Technical Note TN2106

Scripting Interface Guidelines

This document describes what you need to do in order to provide a clean and consistent scripting interface for your application. If you follow these guidelines

- new and experienced users will learn your application faster, because it looks and works like applications they're already familiar with
- users will accomplish their tasks quickly, because well-designed applications don't get in the user's way
- your application will be easier to document, because an intuitive interface and standard behaviors don't require as much explanation
- customer support calls will be reduced (for the reasons cited above)

Introduction
Basics
Style
    General
    Choosing Language Elements
    Naming Rules
    Comment style
Objects
    Designing Your Objects
    Object Specifiers
    Standard Classes
    Standard Properties
Commands
    Designing Your Commands
    Standard Commands
Scripting the View
    Scripting Windows
    Scripting the Selection
General Tips
Tips for Using Cocoa Scripting
References
Document Revision History

Introduction

Why do these guidelines exist?

These guidelines serve a similar purpose to the Human Interface Guidelines: to promote basic design principles and consistency in the scripting interface of all Macintosh applications. The design principles are all about being task-specific; the more closely you match the user's mental model of your application, the more useful it will be. Consistency in the scripting interface is valuable for the same reasons it is in the regular interface: it makes it easier for users to learn new applications and decreases the chances that they will make mistakes by following old habits, which in turn reduces the number of support calls you will have to field. Additionally, consistency among applications promotes AppleScript as a coherent system, not just a thin bit of glue holding together a hodgepodge of what are practically different languages.

By following these guidelines, you can make your users happier, your own life easier, and help promote AppleScript as a useful tool.

Back to Top

What is scripting?

Scripting is programming using a scripting language. That wasn't very helpful: what is a scripting language? A scripting language is a task-specific programming language, as opposed to a systems programming language such as C, C++, or Java. Systems programming languages are designed to build applications from the ground up and are very general purpose – you can build anything, but you also have to build everything, often out of very small pieces. Scripting languages, on the other hand, are designed with a particular task in mind. They can express things appropriate to the task very easily, but are awkward or even mute on other matters. For example, `awk` is very good at manipulating columnar text, and ActionScript is very good at multimedia, but you wouldn't want to use either of them for financial reporting.

Being task-specific is a good thing for several reasons. Not only does it let you write less code, but humans do much better at cognitive tasks (like programming) when the content is set in a familiar domain. Therefore, a publisher who is bewildered by pointers and bit arithmetic would still be able to write a complex script that works in terms of text flows and illustrations.

When thinking about what is task-specific, it is helpful to consider not only what the task is, but also what it is not. For example, the publisher above is concerned with box measurements and kerning, but does not want to be concerned with memory management or

declaring variable types. Accordingly, scripting languages tend to allocate and deallocate memory automatically, not require variable types, and generally take care of a lot of the details that a systems programming language would require you to deal with.

All this focus and convenience comes at a cost, of course: some scripting languages are wildly inappropriate for some problems, and programs written in a scripting language

generally run slower than an equivalent one written in a systems programming language. On the other hand, the scripter might not be able to write the faster version at all, or might simply not have the time, and an unwritten program runs only as fast as a human can do the task by hand.

For a complementary take on this subject, see John Ousterhout's paper Scripting: Higher Level Programming for the 21st Century.

Back to Top

What is scriptability?

Scriptability is the ability of your application to be controlled by a script – that is, a program – rather than a person armed with a keyboard and mouse. By making your application scriptable, users can customize it, tie it into automated workflows, and generally make it more useful to themselves in ways that were too specific for you to bother with, or that you never even imagined in the first place.

There are several ways to approach scriptability, but the most interesting and useful one is to expose your application's model layer. This term comes from the model-view-controller (MVC) design pattern: the model layer holds the application data and logic for operating on it; the view layer presents that data in a visible form, say in a window; and the controller layer mediates between them, updating the view as the model changes and updating the model as the user manipulates the view. For a full description, see The Model-View-Controller Design Pattern.

Exposing the model rather than the view has two key advantages. First, the model is generally more stable than the view. As you produce new versions of your application, buttons may move and commands may change, but what your application actually does is relatively constant. If scripting is in terms of the model, then existing scripts and scripters' knowledge stay valid. Second, the model is more useful, because it has to do with the actual task, rather than how to do it. For example, if you want to change the name of a file in the Finder, you can say to set the name of that file, rather than describing the view-oriented steps: select the file, click on the name part, and type some characters.

Back to Top

What is AppleScript?

AppleScript is a scripting language created by Apple Computer, and is the primary language for controlling scriptable Macintosh applications. It has an English-like syntax, is strongly object-oriented, and has several features that make it uniquely well-suited to working with large collections of objects. For full details, see the AppleScript Language Guide.

AppleScript approaches the problem of being task-specific in an unusual way. The base language does very little, and has no task-specific functions at all. What it does do is provide a framework into which you can plug any number of task-specific functions, namely applications. Depending on which object you are referring to, the vocabulary changes. For example, when talking to the Finder, you can speak of disks, folders, and files; when talking to Mail, you can speak of messages and mailboxes; and so on. In other words, applications are responsible for providing task-specific functions for the tasks that they know about; AppleScript provides the basic infrastructure and glue to pass information from one application to another.

application to another.

Back to Top

Basics

Your application's scripting interface is advertised to the rest of the world, both the system and human users, by your dictionary. It lists all the objects and commands your application understands, along with how they are related and simple explanations of what they mean. AppleScript reads it to determine how to compile scripts, and it is the primary reference for scripters who want to know what your application can do. (This is not to say that it should be the only reference you provide. It shouldn't; ideally, you will also provide documentation and sample scripts. The dictionary is required, though.)

This section describes some general guidelines for creating your dictionary. Later sections contain specific guidance on particular topics such as objects or commands.

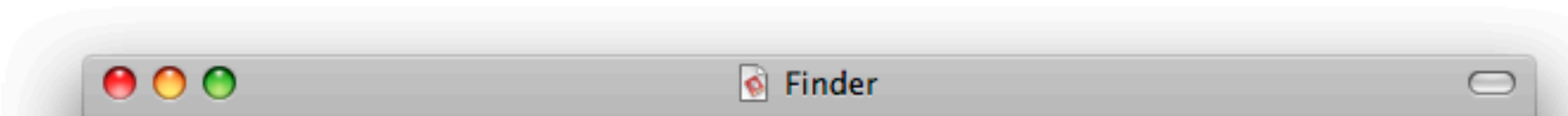
Opinion varies on whether you should write your whole dictionary first and then implement it, or write and implement iteratively bit by bit. Both have their advantages – the former makes it easier to be consistent, the latter makes it easier to ship at any time – choose whichever works for you.

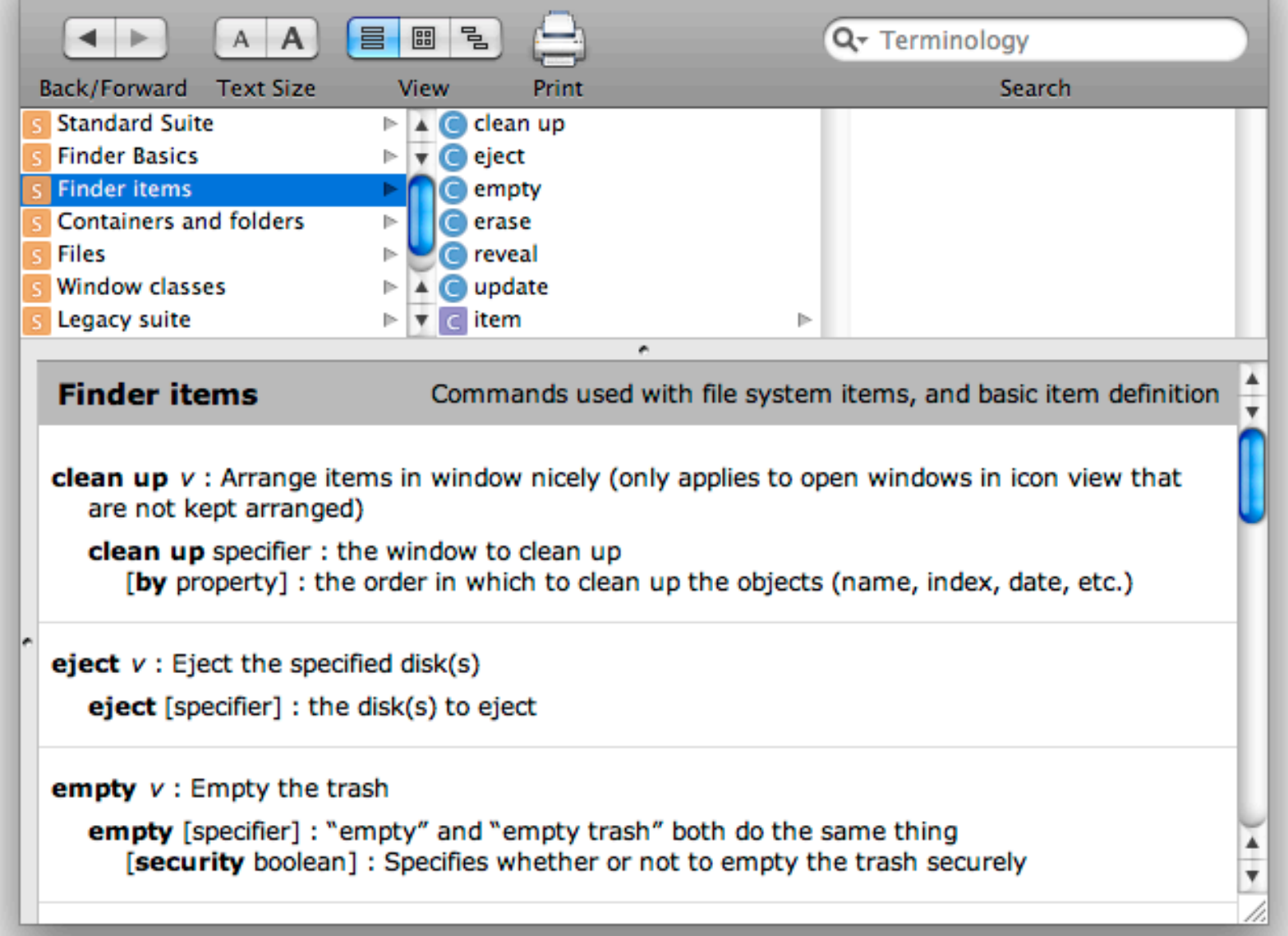
Dictionaries currently come in three different formats:

