I’ve been thinking lately about the purpose of this column, which debuted in the previous issue of *develop*. Permit me to take a moment to say something about that before I get down to some tips about dictionaries.

During the first couple of years after the birth of the Macintosh, there was a period of chaos, when application developers were figuring out how to extend the basic user interface. For example, some of the most commonly used menu commands appeared in different locations in various applications, and, more important, keyboard shortcuts varied or sometimes weren’t present at all. After a while, though, things settled down and almost everyone adopted the standards that were eventually documented in the *Macintosh Human Interface Guidelines*.

AppleScript is the alternate user interface to your application. Now that AppleScript has been available for two years, it’s time to move out of the “free-for-all” and develop the same consistency we’ve all come to enjoy and expect from the Macintosh experience. That’s what this column (and the work I do in the AppleScript development community) is all about — encouraging consistency. The tips I offer here reflect undocumented conventions followed by many developers I’ve worked with, as well as my own thinking about scriptability. Until the time when standards are documented in a “Macintosh Human Scriptability Guidelines,” I encourage you to adopt the techniques suggested here.

Though I’ve said it before, I’ll say it one more time: adopting the object model is the single most important factor contributing to consistency in the AppleScript language across applications of different types. One developer I know resists using the object model year after year, arguing that it “isn’t appropriate for everything.” But the fact is that the object model has been successfully applied to a whole range of applications. Every major C++ framework now supports it or has add-ons to support it, and up-and-coming languages will support it. Even if your application has only one object (such as the dictionary of a small paging program I’ve seen), just do it!

**ORGANIZING YOUR DICTIONARY**

So far in the scripting world, various developers have used different schemes in their dictionaries for organizing the events in a suite, the parameters in an event, the properties in an object, and so forth. Some organize them according to their function, others order them alphabetically, and still others don’t seem to have any scheme whatsoever (probably because scripting support was added a bit at a time or as an afterthought). For the sake of consistency across different scriptable applications, using some standard scheme is preferable.

If you’re including an entire standard suite (such as the Core suite) from AppleScript’s system dictionary (listed in the Rez files named EnglishTerminology.r, FrenchTerminology.r, and so on) and then overriding or extending the suite to add your own terms, make sure that your overrides appear in the same order as they do in the system dictionary and that extensions come after all the overrides. If you’re implementing your own terminology, either as extensions to existing suites or in your own suites, organize it as described in the following paragraphs.

When you’re adding new terms to a previously created dictionary (for example, when upgrading your application to provide deeper scripting support), remember to insert the new terms according to the same scheme or schemes you originally implemented. It’s a good idea to keep some notes in your internal design documents describing the ordering schemes you used, so that you can be consistent with your earlier work (unless you’re redoing your scripting implementation from scratch — for instance, when you’re converting from an old non–object model implementation to the object model).
Suites. So that your dictionary is consistent with dictionaries in other applications, include the standard Registry suites first (Required suite first, then the Core suite, then any other Registry suites). Then include any custom suites you create.

Events. Order commands that correspond to events in one of four ways: by likelihood of use, according to function, chronologically, or alphabetically. The method you choose will depend on how your application is used and the nature of your users. As an example of each of these schemes, I'll show how some of the Core suite verbs might be organized.

If certain commands are to be used more frequently than others, order them according to likelihood of use. Present those commands that will be used most frequently at the beginning and those seldom used at the end:

- get (more of these than anything else)
- set (quite a few of these, too)
- count (a fair amount of counting)
- make (sometimes new objects are created)
- open (sometimes they're opened)
- close (and closed)
- print (printing isn't done as frequently)
- delete (neither is deleting)
- quit (quitting is done only occasionally)

If your users will logically group the operations, use an ordering according to function. Group together commands that are related in some way:

- make (make and delete)
- delete
- open (open and close)
- close
- set (set and get)
- get
- count (the rest are unrelated)
- print
- quit

If the commands are normally used in a certain order, choose a chronological ordering. First present the commands that will be used first, followed by the commands that will be used later:

- make (this often comes first)
- open (or else opening comes first)
- set (then setting properties)
- get (and later getting properties)
- count (counting comes in the middle)
- print (printing happens later)
- close (then comes closing)
- delete (deleting is near the end)
- quit (last, we bail out)

If the commands aren’t going to be used in any particular order, or you don’t know what that order is likely to be, and there’s no logical grouping, list the commands alphabetically, as the Core suite does. Although alphabetical order isn’t as helpful as the other schemes, script writers will at least be able to find commands more easily in your application’s dictionary.

Parameters. Make an effort to list parameters in an order that encourages the writing of natural, grammatically correct sentences for commands. For example:

- make new <type class>
  [at <location reference>]
  [with data <anything>]
  [with properties <record>]

If the order of an event’s parameters doesn’t matter as far as sentence style is concerned, order them according to the frequency of likely use.

- close <reference>
  saving <yes|no|ask>
  saving in <file specification>

Object classes and properties. I’d suggest placing the outermost objects in your containment hierarchy first, objects contained in the outermost objects next, and objects that don’t contain any other objects last. Remember that every object class representing an actual object must be listed as an element of some other object, eventually leading back to the application class (the null container). Primitive class definitions and record definitions (which aren’t part of the containment hierarchy) and abstract classes (which aren’t instantiable objects but are used to hold lists of inherited properties) should be placed in the Type Definitions or Type Names suite, and clearly labeled as a record definition or abstract class. (See my article, “Designing a Scripting Implementation,” in develop Issue 21.)

