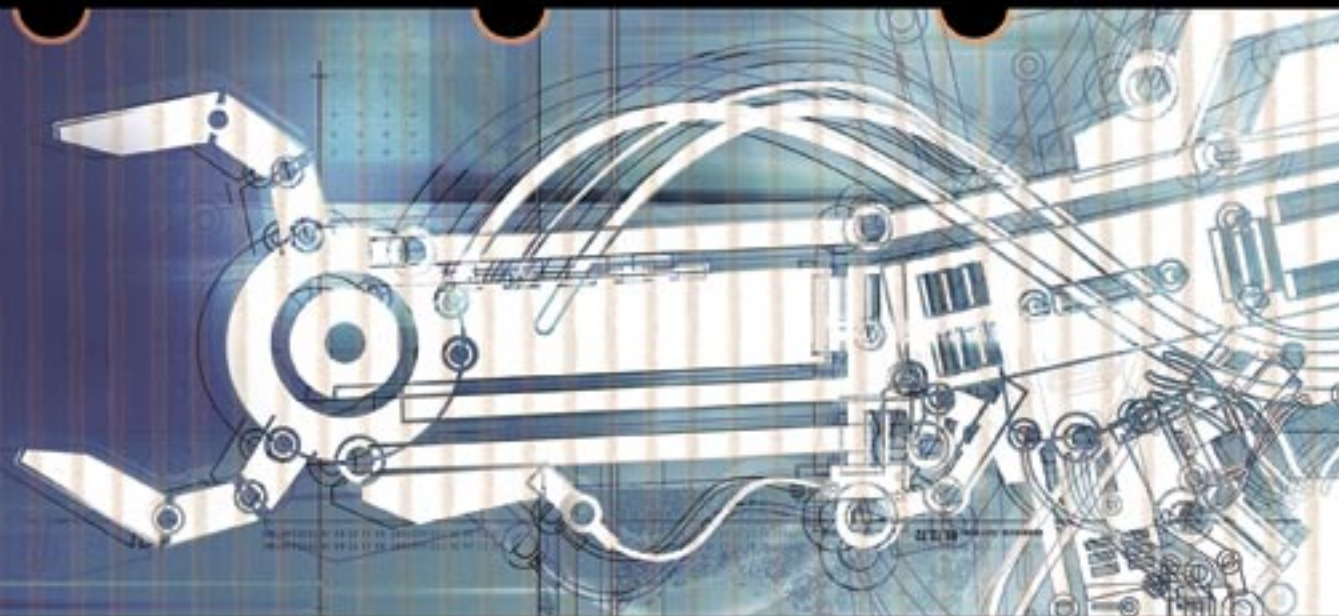


MASTERING MAC OS X TIGER



Mac OS X Technology Guide **to Automator**

BEN WALDIE

Mac OS X Technology Guide to **Automator**

B E N W A L D I E



SpiderWorks

For more great books, order online at

<http://www.spiderworks.com>



Table of Contents

How to Use this eBook	6	Chapter 5: Utilizing a Workflow	43
About the Author	7	Saving Workflows	43
Acknowledgements	8	Opening a Workflow	55
Chapter 1: Introduction	9	Printing a Workflow	56
Goals of this Book	10	Running a Workflow	57
What You Need to Get Started	11	Troubleshooting	59
Example Code	12	Importing Actions	62
Section 1: Using Automator	13	Importing a Workflow	63
Chapter 2: Automator Overview	14	Creating a Workflow from Finder Items	64
Benefits of Automator	14	Chapter 6: Building an	
How Does Automator Work?	15	Example Workflow	66
Chapter 3: Automator's Interface	20	Workflow Example 1 –	
Navigating Automator's Interface	20	Backup Safari Data	66
Chapter 4: Constructing a Workflow	30	Workflow Example 2 –	
Creating a New Workflow	30	Email Photo Contact Sheet	73
Adding Actions to a Workflow	31	Chapter 7: Advanced Topics	81
Configuring Action Settings	32	Working with Spotlight	81
Working with Input and Output Values	35	Triggering UNIX commands	83
Collapsing Actions in a Workflow	39	Working with AppleScript	83
Deleting Actions from a Workflow	39	Developer-Related Actions	85
Disabling Actions in a Workflow	41	Providing Feedback	85
Moving an Action in a Workflow	42	Section 2: Developing for Automator	87
		Chapter 8: Introduction to	
		Developing for Automator	88
		Related Technologies Overview	88
		Types of Automator Actions	97
		What You Need to Get Started	97



Table of Contents

Chapter 9: How Actions Work	98	Chapter 13: Constructing an Action's Interface	141
What is an Action?	98	Preparing for Automator Action Interface Development	141
Threading	102	Building an Action's Interface	143
Where Actions are Stored	103	Interface Design Guidelines	145
Chapter 10: Planning an Action	105	Grouping Interface Elements	148
Action Functionality	106	Chapter 14: Retrieving an Action's Settings	151
Action Input and Output	107	Establishing Interface Element Bindings	151
Action Settings	107	Linking Parameters to the Action's Code	156
Action Naming	108	Chapter 15: Adding Code to an Action	158
Chapter 11: Building an Action Project	109	Action Processing Code Overview	158
Creating the Project	109	Adding Code to an AppleScript Action	159
Action Template Components	111	Adding Code to a Cocoa (Objective-C) Action	165
Chapter 12: Configuring an Action's Property List File	114	Triggering Code From Other Languages	171
Editing Properties	114	Conversion Actions	172
Configuring General Action Properties	119		
Configuring an Action's Icon	121		
Configuring an Action's Description	124		
Configuring Action Input and Output Values	128		
Adjusting Action Behavior	132		
Specifying Required Resources	133		
Configuring a Warning	135		
Localized Property List Strings	137		
Example info.plist File	139		