- `aete`: The original dictionary format. It is stored as a Resource Manager resource of type `aete`, hence the name. See Creating an Apple Event Terminology Extension Resource for further documentation.
- script suite: Cocoa's native format. A script suite is a pair of plist files, one suite definition (.scriptSuite) and one suite terminology (.scriptTerminology). An application contains one or more script suites. In addition to the AppleScript terms, a script suite also contains information on how the terms connect to implementation classes and methods, and tells Cocoa's built-in scripting support what to do. See Specifying Application Scripting Support for further documentation.
- sdef: Short for scripting definition, an XML-based dictionary format. sdef is the recommended format for writing a dictionary, since it is a superset of the other two formats and has explicit support for features missing or not well supported in the others, such as synonyms. While you cannot currently use an sdef directly, you can turn it into either of the other two formats using the `sdp` tool. See the sdef(5) and `sdp`(1) man pages for further documentation.

To see your dictionary as a user would, open it in Script Editor: launch Script Editor, choose Open Dictionary from the File menu, and select your application. If you are new to scripting, examine some other dictionaries. The Finder is a good example: it follows the guidelines reasonably well, and is complex enough to be interesting, but simple enough to be comprehensible.

Figure 1: Part of the Finder dictionary.





Before You Begin

Think about scriptability earlier rather than later.

Thinking about how to make your application scriptable early in the design process will save you considerable time and agony. First, it is always easier to implement a feature that is planned for from the beginning than to bolt it on at the end. Second, and more importantly, scriptability forces you to think about your model. If you know what your conceptual model objects are, you can create implementation objects that match. This not only makes scriptability easier, it's just a good design practice.

Know your model.

Scriptability means being task-specific, and being task-specific means revealing the things your application does: that is, your model. Therefore, it is essential that you have a firm grasp of your application model. What are the objects and commands that define what your application does, and how do they relate to each other? For example, Mail has mailboxes, mailboxes contain messages, messages have addressees and text and can be sent or forwarded, and so on. Notice that this says nothing about buttons, menu commands, or dialogs. Remember: model, not view.

If you can't figure out what your model is, or think you don't have one, you're probably either looking at it the wrong way or your application design needs work. Talking to your users can be helpful here – ask them what they do, and listen for the common objects and commands.

Know your users.

Being task-specific means being user task-specific, not programmer task-specific. Think of your application in terms of things users do with it, not in terms of how you, the implementor, make things happen behind the scenes. For any object or command in your dictionary, it should be immediately obvious to a normal user of your application what it is. If you have a very large and complicated application, knowing your users is also helpful in prioritizing what to make scriptable first. You are not obliged to do it all at once.

Know AppleScript.

You do not have to learn AppleScript in every detail, but you should develop a feel for typical application command syntax, and for what AppleScript will do for you (and you therefore do not have to do yourself). The AppleScript Language Guide is the official Apple documentation for AppleScript, but there are a number of third-party books available as well. Playing with a scriptable application that follows the guidelines, such as the Finder, is also a good way to learn.

Use an application framework.

A large part of implementing scriptability properly involves implementing standard behavior. Using a framework will eliminate much of the drudgery and reduce the number of errors (or at least ensure that you have the same errors as everyone else using the framework). Apple's Cocoa and Metrowerks' PowerPlant both have good support for scriptability.

Back to Top

Designing Your Dictionary

Stay on-task.

Yes, we've said it before, but it's important. Good scriptability is focused on tasks that are meaningful to the user. If you find yourself talking about error codes, bitfields, or other things that the user does not normally see, then you are drifting off-task. Also remember that tasks are based on the model – that is, the what – not the view, which is the how. If you find yourself talking about buttons and menus (and you are not working on an interface-building application), then you are attempting to script the view. Some view scripting is important – see Scripting the View – but it is not the focus. If you have trouble thinking of tasks, remember these two directives instead: talk to the user, not the programmer; and script the model, not the view.

If you find yourself having trouble expressing your model in a way a user would consider sane, it may be a sign that your model needs fixing. (It may also be a sign that you need a break, so try that, too.) This happens sometimes: viewing your model from a different angle can reveal problems that were not apparent before. Perhaps one complex operation needs to be split, or two merged, or maybe you didn't understand the task domain as well as you thought you did. Go ahead and fix it, and fix it in your graphical interface too, if you can – you'll have a better application for it.

Be object-oriented.

Being object-oriented means that your scripting is mostly organized around objects, not commands. Objects can respond to commands – documents can be printed, messages can be sent, and so on – but the focus is on the objects, not the commands. This has several benefits:

First, it matches how the rest of AppleScript, and to some extent the Macintosh as a whole, works. It is therefore more consistent and makes more sense to users.

Second, it lets you exploit various features of AppleScript, like `tell` blocks and `whose` clauses, so users can create more concise and powerful scripts.

Third, it lets you create a smaller, more comprehensible dictionary, because you can apply a multiplicative effect: `n` commands times `m` objects yields `n×m` actions, but using only `n+m` terms. Consider an application that defines a `widget` object with eight properties. To make getters and setters for all the properties using only commands, we would need 16 commands with distressingly repetitive names: `GetWidgetName`, `SetWidgetName`, `GetWidgetColor`, and so on. Being object-oriented, however, lets us define the same power using only 11 terms: `widget`, `get`, `set`, and one for each property, with the added bonus that `get` and `set` can be re-used.

Be accurate.

Make sure you actually support everything you claim to in your dictionary. Be conservative in your dictionary if you need to; sdef provides a way to hide terms if you feel they are not yet ready for prime time. (So does the older `aete` format, but it's much harder to use.)

On a similar point, make sure that the types your dictionary defines for properties and parameters are accurate. Bear in mind that accurate refers to the scripter's (that is, the user's) perspective, not the implementor's. For example, there is no file URL type in AppleScript, but there is a file type, so even though your application fetches the parameter as `typeFileURL`, define it in your dictionary as `file`.

Don't re-invent the wheel.

The less you invent, the more consistent you will be, and the less your users will need to learn.

- Take advantage of inheritance – if two or more classes have common aspects, use inheritance to make a common base class for them so your dictionary doesn't repeat itself.
- Learn from the system – AppleScript defines a number of common objects, properties, and commands.
- Learn from the market – if another application already does something that you do, look for common terminology.

A corollary to this is that you will probably not need to define many of your own commands.

The standard commands take care of basic application functions and object manipulation (`open`, `quit`, `make`, `delete`, and so on – see Commands for a complete list.) For example, Sketch, a sample drawing program, does not define any of its own commands – everything Sketch does can be expressed in terms of the standard commands, mostly by getting and setting object properties.

It's OK to do less than your GUI.

Some operations possible through your graphical user interface may not be appropriate for a scripting interface, because they assume a level of dynamic interaction that simply isn't practical – for example, freehand sketching, or scrubbing through a movie. Do not feel obligated to include a function in your scripting interface just because it exists in your graphical interface.