Properties of objects can be ordered according to one of the schemes described above for events.

WHEN YOU ALLOW MULTIPLE VALUE TYPES
Occasionally in your dictionary you might need to specify a parameter or property for which any of several types is acceptable. Using the wild card (“*”) as the type of a parameter or property tells your user that you’ll accept anything (or at least a wide variety of mixed types). Don’t do this to be lazy or to finish your dictionary quickly; do it only if you mean it. If you
accept only one type, explicitly indicate so. If you allow two different types, you can either create a compound "type" or use identical keyword entries.

**Defining a compound “type.”** One way of handling cases where you can accept two different value types for a parameter or property is to make up a new “type” to represent a combination of acceptable types in your dictionary. This isn’t a real type that you’d have to check for or deal with in your application’s code, but instead just serves to indicate in your dictionary that your application will handle *either* type. This works particularly well when the value types are simple. For example:

```plaintext
class reference or string: Either a reference or a name can be used.
```

You can use your new “type” in a parameter or property definition as follows:

```plaintext
class connection
properties:
  window <reference or string> -- the connection’s window can be referred to either by a reference or by its name
```

To define a new type, make a new object class and place it in the Type Names suite (see my article in Issue 21).

**Using identical keyword entries.** You can also use multiple entries with identical keywords to specify alternative ways of filling in a parameter or property value. This works well when the value types are complex or are highly dissimilar. For example, the `display dialog` command has two with `icon` listings, one for specifying the icon by its resource name or ID and the other for displaying the stop, note, or caution icon:

```plaintext
display dialog <anything> -- title of dialog ...
  other parameters
  [with icon <anything>] -- name or id of the icon to display
  [with icon <stop|note|caution>] -- or display one of these system icons
```

Note the use of “or” in the second entry’s comment: make sure you use the same 4-byte ID for both parameter entries.

Although you could have many entries to show every possible individual type that a parameter or property takes, this might become confusing to the user. So I’d recommend that you use this sparingly, and when you do use it, try to limit the number of similar entries to 2.

**MAKING USE OF THE COMMENT AREA**
You can use the comment area (available for each suite, event, parameter, class, and property entry) to help clarify how your vocabulary is to be used. Since your dictionary is often the initial “window” through which a user looks to figure out what to do, descriptive comments can make the user’s task a lot easier. And remember that your users aren’t necessarily programmers, so you should avoid terms like FSSpec in your comments. I’ll give some examples to show you what I mean.

- For Boolean parameters and properties, if there are two possible states, include a description of the true and false conditions, such as “true if the script will modify the original images, false if the images will be left alone.”

- If the possible states are on and off, you need only include the true condition (“If true, then screen refresh is turned on”) or ask a question (“Is the window zoomed?”).

- For enumerations, include a general description of what the parameter or property represents; the individual enumerators should be self-explanatory. For example, “yes|no|ask -- Specifies whether or not changes should be saved before closing.”

- Don’t use the comment field to explain a set of possible numeric values when an enumeration (with descriptive enumerators) is better. Instead of “0=read, 1=unread, ...” use “read|unread|...”

- For compound “types,” describe the parameter or property, as well as the choices for value types listed: “the connection’s window (either a reference or name can be used).”

- For “anything” (unless you actually allow *any type* the user can think of), describe which specific types you allow: “[... descriptive info] (a string, file reference, alias, or list is allowed).”

- If you allow either a single item or a list, indicate so: “the file or list of files to open.”

- If the parameter or property has a default value (used when the user doesn’t include an optional parameter or set the property), mention it (this applies to values of any type): “replacing yes|no|ask -- Replace the file if it exists? (defaults to ask).”

Keep in mind that if you include an entire standard suite (such as the Core or Text suite), your own comments should reflect the style of the comments in that suite. See the Scriptable Text Editor’s dictionary as an example of fairly good comment style; it shows the standard versions of the Required, Core, and Text suites and adds some of its own terminology.
A COUPLE MORE DICTIONARY TIPS

While I’m on the subject of dictionaries, here are a couple of extra tidbits.

Use only letters and numbers for terms in dictionaries. Don’t use a hyphen (-), a slash (/), or any other nonalphanumeric characters in your dictionary entries. For example, if you use Swiss-German, AppleScript will treat it as Swiss - German (subtraction), which is not what you want; if you use Read/write, it will be treated as Read / write (division). Note that Read/write is in the standard Table suite, but it won’t compile properly.

All terms must start with letters. Using 9600 as an enumerator won’t work; you would have to use something like baud9600.

Finally, pick names for your terms that are descriptive for a user, especially a nonprogrammer. If you pick a term like x, users won’t be allowed to use x as a variable name in their scripts. For instance, instead of “x <small integer> -- the x coordinate” use “horizontal coordinate <small integer> -- the x coordinate.”

IT’S YOUR THING

Unlike writing code, designing a scripting vocabulary isn’t an exact science. It’s up to you to decide in what manner (and how effectively) humans will interact with this new interface. Applying “programming language” concepts and standards won’t always work. You need to keep an eye toward the human aspects of the AppleScript language and to work out a scheme that reflects careful attention to your users.

You may occasionally see guidelines here that aren’t completely clear-cut or that even conflict with each other, and every so often I’ll adjust what I’ve said in an earlier column. This is the nature of an evolving language. If you’re not completely at home with this, seek out an expert in scriptability design for advice. But remember, vocabulary design is by nature as much art as science.