Table of Contents

Chapter 16: Testing and Debugging an Action	174
Building and Running from within Xcode	174
Building, Installing, and Testing	176
Building and Debugging	177
Tips for Testing Actions	178
Common Problems and Possible Solutions	179
Chapter 17: Pulling it Together	181
Log Activity AppleScript-Based Action	181
Adjust Image Color Cocoa Objective-C-Based Action	190
Chapter 18: In Conclusion	201
General Automator Resources	201
Developer Resources	203
In Closing	206
Appendix A: Automator Action Development Step-By-Step	207
Appendix B: Automator Input and Output Uniform Type Identifiers (UTIs)	209
License Agreement	211
Index	212



How to Use this eBook

On-Screen Viewing

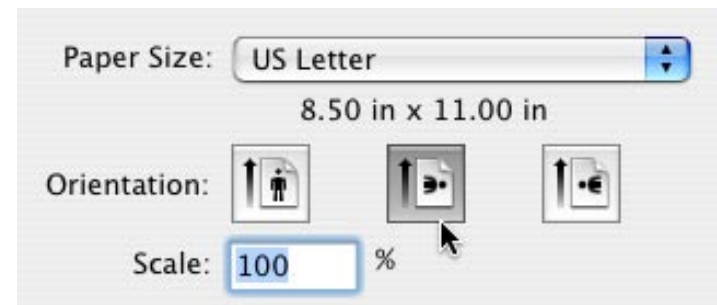
We recommend using Adobe Acrobat or the [free Adobe Reader](#) to view this eBook. Apple Preview and other third-party PDF viewers may also work, but many of them do not support the latest PDF features. For best results, use Adobe Acrobat/Reader.

To jump directly to a specific page, click on a topic from either the Table of Contents on the first page or from the PDF Bookmarks. In Adobe Reader, the PDF Bookmarks can be accessed by clicking on the Bookmarks tab on the left side of the screen. In Apple Preview, the PDF Bookmarks are located in a drawer (Command-T to open).

If your mouse cursor turns into a hand icon when hovering over some text, that indicates the text is a hyperlink. Table of Contents links jump to a specific page within the ebook when clicked. Text links that begin with “http” or “ftp” will attempt to access an external web site or FTP server when clicked (requires an Internet connection).

Printing

Since SpiderWorks eBooks utilize a unique horizontal page layout for optimal on-screen viewing, you should choose the “Landscape” setting (in Page Setup) to print pages sideways on standard 8.5” x 11” paper. If the Orientation option does not label the choices as “Portrait” and “Landscape”, then choose the visual icon of the letter “A” or person’s head printed sideways on the page (see example below).





About the Author

The Author

Ben Waldie is president and CEO of Automated Workflows, LLC, a company offering AppleScript solutions and workflow automation consulting to businesses, including Adobe Systems, Apple Computer, NASA, PC World, and TV Guide Magazine. Ben has presented at Macworld, Seybold, and other popular events, has taught custom AppleScript training classes, and authored an introductory AppleScript training CD for The Virtual Training Company. Ben writes a regular AppleScript column for *MacTech Magazine*, is a contributing editor on MacScripser.net, and is president of The Philadelphia Area AppleScript Users Group.

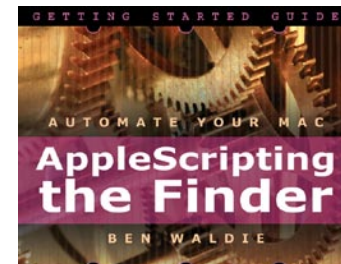


Visit Automated Workflows online:
<http://www.automatedworkflows.com/>

Automator Actions by Automated Workflows:
<http://automatedworkflows.com/automator/actions.html>

Also Available from SpiderWorks

Don't miss Ben Waldie's first book, *AppleScripting the Finder*, now available from SpiderWorks. The Finder is the heart of the Mac OS. When driven by AppleScripts, the Finder can be a powerful desktop assistant, replacing tedious tasks with a single mouse click or keystroke. Put your Mac to work with Ben Waldie's expert, time-saving AppleScript techniques that will streamline your daily workflow.



Download the free preview and order online at
<http://www.spiderworks.com/>

Publisher Credits

Cover Design: **Mark Dame** and **Dave Wooldridge**
Cover Illustration: **Dennis Glorie** (iStockphoto.com)
Interior Page Design: **Robin Williams**
PDF Production: **Dave Wooldridge**



Acknowledgements

I would like to take this opportunity to sincerely thank a number of people who have directly led to, and assisted with, the creation of this book. First off, I would like to thank my wife, Jenifer, who spent many a late night reading, proofing, and offering input. Without her, many of the things that I set out to accomplish would not be possible. I would also like to thank Tim Davis, who gave me the opportunity to learn about AppleScript and development on the Mac so very many years ago and was kind enough to let me run with it. Thanks to Sal Soghoian, who offered very helpful input, suggestions, and information about Automator, and who, along with the incredibly talented AppleScript team at Apple, continues to keep AppleScript alive, well, and growing.