Also, it may not be practical to make everything in your application scriptable all at once: you don't have the time, you expect a feature to change in the next release, whatever. Provide what you can, make it useful, and do the rest later.

It's OK to do more than your GUI.

Scriptability is a great way to expose features in your application without having to implement a graphical interface for them. Sometimes these features are implicit in how AppleScript works: for example, the Finder has no command to delete files that meet a condition, but the `delete` command combined with a filter object specifier will accomplish just that, as in `delete every file whose name ends with ".jpg"`. You can also expose aspects of your model via scripting but not via the graphical interface, whether because you haven't got the time, don't know how, or simply don't want to clutter up the GUI. For example, Address Book exposed nickname and prefix properties for people via AppleScript one release before adding any graphical interface for them.

Get feedback.

Find some users who care about scripting, let them play with your application, and listen to what they say. You may find some on staff already – testers tend to be fond of scripting, because it makes their jobs easier.

Back to Top

When You Think You're Done

Test, test, test.

Scriptability needs testing just like any other feature. Knowledge of AppleScript will be very helpful here, not just so you can write scripts, but so you can truly exercise the language. Newcomers often do not grasp the implications of object specifiers, and create commands that do not work with ranges or filter clauses.

Get more feedback.

Internal development and testing is all well and good, but nothing substitutes for real-world usage. Let real users try your scriptable application. If you discover a mistake in your scriptability after you ship, you can still change it without breaking existing scripts by exploiting synonyms – see the sdef man page for details.

Back to Top

Style

This section describes style guidelines for your dictionary: organization, how to name things, comment style, and so on.

General

Use suites as appropriate.

Dictionaries are broken into suites, which are clusters of related classes and commands. These are purely for the reader's benefit, and have no impact on your application or AppleScript. There is no technical limit, either upper or lower, on the number of items in a suite, but ten to fifteen is considered a comfortable size. However, do not force artificial distinctions just to break up a large suite or combine small ones – the ultimate point is to help the user find things.

Put system-defined suites first, followed by your own. Sort them in order of generality or frequency of use, or, failing that, alphabetically.

Within a suite, and for properties in a class, order items in some sensible manner. For a sufficiently small group, you may sort by importance or frequency of use, but this can be hard to judge. Alphabetical ordering is always a good fallback.

Within a command, order the parameters in a way that makes sense in a sentence. If this is impossible, use alphabetic order.

Be accurate with types.

When specifying the type of a property or parameter, use the most specific type that is both accurate and meaningful to a user. Often, this is a simple choice of a single application-defined class or a primitive data type, but sometimes it's more interesting:

- If there are several possible types that all descend from a common base class, and all descendants of the base class are legal, use the base class.
- If the type is a list of some other type, say so. (See sdef's `type="list of ..."` or `aete`'s `listOfItems` bit.)
- Do not claim to accept all types unless you really mean it. (See sdef's `any` type, or `aete`'s `typeWildCard`.)
- If you support two or more unrelated types – that is, not related by inheritance, as in

- If you support two or more unrelated types – that is, not related by inheritance, as in the first case – say so. (See sdef's `type="T1 | T2 | ..."`. Doing this in `aete` is possible, but awkward.)

Things that refer to objects should be objects.

If a property or parameter refers to an object, its type should be an actual object, never data that signifies an object. Consider this command from many web browsers that breaks this guideline (and several others):

> **CloseWindow**
> [**ID** integer] -- ID of the window to close. (Use -1 for topmost)
> [**Title** string] -- Title of the window to close.

The correct way to do this is to define a `close` command that operates on `window` objects, which can then be specified by name, id, or several other ways. This allows more flexibility, and is simpler to define.

A few words about files

Files in AppleScript are considered to be objects. Therefore, properties and parameters that point to files should also be `file` objects. Do not use file path strings for this purpose. Similarly, do not have use file path strings anywhere – use a `file` object, and let scripts use the `path` property of files if they really need to. (Cocoa Scripting got this wrong when it was first implemented. This is being corrected; you should not consider it as granting permission.) Use sdef's `file` type; it will do the correct thing (or at least the most correct thing possible).

Back to Top ▲

Choosing Language Elements

Properties vs. elements

Objects may contain other objects; these are either named properties (`inbox of application "Mail"`) or indexed elements (`mailbox 3 of application "Mail"`). The general rule is that if there is exactly one object, use a property; if there can be any number, use an element. For complete details, see Objects.

Properties vs. commands

Sometimes setting a property can cause an immediate change on the screen. In deciding whether to use a property in this situation, a helpful rule is: When an action is initiated, use a command; when an attribute changes (even if it produces immediate visible results), use a property. Another way of looking at this is if a visible change is immediate, use a property, but if an action has a duration, use a command.

For example, the following command causes an immediate change on the screen:

```
set the font of the third paragraph to "Courier"
```

Even though setting the font property creates a visible change, the font is still an attribute of the text, not an action. On the other hand, naming a property or enumerator `playing`, as shown in the next two commands, is a poor choice, because playing actually initiates an action:

```
set playing to true
set [the] status to playing
```

The `playing` enumerator value in the second command is fine for obtaining state information, but a status property should be read-only. Instead of creating a property to control an action, use a command. Commands such as play or start playing are better suited for actions, as shown here:

```
play the movie "Wowie Zowie"
start playing the movie "Wowie Zowie"
```

Note that the commands are `play` and `start playing`, not `play movie` and `start playing movie`. In a proper scripting model, `movie` would be an object class.

Enumerations

An enumeration is a set of constants (enumerators) which represent a fixed set of choices. Use an enumeration as the type of a parameter or property whenever there is a choice to be made from a specific list of possibilities. This helps the user see that there is a choice, and lets them know exactly what the choices are.

Table 1 : Enumerations

BAD	`property status integer -- 0=cold, 1=warm, 2=hot set status to 1`
GOOD	`property status cold/warm/hot set status to warm`

Back to Top

Naming Rules

Use user terms, not implementation terms.

Remember that you are speaking to a user, and should therefore use the terms they will find familiar. Ideally, the terms used in your graphical interface, scripting, and documentation

should all match.

Use the the test.

AppleScript allows you to add or remove the word `the` almost anywhere in a script without changing the meaning of the script. Many script writers precede object and property names with the word `the` to make their scripts easier to read. Writing your test scripts in this way helps you determine the degree to which your property names facilitate forming natural sentences.

```
set the service to "America Online"
if the priority is high then ...
```

Do not use `the` as part of a term. Scripters are accustomed to `the` being optional, but using it in a term makes it required.

Use lowercase.

Use all lowercase letters for your terms, except as required for proper names, trade names, and acronyms.

Table 2 : Capitalization

BAD	`Title`
GOOD	`title`
OK	`Macintosh, IEEE`

Use multiple words, not InterCaps or underscores.

If a term consists of more than one word, separate the words with spaces. Do not join the words together with capital letters or underscores. Again, trade names are an exception to this.

Table 3 : Examples.

BAD	`TransferProtocol, network_address`
GOOD	`transfer protocol, network address`
OK	`AppleTalk`

Terms must follow AppleScript identifier rules.

Each word in a term must be a valid AppleScript identifier. That means they must begin with a letter or underscore, and be followed by any number of letters, numbers or underscores, the same as C-like languages. Punctuation is not allowed; underscores are not recommended as mentioned above.

Do not create commands that look like objects, or objects that look like commands.

Creating terms that look like other parts of speech will confuse scripters and lead them to try things that will not work. Consider the Standard Additions command `set the clipboard to`: seeing it in a script, you might think that it is a `set` command operating on a `clipboard` property, but it really isn't, so the supposedly equivalent `copy x to the clipboard` would not work.

Do not start property names with imperative verbs.

Starting property names with verbs leads to confusion when the property appears in the middle of a sentence. For example, naming a property `disable call waiting` leads to commands that do not read smoothly:

```
set disable call waiting to true
if disable call waiting then ...
```

Also, the name disable call waiting looks like a command. A user might be tempted to say

```
disable call waiting
```

This will compile and run, but does not disable anything – it merely returns the current value of the property. You can prevent such confusion by using a participle instead. This is somewhat clearer:

```
set call waiting enabled to false
if not call waiting enabled ...
```

Even better would be to name the property `call waiting` and use an enumeration as its value type. The choices `enabled` and `disabled` allow grammatically correct sentences:

```
set call waiting to enabled
if call waiting is disabled ...
```

Avoid beginning terms with language keywords.

Using a language keyword as the first word in a term may make it impossible to use that

keyword in certain contexts. For example, consider an application that defines a `mailbox` class that contains `message` objects. Adding an application property named `in mailbox` would prevent scripters from using the keyword `in` as a synonym for `of` in some cases: `messages in mailbox 1` would be parsed as `messages in mailbox 1`, which is illegal, instead of the intended `messages in mailbox 1`.

