reference, both the command and the object reference are sent (as an Apple Event containing an object specifier) to the application hosting the target object. The application interprets the object specifier and then performs the action on the specified objects.

In summary, AppleScript views an application as a form of object-oriented database. The application publishes a specialized terminology containing verbs and nouns that describe the logical structure and behavior of its objects. Names in the terminology are composed using a standard query language to create programs that are executed by the remote application. The execution model does not involve remote object references and proxies as in CORBA. Rather than send each field access and method individually to the remote application and creating proxies to represent intermediate values, AppleScript sends the entire command to the remote application for execution. From a pure object-oriented viewpoint, the entire application is the only real object; the “objects” within it are identified only by symbolic references, or queries.

After completing AppleScript, I learned about COM and was impressed with its approach to distributed object-oriented programming. Its consistent use of interfaces enables interoperability between different systems and languages. Although interface negotiation is complex, invoking a method through an interface is highly optimized. This approach allows fine-grained objects that are tightly coupled through shared binary interfaces. For many years I believed that COM and CORBA would beat the AppleScript communication model in the long run. However, recent developments have made me realize that this may not be the case.

AppleScript uses a large-granularity messaging model that has many similarities to the web service standards that began to emerge in 1999 [10]. Both are loosely coupled and support large-granularity communication. Apple Events data descriptors are similar to XML in that they describe arbitrary labeled tree structures without fixed semantics. AppleScript terminologies are similar to web service description language (WSDL) files. It is perhaps not an accident that Dave Winer, who worked extensively with AppleScript and Apple Events, is also one of the original developers of web service models. There may be useful lessons to be learned for web services, given that AppleScript represents a significant body of experience with large-granularity messaging. One difference is that AppleScript includes a standard query model for identifying remote objects. A similar ap-

proach could be useful for web services. As I write in 2006, I suspect that COM and CORBA will be overwhelmed by web services, although the outcome is far from certain now.

AppleScript is also similar to traditional database interfaces like ODBC [38]. In AppleScript the query model is integrated directly into the language, rather than being executed as strings as in ODBC. A similar approach has been adopted by Microsoft for describing queries in .NET languages [3].

User tests revealed that casual users don't easily understand the idea of references, or having multiple references to the same value. It is easier to understand a model in which values are copied or moved, rather than assigning references. The feedback from user tests in early 1993 was too late in the development cycle to address this issue with anything more than a cosmetic change, to use **copy** instead of **set** for assignment.

Writing scriptable applications is difficult. Just as user interface design requires judgment and training, creating a good scripting interface requires a lot of knowledge and careful design. It is too difficult for application developers to create terminologies that work well in the naturalistic grammar. They must pay careful attention to the linguistic properties of the names they choose.

The experiment in designing a language that resembled natural languages (English and Japanese) was not successful. It was assumed that scripts should be presented in "natural language" so that average people could read and write them. This led to the invention of multi-token keywords and the ability to disambiguate tokens without spaces for Japanese Kanji. In the end the syntactic variations and flexibility did more to confuse programmers than to help them out. It is not clear whether it is easier for novice users to work with a scripting language that resembles natural language, with all its special cases and idiosyncrasies. The main problem is that AppleScript only appears to be a natural language: in fact, it is an artificial language, like any other programming language. Recording was successful, but even small changes to the script may introduce subtle syntactic errors that baffle users. It is easy to read AppleScript, but quite hard to write it.

When writing programs or scripts, users prefer a more conventional programming language structure. Later versions of AppleScript dropped support for dialects. In hindsight, we believe that AppleScript should have adopted the Professional Dialect that was developed but never shipped.

Finally, readability was no substitute for an effective security mechanism. Most people just run scripts—they don't read or write them.

## 9. Conclusion

AppleScript is widely used today and is a core technology of Mac OS X. Many applications, including Quark Express, Microsoft Office, and FileMaker, support scripting.

Small scripts are used to automate repetitive tasks. Larger scripts have been developed for database publishing, document preparation, and even web applications.

There are many interesting lessons to be learned from AppleScript. On a technical level, its model of pluggable embedded scripting languages has become commonplace. The communication mechanism of Apple Events, which is certainly inferior to RPC mechanisms for single-machine or in-process interactions, may turn out to be a good model for large-granularity communication models such as web services. Many of the current problems in AppleScript can be traced to the use of syntax based on natural language; however, the ability to create pluggable dialects may provide a solution in the future, by creating a new syntax based on conventional programming languages.