Thanks to Mark Dalrymple, and his extensive knowledge of Objective-C, for providing valuable input regarding the Objective-C-related sections of this book. Finally, I would like to thank Dave Mark and Dave Wooldridge for giving me the opportunity to write this book, and for the incredible things they do to make this book and so many other great books possible.

– Ben Waldie



Chapter 1 Introduction

Automation has always been an important part of personal computing. Since it was first created, the overall purpose of the computer has been to make life easier by performing complex and repetitive tasks in an efficient and reliable manner.

However, does your computer really make your life easier? Or, does it seem to make more work for you to do? For many, the latter seems to be the case, but it doesn't need to be. Your computer's ability to make your life easier is directly related to how you are actually using your computer.

Everything you do on your computer involves software. Software controls the physical hardware of your computer, while providing you with the tools you need to accomplish specific tasks, such as page layout, image editing, and word processing.

If you're like me, then your workflow probably involves numerous software applications, and you probably find yourself doing many of the same things over and over again. Take a look at digital photography, for example. If you have a digital camera, each time you plug it into your computer, you need to download the images into

a folder structure, import them into a photo catalog, rename the images, etc. Applications like iPhoto can help tremendously with these tasks by automating the process for you. If you look closely at your own workflow, you will probably see a number of tasks that are being automated by your software.

Some of the time, your software will do everything that you need it to do, and in an efficient manner. However, this isn't always the case. In some cases, your needs may be quite unique, and you may require customized features or tools that don't yet exist within your software. In other cases, your software may work perfectly, but it doesn't provide a way to automate those time-consuming tasks that you perform on a regular basis, or it may not provide a way to move your data efficiently between multiple applications.

If you are a programmer, then you may be able to write your own custom software, or you may be able to automate your existing software using something like AppleScript. But, what if you're not a programmer? With the release of Mac OS X 10.4, your prayers have been answered. Installed with the Mac OS is



Chapter 1:
Introduction

Automator, a tool for the average user with one specific function – to automate time consuming and repetitive tasks for *your* specific workflow.

Introduction to Automator

As we will discuss throughout this book, Automator will allow you to define how repetitive and time-consuming tasks are automated on your Mac, with no programming required. With a user-friendly interface, Automator will allow any user to construct a customized automated workflow, simply by dragging and dropping icons around, and by specifying a few options.

Within the Automator application, users work with two main types of components, **actions** and **workflows**. Actions are built by developers, and each action's goal is to perform a single task, such as opening a file, checking your email, or rotating an image. Workflows are designed and constructed by users, by piecing together actions in order to create a virtual assembly line of automated tasks. Once constructed, workflows may be saved and triggered by a user in a variety of ways.

Programmers are in luck too. Mac OS X includes all of the tools needed to build your own custom actions, which can be plugged right into the Automator application, extending the possibilities of user-defined automation even further.

Goals of this Book

Throughout this book, we will take an in-depth look at Automator from the perspective of both a user and a developer. The book will be broken into two main sections.

In the first section, geared toward users, we will explore the Automator application itself. We will discuss the features and benefits of Automator, and we will walk through using it in order to construct and execute automated workflows that meet your specific needs.

The second section of the book will focus on how you can develop your own Automator actions. We will explore the tools used to build Automator actions, take a look at example code, and discuss resources for continued learning.

By the end of the users section of this book, you should have a good understanding of the Automator application, and you should feel comfortable using it. If you are a developer, then the developer section of this book should provide you with the confidence and knowledge that you will need to begin building your own actions, whether for personal use, or for commercial or freeware distribution.



Chapter 1:
Introduction

What You Need to Get Started

For the first section of this book, you won't need much to get started, other than a basic familiarity with Mac OS X and a copy of Mac OS X 10.4 or higher. Since Automator interacts with existing applications on your machine, you should also be comfortable using at least the core applications that are installed with Mac OS X, such as iTunes, iPhoto, Mail, and Address Book.

The second section of this book focuses on development and assumes that you have at least some prior experience developing software for the Mac OS X platform. You should have some familiarity with Xcode and Interface Builder, Apple's free software development tools. You should also have a basic knowledge of AppleScript and Objective-C.

Brief introductions to both of these languages will be provided at the beginning of the developer section of the book. However, you may wish to consult more in-depth resources as well. *Danny Goodman's AppleScript Handbook* serves as an excellent training guide for any developer interested in learning AppleScript. For an introduction to programming, check out Dave Mark's excellent *Learn C on the Macintosh* (there's a brand new, Mac OS X edition). The sequel to that book is Mark Dalrymple and Scott Knaster's *Learn Objective-C, Mac OS X Edition*.

All of these titles are available from SpiderWorks, <http://www.spiderworks.com>.