It is possible to successfully begin terms with keywords – consider the Standard Additions command `set the clipboard to` – but it is not recommended. Terms may use language keywords in the middle or at the end of terms, but see some of the following guidelines.

Table 4 : Reserved words in AppleScript 1

after	does	get	my	second	to
and	eighth	given	ninth	set	transaction
as	else	global	not	seventh	true
back	end	if	of	sixth	try
before	equal	ignoring	on	some	until
beginning	equals	in	or	tell	where
behind	error	into	prop	tenth	while
but	every	is	property	that	whose
by	exit	it	put	the	with
considering	false	its	ref	then	without
contain	fifth	last	reference	third	
contains	first	local	repeat	through	
continue	fourth	me	return	thru	
copy	from	middle	returning	timeout	
div	front	mod	script	times	

Avoid using 'is' in Boolean property and parameter names.

This is a specific case of the previous rule: `is` is a reserved word meaning is equal to. For example, instead of `is minimized`, `is encrypted`, or `is busy`, use `minimized`, `encrypted`, and `busy`.

Also, the AppleScript parser pulls a few special tricks on Boolean properties and parameters that make them read better without the is.

AppleScript will change `true` and `false` in Boolean parameters to `with` and `without`. For example, if a scripter writes

```
send message "Fred" queuing true
```

it compiles to

```
send message "Fred" with queuing
```

Also, tests of Boolean properties can be written specially: instead of saying `property` of `object` is true (or false), you can say `object` is (or is not) `property`. For example, the following two lines are equivalent:

```
if miniaturized of window 1 is true then ...
if window 1 is miniaturized then ...
```

If the property were named `is miniaturized`, the second line would have to be written `if window 1 is is miniaturized`, which is awkward. Notice that these special forms read best if the property name is simply an adjective, not an adjectival phrase such as `has x` or `wants x`.

Avoid using 'of' in a name.

`Of` is a reserved word used between parts of an object specifier. AppleScript will let you get away with this, but it can be very confusing to read, because it makes the term look like a property of an object, which it isn't.

Avoid using 'end' as the first word in a command.

Several AppleScript constructs use `end` to end a block: `end if`, `end repeat`, and so on. In these cases, AppleScript will fill in the word after the `end` automatically. Users may be tempted to try the same thing with your command, and it won't work.

Table 5 : Examples of terms.

BAD	`begin animating, end animating`
GOOD	`start animating, stop animating`

Back to Top ▲

Comment style

Any dictionary term can have an associated comment (sdef's and suite terminology's `description` attributes, or `aete`'s comment fields). Use them 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. Remember that your users are not necessarily programmers, so avoid implementation terms in your comments.

- 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 starting with is or does (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 | …
- If you allow more than one type (see sdef's `type="T1 | T2 | ..."`), include a description of the choices for the types listed: the connection's window (either a reference or name can be used).
- 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).

Back to Top

Objects

Because AppleScript is so object-centric, defining a sensible object and class hierarchy and making it behave in all the standard ways is perhaps the most important part of creating a good scripting interface for your application. To paraphrase Fred Brooks slightly: Show me your code and conceal your objects, and I shall continue to be mystified. Show me your objects, and I won't usually need your code; it will be obvious.

Designing Your Objects

As always, the basic rule is be task specific. Reveal objects that have a direct bearing on things the user will do with your application. Expose the model, not the interface. Do not reveal implementation details – users will not care, or worse, it will confuse them.

There are two different hierarchies to consider: the inheritance hierarchy – what behaves like what – and the containment hierarchy – who owns what. Make your hierarchies match what a user would expect.

Inheritance

AppleScript supports single inheritance, so you can define a class to be just like another class, but with additional properties or elements. A subclass does not have to define any new

parts; sometimes just being a different class is enough. Mail's `recipient` classes do this.

Use inheritance to effectively shrink your dictionary by removing redundant information. If several classes share common elements or properties, make them all inherit from a base class that defines the common parts. Do not use inheritance if a class has only one descendant, because that just complicates the dictionary.

Back to Top

Containment

AppleScript defines two kinds of containment relationships: elements and properties. In entity-relationship modeling terms, an element is a to-many relationship, and a property is either an attribute or a to-one relationship. (Entity-relationship modeling distinguishes between primitive data and other objects; AppleScript does not.) A property is a single value, while there may be any number of elements. Special cases such as zero-or-one or any-number-in-a-range can be emulated using properties or elements with additional behavior, as shown here:

Table 6 : Containment terminology.

how many?	use	example
exactly one	property	`name`
any number	element	`documents`
zero or one	property, value may be `missing value`	`current printer`
exactly `n`, `n` > 1	element, but can't `make` or `delete`	`domains of application "System Events"`

In general, prefer an element to a property whose value is a list of objects – it is more consistent and is easier to implement. In particular, do not do something like this:

```
class widget (pl. widgets)...

class box:

    property widgets: list of widget
```

This is especially evil, because it makes `widget` look like an element of `box` when it really isn't. For example, `widgets of box 1` works, and looks like an every specifier, but the supposedly equivalent `every widget of box 1` will fail, because it was really a

property specifier.

If you do implement such a property (see the standard `selection` property for an

If you do implement such a property (see the standard `selection` property for an example), make sure that it works exactly as a normal container in object specifiers – element subclasses, filters, and so on should all work.

Back to Top ▲

Object Specifiers

All objects in an application, aside from the root `application` object, are contained by some other object. To select a particular object or set of objects, a script uses an object specifier, or specifier for short, which is an expression of the form `object-expression of container`. `Container` is itself an object specifier, so the application recursively works its way up the containment chain until it reaches the (often implied) root `application` object, which contains everything. For example:

```
length of word 1 of every paragraph of document "Smith Project"
```

The underlined portions show each component. Notice that each component uses a different key form. The complete set of key forms is listed in the table below. Most key forms imply the existence of a particular property for that object: if the object has that property, then it should support that key form, and vice versa.

Table 7 : Object Specifiers

key form	property	example
arbitrary	—	`some file of the startup disk`
every	index	`every word of document 1`
filter	—	`every document whose name starts with "Smith"` `first paragraph where it contains "Anderson"`
ID	id	`application file id "com.apple.finder"` `message viewer id 4871536`
index	index	`word 3` `paragraph index -1`
middle	index	`middle shape of drawing 1`
name	name	`folder "Bob"` `file named "Smith Project"`
property	—	`name of document 1`
range	index	`words 3 through 8` `every word from character 1 to paragraph 3`
relative	index	`word after word 4`

Additionally, allowable key forms for a container are determined by the nature of the container and its elements. This also determines what locations, if any, are meaningful for the `make` and `move` commands. There are, broadly, three types of containers:

In an ordered container, the user determines exactly how the items are arranged. Any item

can be placed before (or after) any other item using `make` or `move`, and it will stay there. Paragraphs of a text document are ordered, as are items in an AppleScript list.

In an unordered container, the data itself determines the order of the items – for example, alphabetical order – or there simply is no particular order. While you can ask for the `nth` item, iterate through all of them, or speak of the item before or after another one, asking to make an item at or move it to a particular position is meaningless – all you can do is ask to put it in the container. People in Address Book, files in the Finder, and items in an AppleScript record are all unordered.

In an uncounted container, the application does not know how many items there are. All index forms are meaningless, as is the idea of `every`. To get an item, you must know something else about it: its name, its id, some property, or similar. Web servers on the Internet are uncounted.

Table 8 : Object Specifier Types

container type	example	access by	make at
ordered	an AppleScript list	indexed, relative, arbitrary	indexed, relative, container
unordered	an AppleScript record, people in Address Book	indexed, relative, arbitrary	container
uncounted	all Web servers on the Internet	–	container

Key Forms

This section gives behavior details for each of the key forms. It does not give syntax or usage details; for that, see Chapter 5 of the AppleScript Language Guide.

Arbitrary

If you support this key form, it should return a genuinely random object, not always the same one. This key form is useful whenever a script needs random behavior. For example, a script could select a random signature in Mail by saying `some signature`, rather than the more complicated `signature (random number from 1 to (count signatures))`.

Every

`every item` is equivalent to `items 1 through -1`, except that `every` returns an empty list if there are no elements, instead of an error. See Range for more details.

Filter

For historical reasons, Filter is considered its own key form, though it would be more accurate to think of it as a modifier to index-based key forms (index, arbitrary, middle, range, and every). While the filter key form is most often used with a test on a property of the object (such as `files whose name contains "Smith"`), tests on an element or the object itself are also valid. For example:

the object itself are also valid. For example:

```
paragraphs where it starts with "The"
albums where it contains photo "White Rabbit"
documents whose first paragraph contains the word "Metacortex"
```

In theory, any boolean expression should be valid, but AppleScript currently cannot deal with function calls in a filter expression.