## Acknowledgments

Thanks to Jens Alfke, Paul Berkowitz, Bill Cheeseman, Chris Espinosa, Michael Farr, Steve Goldband Tom Hammer, David Hoerl, Alexander Kellett, Wayne Malkin, Matt Neuburg, Chuck Piercey, Hamish Sanderson, and Stephen Weyl, for discussions about this paper. Special thanks to Andrew Black and Kathleen Fisher for their guidance, encouragement, flexibility, and careful reading of my work in progress.

## References

- [1] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.
- [2] Apple Computer Inc. *AppleScript Language Guide*. Addison-Wesley, 1993.
- [3] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in  $\omega$ . In *European Conference on Object-Oriented Programming*. Springer Verlag, 2005.
- [4] Don Box, David EhneBuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielson, Satish Thatte, and Dave Winer. Simple object access protocol 1.1. <http://www.w3.org/TR/SOAP>.
- [5] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 303–311, 1990.
- [6] Peter Canning, William Cook, Walt Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.
- [7] Peter Canning, William Cook, Walt Hill, and Walter Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 457–467, 1989.
- [8] Bill Cheeseman. Applescript sourcebook. <http://www.AppleScriptSourcebook.com>.

- [9] Peter P. Chen. The entity-relationship model — toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.
- [10] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, and Sanjiva Weerawarana. Web Services Description Language Version 1.2, July 2002. W3C Working Draft 9.
- [11] Apple Computer. *Inside Macintosh: Interraplication Communication*. Addison-Wesley, 1991.
- [12] Apple Computer. *Inside Macintosh: Macintosh Toolbox Essentials*. Addison-Wesley, 1991.
- [13] Apple Computer. ichtat 2.0 dictionary. FooDoo Lounge web site by Richard Morton, 2002-2005.
- [14] William Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
- [15] William Cook. A proposal for making Eiffel type-safe. In *Proc. European Conf. on Object-Oriented Programming*, pages 57–70. British Computing Society Workshop Series, 1989. Also in *The Computer Journal*, 32(4):305–311, 1989.
- [16] William Cook. Object-oriented programming versus abstract data types. In *Proc. of the REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
- [17] William Cook. Interfaces and specifications for the Smalltalk collection classes. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 1992.
- [18] William Cook, Walt Hill, and Peter Canning. Inheritance is not subtyping. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 125–135, 1990.
- [19] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 433–444, 1989.
- [20] William R. Cook and Victor Law. An algorithm editor for structured design (abstract). In *Proc. of the ACM Computer Science Conference*, 1983.
- [21] Allen Cypher. Eager: programming repetitive tasks by example. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 33–39, New York, NY, USA, 1991. ACM Press.
- [22] Allen Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993. Full text available at [web.media.mit.edu/~lieber/PBE/](http://web.media.mit.edu/~lieber/PBE/).
- [23] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.
- [24] A. Goldstein. *AppleScript: The Missing Manual*. O'Reilly, 2005.
- [25] Warren Harris. Abel posthumous report. HP Labs, 1993.
- [26] Apple Computer Inc. Scripting interface guidelines. Technical Report TN2106, Apple Computer Inc.
- [27] Apple Computer Inc. *HyperCard User's Guide*. Addison Wesley, 1987.
- [28] Apple Computer Inc. *HyperCard Script Language Guide: The HyperTalk Language*. Addison Wesley, 1988.
- [29] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.
- [30] S. Michel. *HyperCard: The Complete Reference*. Osborne Mc-GrawHill, 1989.
- [31] M. Neuburg. *AppleScript : The Definitive Guide*. O'Reilly, 2003.
- [32] Object Management Group. *OMG Unified Modeling Language Specification Version 1.5*, March 2003.
- [33] David A. Patterson. Latency lags bandwidth. *Commun. ACM*, 47(10):71–75, 2004.
- [34] Trygve Reenskaug. Models — views — controllers. Technical report, Xerox PARC, December 1979.
- [35] Alan Snyder. The essence of objects: Concepts and terms. *IEEE Softw.*, 10(1):31–42, 1993.
- [36] Lynn Andrea Stein, Henry Lieberman, and David Ungar. A shared view of sharing: The treaty of orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press and Addison-Wesley, 1989.
- [37] D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 227–242, 1987.
- [38] Murali Venkatrao and Michael Pizzo. SQL/CLI – a new binding style for SQL. *SIGMOD Record*, 24(4):72–77, 1995.
- [39] Steven R. Wood. Z — the 95% program editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 1–7, New York, NY, USA, 1981. ACM Press.