Software Utilized in this Book

The following specific software versions were used during the creation of this book:

- ▶ Mac OS X version 10.4
- ▶ Automator version 1.0
- ▶ Interface Builder version 2.5 (*for developers*)
- ▶ Xcode version 2.0 (*for developers*)

If you are using older versions of this software, please be aware that some or all of the functionality discussed may not be available to you. If you are using newer versions of this software, then you should be aware that certain aspects of the software might differ slightly between versions. Regardless of the software versions you are using, it is always recommended to conduct testing of any sample code prior to usage, as minor adjustments may be required, due to changes in terminology between versions.

In addition to the specific software listed above, the examples in this book will make use of various Apple applications in order to construct Automator workflows. If your system software is up to date, then you probably won't need to be concerned about the specific versions of these applications, although there could possibly be differences in behavior between versions.



Chapter 1:
Introduction

Example Code

All of the examples and sample code included in this book can be found in a separate download, named *AutomatorExamples.sit*. This download can be found in the SpiderWorks Customer Download Center at <http://www.spiderworks.com/extras/>. To login, you will need the special username and password that was listed in your SpiderWorks Login Access email (which you should receive within an hour after completing your online purchase of this book).

Once you have downloaded and decompressed *AutomatorExamples.sit* (using Stuffit Expander), you will see a directory named *Automator Examples*. Nested inside of this directory are sub-directories, named for the chapter to which the example files apply. Please note that not all chapters will have example files in the *Automator Examples* collection.

Move the *Automator Examples* directory to a convenient location on your hard disk, from which you can view and edit the files.

What's Next

The next chapter begins the first section of the book, and is geared primarily toward users. We will begin discussing the Automator application itself in greater detail, and discuss its features and benefits. We will also look at some examples of the types of workflows that can be made more efficient with the use of Automator. If you are a developer, you may still want to review the chapters in this section in order to become familiar with the Automator application.



Section 1

Using Automator



Chapter 2 Automator Overview

Automator's purpose is simple – to automate the time consuming, repetitive, manual tasks that plague our lives on a daily basis. Automator is easy to configure and highly customizable. With Automator, there's no need to wait for your favorite applications to implement macros or other scripting devices, and there is no need for custom programming. Automator gives you complete control over your workflow and puts *you* in the driver's seat.

Automator is installed in the *Applications* folder with Mac OS X, versions 10.4 and higher. It is easily distinguishable by its icon, which, appropriately enough, looks like a robot. See figure 2.1.



Figure 2.1 *The Automator Icon*

Benefits of Automator

As with any type of automation, there are a number of benefits that come from using Automator. A primary benefit of Automator is that its user-friendly interface makes automation simple for the average user. The key here is *average user*. With Automator, it doesn't take a rocket scientist to build an automated workflow. By putting such an easy-to-use tool directly into the hands of the average user, Apple ensures that virtually *everyone* is capable of implementing automation into their daily routines.

Automation can actually reduce stress. That's right, you read that correctly. Automation reduces stress. By removing the same old repetitive, boring tasks from your workload, you will feel a renewed sense of energy and motivation, as you will become free to focus on things that you actually enjoy, like graphic design, photography, and more.

Another benefit of automation is a reduction in user errors. No matter what we might like to think, we all do make mistakes from time to time. On the computer, these mistakes are often as simple as entering an



Chapter 2: **Automator Overview**

incorrect file name, accidentally deleting something, moving something into the wrong folder, etc. However, by automating manual tasks, you can eliminate the possibility of user errors occurring when those tasks are performed. A properly configured automated workflow will not make mistakes. It will perform consistently and accurately day in and day out, like a robot.

In addition, since an automated process can interact directly with the computer, it eliminates the time that would have been necessary during manual processing for things like moving and clicking the mouse, pressing keys on the keyboard, mulling over what to do next. The result is a much faster workflow, which can allow you to accomplish a lot more in the same amount of time, or possibly in even less time. Depending on how automation is implemented, it may even be possible for your processes to run unattended, allowing you to trigger it while you are at lunch, or when you leave your computer at the end of the day.

Increased accuracy, efficiency, and quality are all some of the benefits of automation. As you begin to use Automator, you will encounter these benefits first hand, and, once you do, you will wonder how you ever functioned without Automator.

How Does Automator Work?

Automator works by interacting with existing files, folders, and applications on your computer, or by interacting with the operating system itself. When using Automator, you first select the applications or types of processes that you want to automate, and then you specify the tasks that you want to perform within those applications or processes.

In Automator, each task is called an **action**, and you can link multiple actions together to form a **workflow**. The workflow you create may then be triggered from within Automator, or it may be saved and triggered outside of Automator in a number of ways. Once triggered, the actions within the workflow will process sequentially.

Automator Actions

Most Automator actions are designed to perform a single specific task, such as writing text to a file, opening a URL, or copying files from one location to another. Because of this, actions may be linked together sequentially to form a larger, multi-part, automated process.

When you begin planning an automated workflow, think of the tasks that you would normally need to perform manually. You might step through the sequence manually, and outline each step as you go. Think of each of these steps as a single Automator action. When you build your workflow in Automator, you will assemble these actions together to form a complete workflow.