If an object under consideration does not have an attribute specified in the filter, it is not an error – consider the value to be `missing value` and respond appropriately.

ID

Returns a single object whose `id` property matches the specified value. ID data should not honor the current AppleScript considerations, such as `considering case`. Treat IDs as always case-insensitive, or as always case-sensitive if case is relevant (that is, if `"ijkl"` and `"IJKL"` are both valid and different).

Index

Item indicies should match the user's model of a first-to-last or front-to-back ordering. (In fact, `first` and `front` are synonyms in AppleScript, as are `last` and `back`.) This means that `window 1` should always be the front window, `table 1` should be the first table in the document, and so on. Negative indicies go backwards from the end, so `item -1` and `last item` mean the same thing, `item -2` is the second-to-last item, and so on.

If the items are unordered as defined above, then ideally the index should match how the items are currently displayed. Failing that, pick a consistent order – say, alphabetical – and use that.

Named

Returns a single object whose `name` property matches the specified string. String matching should honor the current AppleScript considerations, such as `considering case` or `ignoring diacriticals`. Name data may come to your application as plain or Unicode text, so be prepared to handle either, even if you do not use full Unicode internally.

Range

Ranges may be specified as either two indicies or two endpoint objects, which are included in

the range. If either endpoint does not exist, or if they are not both in the same container, return an error. Guidelines for the meaning of indicies apply as for the index key form above.

The order of the endpoints does not matter – always return the objects in order, lowest index to highest. For example, `words 8 through 1` should return the same thing as `words 1 through 8`.

Text presents a special case: because of how its containment works, it is possible to specify two endpoints where one is inside the other. (Normally, this would be an invalid range, because the two objects would have different containers.) To resolve this, find the beginnings of both endpoints, start from the one closest to the beginning of the text, and then go to the

end of the other endpoint object. For example:

```
set t to "Through three cheese trees three free fleas flew"

words from paragraph 1 to word 3 of t
--> {"Through", "three", "cheese"}

characters from character 10 to word 2 of t
--> {"h", "r", "e", "e"}
```

Range specifiers always return a list, even if the range contains only one object (or, in the case of every, no objects). If a range reference is used as the container for another property or object, the result is that property or object of every item in the range. For example:

```
name of files 2 thru 3 of home
--> {"Memory report", "BBEdit update.dmg"}
```

The list of values returned should always parallel the list that would have been returned for the range. That is, it should be in the same order and have the same number of values. For example:

```
every duck --> {mallard "Daffy", pintail "Marvin", decoy duck "Q-
36"}

name of every duck
--> {"Daffy", "Marvin", "Q-36"}
```

This applies even if some of the objects do not have the specified property or element. For such objects, return `missing value`. Continuing the example, consider a feet are webbed property, given that decoys do not have feet:

```
feet are webbed of every duck
--> {true, true, missing value}
```

If the specified element is another range, then the result is a list of lists. For example:

```
every duck of every flock
--> {{mallard "Ned", mallard "Ted"}, {pintail "Ann", pintail
"Dan"}}
```

Relative

Relative specifiers are of the form `class (before | after) specifier`, where

`specifier` may be any other specifier, including another relative specifier. Therefore, `word after word after word 1` is a perfectly valid way to describe the third word. This may seem peculiar, but is relevant when working with text selections: if the selection is an insertion point, say, `insertion point after word 12` the word after that point would be `word after insertion point after word 12`.

Back to Top

Elements and Inheritance

A container does not need to define a distinct element relationship for every class it holds – it may imply them by inheritance. If an object contains objects of some class, a specifier may specify any subclass and the request will be restricted to objects of that class. For example, consider the fictitious application DuckTracker:

class **duck**: …
class **canvasback**: inherits from **duck** …
class **mallard**: inherits from **duck** …
class **pintail**: inherits from **duck** …
class **application**
    element **duck** by index, by name…

Even though it only explicitly specifies `duck` as an element, a script could ask for `every mallard` and receive a list of all the mallards.

Back to Top

Returning Object Specifiers

An object in your application can typically be referenced in one of several ways. For example, `window 1`, `window "Lister"`, and `window id 743` might all refer to the same window. If you need to return an object, though – say a script asks to get the first window whose name begins with "L" – you will need to pick a particular key form. The form you choose is known as a canonical object specifier, and should be the most reliable way to refer to that object. That is, it stands the best chance of referring to the same object later as things change.

The general rule for the form to use is:

- If the object has an `id` property, use that. (`window id 743`)
- If the object has a `name` property, use that. (`window "Lister"`)
- Otherwise, use its index (`window 1`). The index should always be positive.

The class of an object specifier should be the actual class of the object. Continuing the DuckTracker example from above, this would be something like `mallard id 8237`, not `duck id 8237`, even though both refer to the same object.

Back to Top

Unusual Concerns

Many-to-one and many-to-many relationships

In most applications, all object relationships are one-to-one or one-to-many, the result being that all objects are contained by exactly one other object. Some applications, however, have many-to-one or many-to-many relationships, so a single object may appear in several different containers. iPhoto behaves like this: a `photo` object may be in any number of `albums`. There is nothing wrong with this model, but it does require certain adjustments.

First, you must support the commands `add` and `remove`. These commands are designed to only manipulate containment, as opposed to creating and destroying objects like `make` and `delete` do. The distinction is important because `make` and `delete` do not have the proper meanings for some tasks. Say you want to add an existing photo to an album in iPhoto. `make` is not really appropriate, since you're not making anything. Similarly, say you want to remove a photo from an album. `delete` is ambiguous: will it merely remove the photo from that album, or will it delete the photo entirely? You could resolve the problem without different commands by introducing a photo reference object, except that there is no such thing in the user model, and would therefore violate the stay on-task guideline.

Second, is it possible for your objects (call them photos, to continue the iPhoto analogy) to not belong to any specific container (album)? For example, in iPhoto, a photo may not be in any album. If this is the case, then you will need to designate a single master container that holds all the photos; they may be added to and removed from albums from there. (iPhoto uses `application` for this.) If not, then you do not need a master container, but removing a photo from its last album must delete it.

Non-symmetric specifiers

In general, if an object is obtained by a particular property, then the object's property ought to match. For example, `name of thing "Bob"` ought to always be `"Bob"`, `id of thing id 427` ought to always be `427`, and so on.

In some applications, however, this may not be the case. Typically, the reason is that an object may have more than one valid name (or id, and so on). For example, the Finder's `application file` class responds to an id of either the old-style creator type or the new-style bundle id: `application file id "emal"` and `application file id "com.apple.mail"` both return the same thing.

It is perfectly acceptable for multiple key data to return the same object. (If this creates ambiguities, then see below.) However, you should have some way for the user to discover all the different data an object can be specified by. The basic property (for example, `name`) should reflect the canonical data; add other properties for the rest. For example, if a file's extension is hidden in the Finder, it will respond to either the visible name (without the extension) or on-disk name (with the extension). The on-disk name is considered the canonical name; the visible name is available via the `displayed name` property.

Non-unique key forms

Typically, names and IDs are unique at least within in a container, so there is only one object `folder "Bob"` could possibly return. However, this is not always the case – for example, in some applications, names are purely for the user's reference, and the application makes no

in some applications, names are purely for the user's reference, and the application makes no attempt to keep them unique.

For such applications, invoke the be consistent about return types rule: named and indexed key forms normally return a single object, so they should still do so even in the face of ambiguous objects. If one object is a better match than the other, return that one; if there is no best match, return the first one. However, ensure that users can select all the matching objects using a filter expression of some sort, such as `every item whose name is ...`

Back to Top ▲

Standard Classes

There are a few standard classes that applications should define. The only one that an application must define is `application`, the root of the containment hierarchy. (The root of the inheritance hierarchy, `item`, is defined by AppleScript itself and does not need to appear in your dictionary.) The properties shown here are merely starting points; a real application will need to define more, since the ones shown here do almost nothing. Classes and properties may be omitted if they are not relevant to an application. For example, not all applications have documents, or even a selection. Similarly, properties marked here as read-only might be writable (or vice versa) in some applications.

application
    name text [r/o] -- The name of the application.
    frontmost boolean -- Is this the frontmost application?
    version text [r/o] -- The short version string of the application.
    selection `varies` -- The current selection; the exact type depends on the application.
    element **window**
    element **document**

`application` is the root of the containment hierarchy. That is, it contains, directly or indirectly, all the objects belonging to the application. `window` elements are only those application windows that contain other scriptable objects, such as document windows – modal dialogs, sheets, and formatting or tool palettes should not be included. (Palettes may be included as properties, however.)

document
    name text [r/o] -- The name of the document.
    modified boolean [r/o] -- Does the document have unsaved changes?
    file file [r/o] -- The file for the document; may be missing value if the document has never been saved.

An application document. The `name` property is user-visible name, not the on-disk name. `name` and `file` are typically read-only; use the `save` command to change them.

window

name text [r/o] -- The title of the window.
bounds rectangle -- The bounding rectangle of the window.
closeable boolean [r/o] -- Can the window be closed?
minimizable boolean [r/o] -- Can the window be minimized?
minimized boolean -- Is the is window minimized?
resizable boolean [r/o] -- Can the window be resized?
visible boolean -- Is the window visible?
zoomable boolean [r/o] -- Can the window be zoomed?
zoomed boolean -- Is the window zoomed?

A window, which may be a document window or palette, but not a sheet or modal dialog. `bounds` is given as a list of four numbers: {top, left, bottom, right}, with the origin at the top left of the display and x, y values increasing to the right and down, respectively. (In other words, a Quickdraw rectangle. This differs from AppleScript Studio, which returns bounds as {left, bottom, right, top}, with the origin at the bottom left and x, y values increasing to the right and up.) `visible` refers to whether or not the window is visible as with a show/hide command, not whether or not the window has been moved to lie mostly off-screen.

item
class class [r/o] -- The class of the object.
properties record -- All the properties of the object.

The root of the inheritance hierarchy, and the ultimate ancestor of all classes. See below for complete descriptions of the `class` and `properties` properties. Since `item` is defined by AppleScript, it should not appear in your dictionary. In particular, do not redefine `item` with additional properties or elements – create your own class with a different name. Classes do not need to explicitly define that they inherit from `item` – if there is no other parent, `item` is assumed.

Back to Top ▲

Standard Properties

class

The class of the object. This should always be the precise class, never a superclass, and is always read-only. You do not need to explicitly define `class` in your dictionary, since it is defined by `item`, which is the ultimate ancestor of all classes.

container

Objects in a containment hierarchy may define a container property that points to their enclosing object. (This property is purely optional because it is often difficult to implement.) The precise name depends on the possible container classes: if there is exactly one, name the property the same as the class. For example, messages in Mail always belong to a mailbox, and therefore have a `mailbox` property. If there are several possibilities, use the generic

term `container`. For example, a file in the Finder may belong to either a folder or a disk. Do not use the term `parent`, since that refers to the object's immediate ancestor in the inheritance hierarchy.

The container property is read-only. Writing to the container property to move objects is not recommended; use the `move` command instead.

Objects in many-to-`x` relationships may not have a single meaningful container. Such objects should omit this property, but be sure to support specifiers of the form `containers where it contains object`.

contents

If an object's contents can be represented as a single value, it should define a `contents` property. For example, documents in a word processor would define a `contents` property that returned all the text as a string. `contents` is usually writable.

entire contents

Recursive containers – that is, objects that have their own class as elements – should define

an `entire contents` property, which returns a flat list containing every descendant element. `entire contents` itself is always read-only, but elements of it may be writable.

id

A value that uniquely identifies the object. IDs are never localized and are typically not under the user's control and therefore are read-only. They should at least be unique within a container – in most applications, they are unique within the entire application – and must remain valid for at least the lifetime of the application process or the object, whichever ends first.

The type of value may be anything: common choices are an integer, a UUID, or a bundle identifier. The type of value for any particular class should always be the same.

name

An object's `name` property is its user-visible name, never a system identifier. (Some applications may have a somewhat flexible notion of what the user-visible name is; see Non-symmetric Specifiers above.) `name` is usually writable, though in some applications it may be read-only.

properties

All objects should have a `properties` property, which returns a record containing all the properties of the object. You do not need to explicitly define `properties` in your

dictionary, since it is defined by `item`, which is the ultimate ancestor of all classes. If any properties are writable, then `properties` is as well, and can accept a record containing any number of the properties for that object. For example:

get the properties of paragraph 4 -- Returns font, size, style, and so on. set the properties of paragraph 4 to {font:"Helvetica", size:14} -- Sets just the font and size.

The `properties` property need not include every property of an object. Typically, properties that can be derived from other properties are not included, such as the `reverse` property of a list.

Back to Top

Commands

Commands in AppleScript are defined independently of any particular object; a class may then list the commands that it responds to. To Java and Objective-C programmers, commands therefore resemble protocols more than member functions. There is currently no

straightforward way to define a command differently (say, with different parameters) for a particular class. The best you can do is either to list all possible parameters and use the comments to say which parameters are meaningful when, or to define separate commands with different names.

Designing Your Commands

Be English-like.

AppleScript commands normally look like imperative English sentences:

```
verb [noun] [preposition value]...

save the front document in file "Goofballs:Razor"
```

Try to emulate this style with your own commands. However, bear in mind that this is a guideline, so your English style does not have to be perfect. Also, you are by no means obliged to support all possible English sentences. Synonyms are fine where they make sense, but don't go overboard.

Don't invent commands if you don't have to.

Try to use the standard commands where it makes sense, and only create your own custom commands if there is a clear need. For example, Mail's send action does not fit into the standard commands in any obvious way, so it gets its own command. On the other hand, the Finder does not define a `rename` command, because it can simply set the value of the object's `name` property.

Singular specifiers return values, ranges and 'every' return lists.

In general, commands that return a result should return the same shape of result as the

direct parameter specifier: singular specifiers return bare values, range and `every` specifiers return lists. (There are exceptions to this; see `count` and `exists`.) A range specifier where one or both endpoints do not exist is an error; an `every` specifier that specifies no objects is not. This rule also applies to filtered range and `every` specifiers. For example:

```
-- Singular specifiers like index return a value.
get name of person 1
--> "Fillmore"

-- Range and "every" specifiers return a list...
get name of every person
--> {"Fillmore", "Poppleton"}


-- ...even if there is only one object...
get name of every person whose name starts with "P"
--> {"Poppleton"}

-- ...or even no objects for "every"...
get name of every person whose name starts with "Q"
--> {}

-- ...but both endpoints must exist for a range.
get name of people 1 through 17
--> Error: Can't get person 17.
```

Commands that break this rule – for example, given a range they sometimes return a value and sometimes a list – are annoying to use, because they require the user to write logic to find out what happened. See Key Forms for more details.

Exploit the target.

The subject of the innermost `tell` block is the target. Use the target to complete parameters that the script omits. In some cases, AppleScript will do this for you: since the target usually corresponds to the direct parameter, AppleScript will automatically pass the target as the direct parameter if the command requires one and the script does not supply it. In some cases, however, the target makes sense as a prepositional parameter. Consider `add x to y`: it would be reasonable to say `tell y to add x` instead. Your application can detect the missing `to` parameter and pick the target out of the subject parameter, an extra parameter that AppleScript passes the target in.

Use error return parameters, not status codes.

The return value of an AppleScript command should always be the value or object result of the command, never a status code or message. If there is no result – that is, the command only performs an action – return nothing.

To signal an error, return a non-zero error code from your event handler, or set the `kOSAErrorNumber` parameter in the reply. Use standard error codes where appropriate, in particular the AppleEvent error range from -1700 to -1799 (described in Result Codes in Apple Event Manager Reference). Use of the additional error parameters such as `kOSAErrorOffendingObject` (see OSA.h for a complete list) is highly recommended – the idea is to return as specific an error as possible. For example, say a script passes a list where your command wanted a number:

Table 9 : Error handling.

BAD	Return `paramErr` `→ error: Parameter error.`
BETTER	Return `errAECoercionFail` `→ error: Can't make some data into the expected type.`
BEST	Return `errAECoercionFail`, set `kOSAErrorOffendingObject` to the given value, set `kOSAErrorExpectedType` to `cNumber` `→ error: Couldn't make {1, 2, 3} into a number.`

The user canceling an operation is considered an error; use `userCanceledErr` (-128) when this happens.

No interaction required.

Scripting commands should not require any user interaction, since part of the point of scripting is to create completely automated functions. It is acceptable to interact with the user if information was omitted from the command, such as attempting to `save` a never-before-saved file without specifying a location, but there should always be parameters available so a script can work without any interaction. Commands should never ask for confirmation. For example, the `empty trash` script command in the Finder does not ask for confirmation, even though the menu command does.

Before attempting to interact with the user in response to a scripting command, call `AEInteractWithUser`. If it returns an error, do not interact – either do without the user interface, or fail. This lets scripts specify whether or not they want to allow interaction.

Control the number of parameters.

Sometimes you may find yourself implementing a command that contains lots of options, for which you might be tempted to make separate Boolean parameters. When the number of parameters is small, it looks good to be able to say with a, b, and c. Excessive use of this technique, however, can lead to unwieldy dictionary entries for these events with long lists of

parameters.