Chapter 2: Automator Overview

Action Input/Output Values

In an Automator workflow, each action has the ability to pass information along to the next action in the sequence. Likewise, actions can also receive information from the previous action in the sequence.

For example, you may build a workflow that retrieves a list of files from a folder, compresses the files into an archive, and attaches the archive to a new email message. A workflow of this nature might consist of three separate Automator actions (see Figure 2.2).

- The first action retrieves a list of files within a specified folder, then passes that list on to the next action.

- The second action takes a list of files as input, then compresses the files into a single archive. The location of the compressed archive is output to the next action.
- The third and final action would tell Mail to create a new email message and attach the archive, which was received as input from the second action, to that message.

Information passed between actions can take on a variety of forms. Some actions may pass file and folder paths to the next action, some may pass text, some may pass email messages. The type of information input or output by an action will depend on the task that the

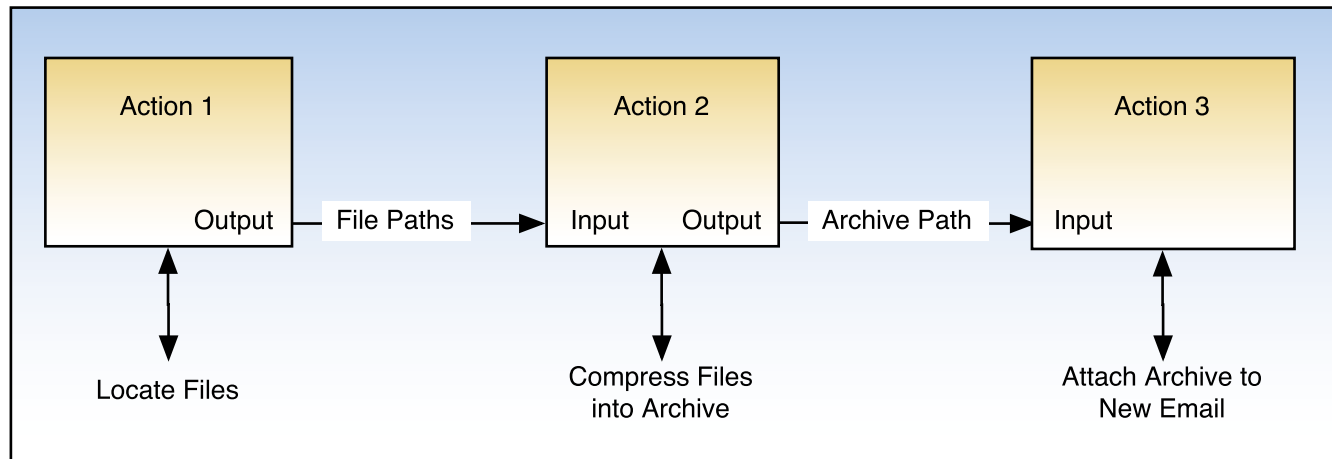


Figure 2.2 Automator Action Interaction



Chapter 2: **Automator Overview**

action is performing.

Since many actions are designed to process the information that is passed to them by the previous action, when you link actions together to form a workflow, you will need to make sure that each action passes the appropriate type of information to the next action in the sequence. For example, an action that will compress files into an archive will only be able to accept file paths from the previous action in the workflow sequence. Passing the wrong type of information to the action would produce an error. We will discuss this further when we begin building a workflow.

An action might take one type of information as input, and output a different type. For example, the *Download URLs* action, included with Automator, will accept a list of URLs from a previous action as its input, and output the file paths of the downloaded files to the next action in the sequence. Some actions may be configured to serve as *conversion* actions, whose purpose is only to convert information from one type to another, to be processed by the next action in the sequence. Other actions will perform a task, and then simply pass the output of the previous action through to the next action in the workflow sequence.

In some cases, actions may also be configured to ignore the output of the previous action in a workflow. This is sometimes necessary, as we will see when we begin constructing workflows.

Action Settings

As we will discuss in detail in a future chapter, when you build a workflow, you will also specify settings for many of the actions in a workflow. The settings that you specify will control how the action behaves when it is triggered in the workflow.

For example, using the scenario we discussed earlier for sending compressed files in an email, the compression action in the workflow would need to be configured to know where to save the compressed files. This action might be configured to save the compressed files to a specific folder, such as the Desktop, every time the workflow is run.

In some cases, you may not want to specify the settings for an action when you configure the workflow. Instead, you may want to give more control to the user, and allow the user to specify the settings when the workflow is triggered. Many actions offer this ability, and can be configured to display settings to the user during processing. For example, using the compression example again, you could configure the action to prompt the user to specify the output folder for the compressed files each time the action is triggered.

Some actions that will perform a single, very basic task may not require settings to be specified at all. For example, Automator includes an iTunes action for updating a connected iPod. This action does not require any settings to be specified. It will simply update any connected iPod, using iTunes.



Chapter 2:
**Automator
Overview**

Application Interaction

The applications with which Automator can interact depend entirely on the actions you have installed on your computer. Fortunately, Apple includes hundreds of actions to get you started. The actions that are installed with Mac OS X cover most of the core Apple applications, including Address Book, the Finder, iPhoto, iTunes, Mail, and Safari, among others. Apple also provides actions for interacting directly with the operating system and files on the system.