There are two solutions to this: either make a parameter or parameters that accept a list of enumerators for the option or set of options, or break the command into separate commands with more focused functionality, reducing the number of options for each one.

For example, suppose a statistics package creates a single command to perform any type of analysis with lots of parameters, like this:

> **analyze** data set -- 25 Boolean options

It would be better to split the analysis capability into multiple commands, each with a small group of Boolean parameters, such as

> **cluster** data set
> **correlate** data set
> **fit curve** data set

and so on. If you find yourself with parameters that can never appear together, it's a sign you may need to break up the command.

Extra information is not an error.

If you receive a command with extra parameters you were not expecting, just ignore them and do not return an error. Similarly, do not bother checking the `keyMissedKeywordAttr` event attribute – it serves no useful purpose any more, and always returns false in Mac OS X 10.2 and later.

Back to Top

Standard Commands

The Standard Suite defines 15 commands. Applications should use these in preference to creating their own. Not all the commands are meaningful to all applications. Irrelevant ones (for example, `open` in an application without documents, like System Preferences) should not appear in your dictionary.

In most cases, parameters of type specifier can specify any number of objects. Be sure your commands function correctly with singular, range, and `every` specifiers, and that they follow the ranges return lists rule described above.

A few words about location parameters

Several commands take parameters of type location specifier. This may be an insertion point specifier or a container object specifier, and may even be a range in most cases.

An insertion point is a location where an object may be inserted: `beginning`, `end`, or `before` or `after` another specifier. The other specifier may be a range or `every` specifier, in which case the operation is repeated for all the resulting insertion points.

```
add photo "On the beach" to end of album "Vacation"
```

```
move paragraph 1 to after paragraph 3
make new word at beginning of every paragraph with data "Note:"
```

A container object specifier is a container in which the objects are to be added, moved, made, or whatever. The application chooses a reasonable location within the container. Again, the specifier may be a range or `every` specifier.

```
add photo "On the beach" to album "Vacation"
duplicate message 1 to every mailbox whose name contains "backup"
```

Some containers may accept only container specifiers; see Object Specifiers for details.

add

add specifier [**to** location specifier]
-- Returns a specifier to the object in its new location.

Adds the specified objects to a (position within a) container, and returns their new locations. If the `to` parameter is omitted, it is inferred from the target. (For example, in `tell album "Waterfowl" to add photo 4`, the `to` location is the target, `album "Waterfowl"`.) Do not support this command if your application has only one-to-one and one-to-many relationships; see Many-to-many relationships for more details. Use of a range or `every` specifier in the `to` parameter assumes that an object may be added to any number of containers; this may not be true for all applications.

close

close specifier [**saving** yes | no | ask]

Closes the specified objects. Documents and windows should be closeable if they exist; applications may apply `close` to other objects as well, and may add additional parameters.

count

count specifier
-- Returns the number of specified objects.

Returns the number of specified objects. This command is an exception to the ranges return lists rule; `count` with a range or `every` returns the number of objects specified, which may be zero for `every` specifiers. Attempting to `count` non-existent objects is an error. Using `count` with a singular specifier such as by-name is valid but not very useful, since it will either return 1 or fail if there is no such object. (Scripts should use `exists` in such

cases.) For historical reasons, `count` used to define an `each` parameter, but it was not necessary and has been deprecated.

delete

delete specifier

Removes the specified objects from any containers they may be in and destroys them. `delete` should always delete the object, no questions asked. If you want to have a trash of some sort (such as the Finder does), define a `trash` object and let users `move` objects to it. Depending on the application model, deleting an object may cause its container to be deleted as well – it may not be possible for a container to have no elements.

duplicate

duplicate specifier
  [**to** location specifier]
  [**with properties** record]
-- Returns a specifier for the new object.

Duplicates the specified objects to the specified locations, optionally changing some of their properties using the `with properties` parameter, and returns specifiers for the new objects. If the `to` location is not specified, create the new objects in the same container, suitably altered. (For example, `duplicate thing "Bob"` might result in a new thing named "Bob copy".) It is not specified what happens if the new object would create a conflict; an application can simply fail, or it can resolve the conflict by tweaking the new object, or it can define additional parameters to specify how to handle conflicts.

exists

exists specifier
-- Returns true if the object exists, false if not.

Returns true if all of the specified objects exist, or false if they do not. Unlike most commands, specifying non-existent objects is not an error; it merely means the result is false. Like `count`, `exists` is an exception to the ranges return lists rule; range specifiers return true if and only if all the objects in the range exist, `every` specifiers return true if at least one such object exists.

get

get specifier [**as** class]
-- Returns the value of the specified object.

Returns the value of the specified objects. You do not need to define `get` in your dictionary, since it is built into AppleScript. You may not redefine `get` with different parameters.

make

make [**new**] class
  [**at** location specifier]
  [**with properties** record]
  [**with data** anything]
-- Returns a specifier to the new object.

Makes a new object and returns a specifier to it. `at` should be optional. If it is omitted, pick a reasonable default location – the beginning or end of the current document are common choices. Ideally, the `with` parameters are optional, but one or the other (but not both) may be required for some classes. Be careful to distinguish between `with properties` and `with data`. Use `with properties` when you are overriding default properties on an application-created object. When the object is effectively defined by data from an outside source – say, a string or a file – use `with data`, even if one of the created object's properties is (a pointer to) that data.

move

move specifier **to** location specifier
-- Returns a specifier to the object in its new location.

Moves the specified objects to a new location and returns specifiers for them in their new locations. The `to` parameter is always required, and may not be a range or `every` specifier.

open

open file | list of files
-- Returns a specifier to the created application object.

Opens the specified files, and returns the new application objects for them – probably `documents`. The standard version is designed for opening document files, but applications may apply `open` to other objects and add additional parameters.

print

print specifier
[**with properties** print settings]
[**print dialog** boolean]

Prints the specified objects. For most applications, the objects are always documents, but this is not required. Support for the `with properties` and `print dialog` parameters was added in Mac OS X 10.3 (Panther); see TN2082 for details.

quit

quit [**saving** yes | no | ask]

Quits the application. The value of the `saving` parameter, which defaults to `ask`, is passed to any resulting `close` commands. Applications that automatically save everything should not define the `saving` parameter.

remove

remove specifier [**from** specifier]

Removes the specified objects from their container. If the `from` parameter is omitted, it is inferred from the direct parameter if possible. (For example, in `remove first photo of album 1`, the `from` parameter is understood to be `album 1`. However, in `remove photo id 437`, it is impossible to tell, and the application should return an error.) Do not support this command if your application has only one-to-one and one-to-many relationships. `remove` may or may not delete the object, depending on the application model; see Many-to-many relationships for details. Depending on the application model, removing an object may cause its container to be deleted – it may not be possible for a container to have no elements.

save

save specifier [**in** file]

Saves the specified objects, possibly in a specified file. `save` without `in` acts like the Save menu command, and may ask the user for a file location; `save in` with an unsaved file acts like Save As; `save in` with an already-saved file acts like Save A Copy.

select

select specifier

Selects the specified objects in the user interface. See Scripting the Selection for details on behavior.

set

```
set specifier to value
-- Returns the value assigned.
```

Sets the specified objects to a value and returns the value assigned, that is, the value of the completely evaluated `to` parameter. This is not necessarily the same thing as the original value of the parameter, in particular if it was an object specifier. In most applications, the direct parameter must be a property specifier, since they do not allow setting of actual objects to values. You do not need to define `set` in your dictionary, since it is built into AppleScript. You may not redefine `set` with different parameters.

Back to Top

Scripting the View

After all this talk about scripting the model, not the view, this section probably comes as a bit of a surprise. Nonetheless, a small amount of view-oriented scriptability is valuable. One of things users do with scripting is to add new commands to their applications, and in order to do that effectively, they need to be able to find out what is visible and what is selected.

Scripting Windows

As mentioned in Objects, windows should be indexed front to back, so `window 1` or `first window` is always the frontmost window, and `window -1` or `last window` is always the rearmost. This is implicit in the AppleScript language, because it allows `front` and `back` as synonyms for `first` and `last`, respectively.

If you have more than one kind of window – say, document windows and palette windows – create different `window` subclasses to handle them. Given a window specifier, a script should be able to determine what kind of window it is by inspecting the `class` property.

Windows that display model objects should have appropriately named properties to get the object (or objects) that they display. The most common case of this is document windows, which should have a `document` property that points to the document for that window. If there is exactly one model object for a window, the property should be the implicit subcontainer for the window. (In sdef terms, define it using a `contents` element.) That way, a script can ask for model elements of a window without having to explicitly go through the model object property. For example, if `documents` have `words`, then `get word 1 of window 1` should work. Without the implicit subcontainer, a script would have to say `word 1 of document of window 1`. The reverse relationship is not recommended; for instance, documents do not have windows as implicit subcontainers.