It's probably fairly safe to assume that you also use applications other than the ones Apple provides to you. If you do, then you're not out of luck either. Third-party Automator actions for a variety of applications are available from a number of developers. Since Automator is still new (at least at the time this book was written), the list of third-party actions may be small at the onset. However, the number of third-party actions is sure to grow quickly. For a comprehensive up to date list of third-party Automator actions, visit Apple's Mac OS X download website at <http://www.apple.com/downloads/macosx/>, or check out <http://www.automatoractions.com>.

Limitations of Automator

Like most applications, Automator does have some limitations, which are worth covering.

First, there is a possibility that, in some cases, actions cannot be developed for certain applications. If you encounter this situation, it is because the developer of the application has not provided the necessary “hooks,”

allowing developers to create actions that interact with that particular application.

If you learn of an application that does not support Automator action interaction, contact the application's developer to suggest that such support be implemented. A developer can implement Automator action support in two main ways. The developer can make the application AppleScriptable, which can allow for even greater automation possibilities, or the developer can implement a public API (Application Program Interface, a way for programmers to hook into a third-part application).

Another limitation of Automator is that, while it can be used to automate a large number of tasks, it cannot be used to create really complex automated processes. When you configure an Automator workflow to trigger specific actions, those actions will execute in the exact order that they have been configured, one right after the other. A workflow cannot include any logic to take a different course of action if a certain situation occurs during processing.

In addition, because actions in a workflow trigger in sequence, each action can only receive as input the output of the previous action. An action has no knowledge of the output values of other actions in the workflow. This can put some limitations on the ability to pass information around throughout the workflow.

Because of this, you probably wouldn't use Automator to do something extremely complex, such as building



Chapter 2:
**Automator
Overview**

a two thousand page catalog of products for your company in QuarkXPress, using data from FileMaker Pro. In order to achieve complex automation like this, you would probably want to use something like AppleScript instead. AppleScript powers much of the behind-the-scenes functionality of Automator, and is a more appropriate fit for more complex automation.

An AppleScript can be written to perform just about any task imaginable on the Mac, and can include all the logic your unique scenario could possibly require. Though relatively simple to learn in comparison to other languages, AppleScript does have a much larger learning curve than that of Automator.

To learn more about AppleScript, check out Danny Goodman's *AppleScript Handbook (Mac OS X Edition)*, <http://www.spiderworks.com/books/ashandbook.php>.

What's Next

Now that we have learned a little bit about how Automator works, it is time to get started actually using it. In the next chapter, we will explore various aspects of Automator's interface. Once you understand the interface, we can begin to implement some example workflows, helping you to take advantage of this awesome technology.



Chapter 3 Automator's Interface

Now that we have covered some of the how, where, and why of Automator, we can begin to explore the application's user interface. Automator's interface is relatively straightforward. If you've used other Apple applications, such as iTunes, iPhoto, or Mail, then many aspects of the interface will seem very familiar to you.

Navigating Automator's Interface

When you first launch Automator, you will be presented with a new window. This window represents a new workflow. See figure 3.1.

The workflow window consists of a toolbar, along with four main browser views. If desired, the views in the window may be condensed or expanded. You may also want to expand the window to take up most of your screen in order to reduce scrolling. If you do choose to modify the layout of the window, and you wish those changes to apply to future windows, you may select *Save as Default* from the *Window* menu in the menu bar. See figure 3.2.



Chapter 3: Automator's Interface

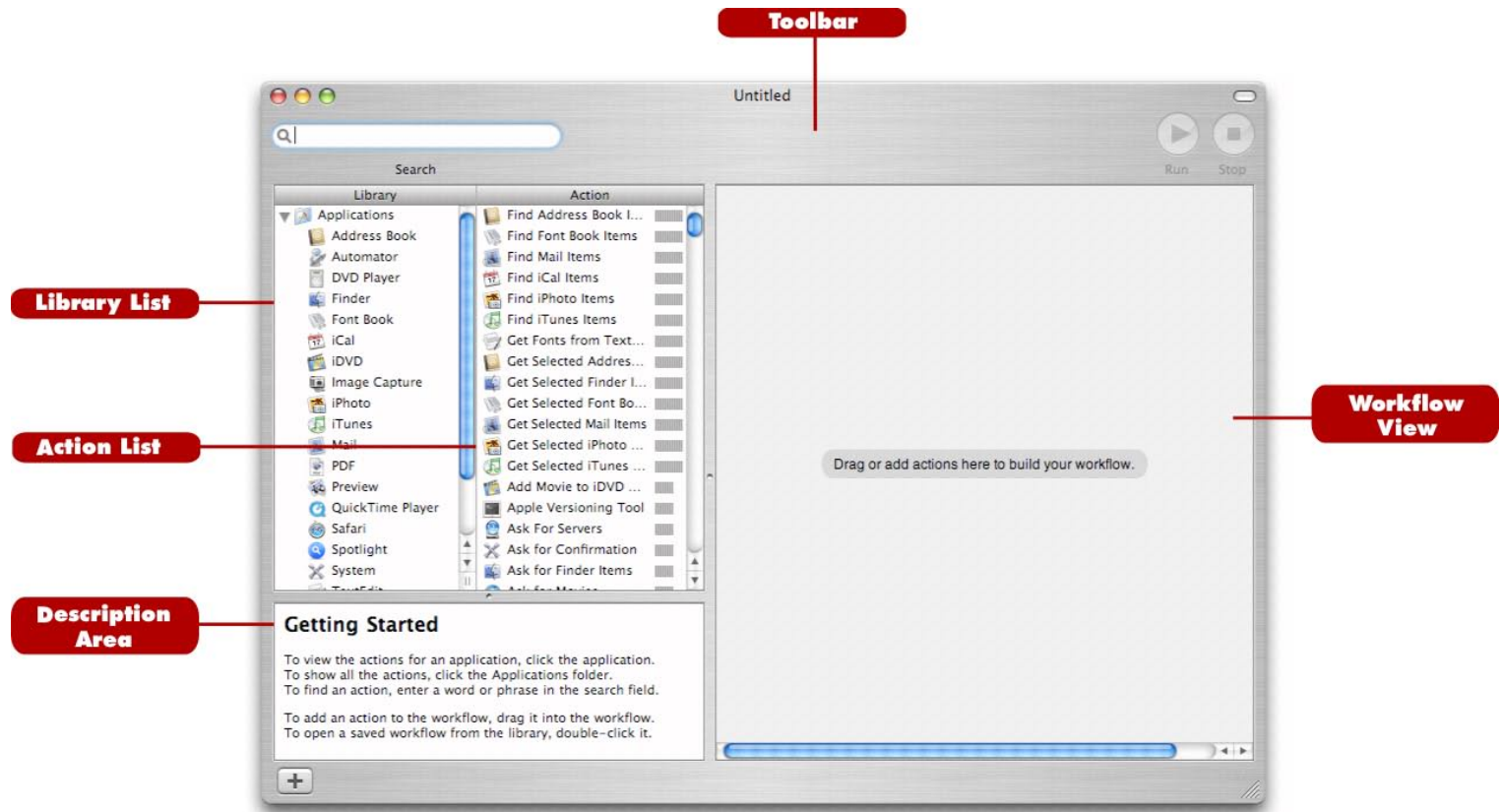


Figure 3.1 A New Workflow Window



Chapter 3: Automator's Interface

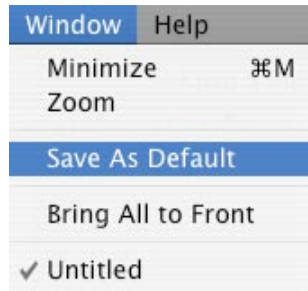


Figure 3.2 The Save as Default Menu Item

The Toolbar

By default, the toolbar in a workflow window will contain a search field, which may be used to locate actions matching a specified name or keyword. See figure 3.3 A. Since Automator contains hundreds of actions, not including any third-party actions you may have installed, this search field can help you to quickly narrow the lengthy list of actions down to only a select few.



Figure 3.3 The Workflow Window Toolbar

The workflow window's toolbar also contains a *Run* and a *Stop* button by default. See figure 3.3 B. These buttons are used to run and stop a workflow from

within Automator.

Like most Mac OS X applications, Automator allows you to customize the look and feel of its toolbar. To display the toolbar customization panel, select *Customize Toolbar* from the *View* menu. See figure 3.4.

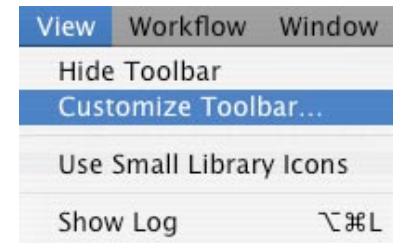


Figure 3.4 The Customize Toolbar Menu Item



Chapter 3: **Automator's Interface**

Once the customization panel has been displayed, you may rearrange the toolbar by dragging the icons around in the toolbar as desired, or by dragging and dropping new icons from the customization panel into the toolbar. To reset the default Automator toolbar, drag the default set from the customization panel into the window's toolbar. See figure 3.5.

When configuring the toolbar, you may optionally specify whether to display the toolbar's buttons with labels, without labels, or as a label without an icon at all. You can also tell Automator to display the text in the toolbar in a smaller font size. Click the *Done* button to close the customization panel.

menu. To display this contextual menu, hold down the *control* key and click in the toolbar (see figure 3.6).