The activate command

To activate a window – that is, to bring it to the front – use the `activate` command and specify a window. Calling `activate` with no direct object should bring the entire application forward. `activate` should also work with objects that are one-to-one mapped with windows, such as documents. The system will automatically handle the activate-whole-application case for you, but applications must handle the other cases themselves.

Back to Top ▲

Scripting the Selection

The selection, in scripting terms, refers only to selected model objects. Do not attempt to represent the selection in dialog boxes. Not all applications need to support scripting the selection – for some very simple ones, such as control panel-like applications, it may not be meaningful.

There are two ways to work with the selection: the `select` command, and `selection` properties. To find out what is selected, get the value of the `selection` property. To select something, either `select` it, or set the appropriate `selection` property.

The select command

`select`, given an object specifier, will select that object in the user interface and make it active. This means

- bringing the containing window to the front.
- changing input focus to the appropriate part of the window.
- scrolling the view so that the object is visible.

Whenever you `select` something, the appropriate `selection` properties should change to that same something.

The selection property

The `selection` property is the current selection for that object. There may be more than one `selection` property in an application:

- The application class should always have a `selection` property. Setting it behaves as the `select` command.
- If you have one window per document and each document maintains its own selection (as, for example, TextEdit), then your document class should have a `selection` property. Setting it changes the selection for that document, but does not change focus, so a scripter can change the selection of an inactive window without making it active.
- If you can have multiple windows viewing the same model objects (as, for example, Mail can), then windows should have a `selection` property. Setting it changes the selection for that window, as the previous case. If selection is purely a view concept,

then the document class (if any) should not have a `selection` property.

Other objects that have their own selection should also have a `selection` property.

As much as possible, `selection` properties should always be the same type of value: always a single object, always a list of objects, or whatever is appropriate. In an application that has several different kinds of selectable data, the application's `selection` property will often change type; this is acceptable. For example, in Mail, the selection could be a list of messages, a list of addresses, or text.

Selecting Text

In a text-based object model, the value of the selection is a specifier that gives the location of the selected text, not the actual text – for example, `characters 1 through 3 of document 1`, not `"The"`. This allows scripts to change both what is selected and the selected text (see below), and allows them to speak of positions relative to the selection. To get the selected text, get the `contents of the selection`.

If the selection is a range of characters, then you should return a range of some sort, such as `characters 1 through 3`. (However, see the following paragraph.) If the selection is just an insertion point and contains no characters, then use the `insertion point` class, as shown in the table below.

Some applications are aware enough of the text object model to report the selection in terms of units larger than characters. For example, if the user selected the third paragraph, the `selection` property would report `paragraph 3`, not `characters 173 through 254`; selecting the second word of that paragraph would yield `word 2 of paragraph 3`. This behavior is encouraged, but not required.

If you support disjoint text selections, then the `selection` property should be a single specifier if the selection is not disjoint, but should be a list of specifiers if it is. This breaks the always be the same type of value guideline, but yields less surprising behavior in most cases.

Table 10 : Selecting Text.

Now is the time	`insertion point before character 1`
Now is the time	`insertion point after character 3` `insertion point after word 1`
Now is the time	`characters 1 through 3` `word 1`
Now is the time	`{characters 1 through 3, characters 12 through 15}` `{word 1, word 4}`

(In real life, the various specifiers would continue on with something like of document 1. This has been removed for brevity.)

Writing to the selection

Since writing to the selection property itself merely changes what is selected, you must write to a property or element of the selection to change the actual text. As mentioned above, `selection` should allow access to sub-elements – in this case, `characters`, `words`, and so on. To get at the whole selection, use a `contents` property. Setting a property or element changes the text, and may change the length of the selection. For example:

Table 11 : Writing to the selection.

`select word 3`	one two four
`set contents of the selection to "thref"`	one two thref
`set last character of the selection to "e"`	one two three

Moving the selection

The text model allows moving the selection relative to itself by use of `before` and `after` specifiers using the selection as a base. To select a point, use the `insertion point` class. Use of `beginning` and `end` is also allowed.

Table 12 : Moving the selection

`select word 1`	Now is the time
`select word after the selection`	Now is the time
`select insertion point after the selection`	Now is the time
`select beginning`	Now is the time
`select end`	Now is the time

As stated above, using `set the selection to` would have had the same effect.

Unusual Concerns

Multiple selections in one window

Some applications have multiple views in one window, each with its own selection. For example, viewer windows in Mail can have as many as three selections: the selected mailboxes, the selected messages within that mailbox, and the selected text within a message. Only one of them should have focus, and this determines the value of the `selection` property for the application and the window as a whole. (If you have trouble determining where the focus is, think of it as if I select Cut or Copy, what data will go onto the clipboard?) In such cases, you should add additional selected `thing` elements to get at the various sub-selections. For example, Mail would define the element types `selected mailbox`, `selected message`, and `selected text`.

Back to Top

General Tips

Be careful when reusing codes.

If you use the same term in more than one place in your dictionary, all of them must use the same 4-byte code. For example, if you use `input` as a parameter, again as a property, and later as an enumerator, use the same type code in all three places. Conversely, if you use the same type code in more than one place, all of them must use the same term. This also applies to terms and codes inherited from system- or framework-supplied dictionaries. For example, if you define a `color` property for a class, its code must be `'colr'`, because that is how it is defined in the AppleScript dictionary and Cocoa's NSCoreSuite script suite. Failure to follow this rule will cause your scripts to fail or change in odd ways when they are compiled or decompiled – AppleScript will change the original term to be the last one defined.

`aete` dictionaries actually exploit breaking this rule in order to define synonymous terms or codes. sdef provides direct support for synonyms with its synonym elements; see the sdef documentation for details.

Don't localize your terms

For one thing, basic AppleScript syntax is oriented towards English; having identifiers in another language would be odd. For another, it probably would not work: AppleScript identifiers cannot contain non-ASCII characters, which severely restricts what you can say in most languages.

Back to Top

Tips for Using Cocoa Scripting

Cocoa is Apple's application framework: it is strongly object-oriented, and makes it easy to add scriptability. For documentation, see Scriptable Applications. However, Cocoa's default scripting support does not follow the scripting interface guidelines in all cases. Fortunately, there are workarounds:

For Everyone

Add an 'aete' to your application.

Cocoa applications currently define their scriptability by providing a set of script suites. Each script suite is a pair of plist files, a suite definition (.scriptSuite) and a suite terminology (.scriptTerminology). Cocoa can generate an `aete` format dictionary from these if you let it, but the result tends to be sub-standard, because script suites cannot express certain concepts. (Also, your application will have to be launched to get the dictionary, which tends to annoy scripters.)

Therefore, you should build a parallel `aete` dictionary as part of your application. The easiest way to do this is to write your dictionary using an sdef, and then generate all three files (.scriptSuite, .scriptTerminology, and aete.r) from it using `sdp`(1).

Do not require the 'at' parameter for 'make'.

Because of an early design mistake, Cocoa requires the `at` parameter for `make` by default. You should always override this using the `LocationRequiredToCreate` attribute – see the Cocoa Scripting release notes for more details. If you write your dictionary using an sdef, `sdp`(1) will add the correct attribute for you.

For The Ambitious

Make specific definitions instead of using the generic ones.

Cocoa Scripting suite definitions have trouble being completely accurate. Definitions you inherit from the frameworks – in particular, the Standard and Text Suites – are almost completely generic. Commands that your application does not support are still visible, direct parameters are defined as merely type reference, and `application` has `document` elements whether it needs them or not. You can work around this by creating an `aete` dictionary for your application (see above), but it is more work. If you use an sdef to generate your `aete`, you will have to modify the sdef version of the Standard Suite (`/Developer/Examples/Scripting Definitions/NSCoreSuite.sdef`), since it currently will have the same problem.

Provide good error messages.

Cocoa Scripting's default error messages are written in generic programmerese (such as NSReceiverEvaluationScriptError: 4) rather than specific English (such as Can't get window 17). As of Mac OS X 10.3 (Panther), you can set your own error numbers and messages by calling `[NSScriptCommand currentCommand]` and then calling `-setScriptErrorNumber:` and `-setScriptErrorString:` on the returned object. See the NSScriptCommand documentation for further details.

Back to Top

References

- AppleScript developer page
- AppleScript on Mac OS X
- AppleScript documentation
- Scriptable Applications

Back to Top

Document Revision History

Date	Notes
2004-03-01	Directions for providing a clean and consistent scripting interface for your application.