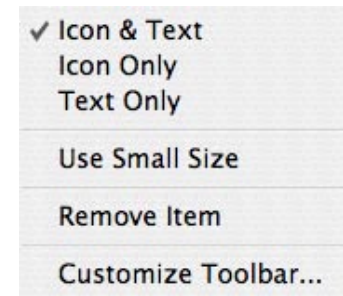


Figure 3.6 *The Toolbar Contextual Menu*

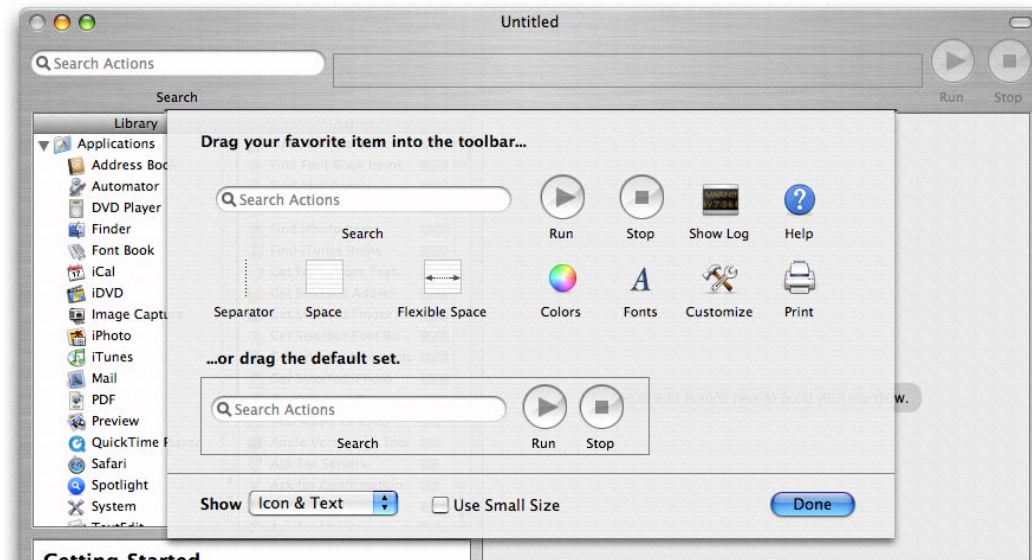


Figure 3.5 *The Toolbar Customization Panel*

You can also customize the toolbar via a contextual



Chapter 3: Automator's Interface

Any changes that you make when customizing the toolbar will automatically be applied to all future workflow windows.

The Library List

The *Library* list appears along the left side of the workflow window. See figure 3.7. The *Library* list is a list of Automator action categories and groups of saved workflows. At the top of the *Library* list, a group named *Applications* contains a variety of action categories. Click on a category, and the actions within that category will be displayed in the *Action* list, which is located to the right of the *Library* list. We will discuss the action list shortly. Within the *Library* list, a category typically represents an application or process that may be controlled by Automator. As you can see in Figure 3.7, the *Applications* group features categories for the *Address Book*, *Automator*, *DVD Player*, etc.

Note that not every category within the *Applications* group is actually an application. For example, the *Applications* group shown in Figure 3.7 lists *PDF* and *System* as categories. These categories are used to organize actions that are not tied to a specific application.



Figure 3.7 The Library List

Beneath the *Applications* group in the *Library* list are additional groups. These groups are used to organize your saved workflows. By default, Automator includes a group named *Example Workflows*. See figure 3.8 A. This group should already contain some example



Chapter 3: Automator's Interface

workflows to get you started. The *Library* list also includes a group named *My Workflows*, for storing your own workflows. See figure 3.8 B. You may optionally add additional groups (see figure 3.8 C) to the listing by clicking the + button at the bottom of the workflow window. See figure 3.8 D.



Figure 3.8 Adding a Custom Workflow Folder

If desired, you may adjust the size of the category icons in the *Library* list, by selecting or deselecting *Use Small Library Icons* from the *View* menu in the menu bar. See figure 3.9.

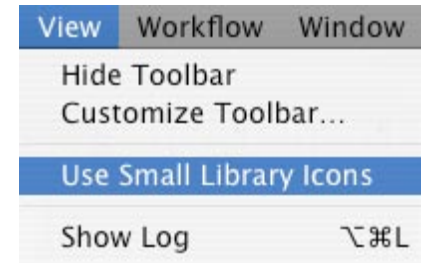


Figure 3.9 The Use Small Library Icons Menu Item

Action List

The *Action* list can be found immediately to the right of the *Library* list in the workflow window. See figure 3.10. If the *Applications* group has been selected in the *Library* list, then the *Action* list will display a list of all installed actions.



Chapter 3: **Automator's Interface**

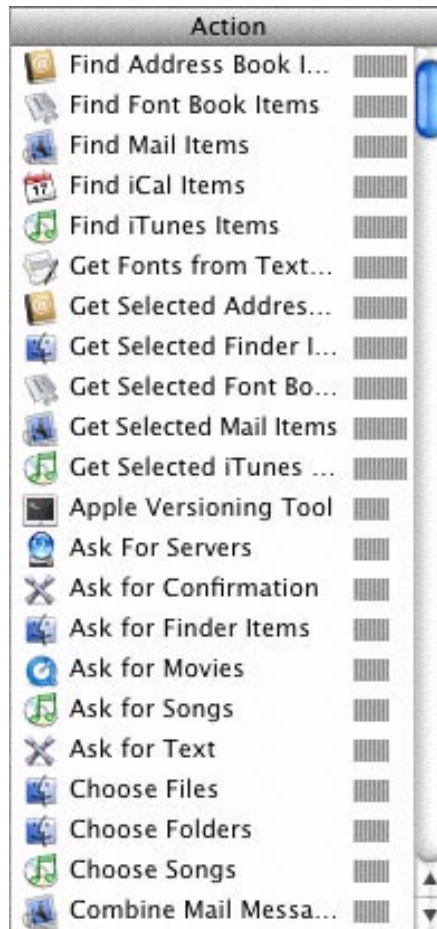


Figure 3.10 The Action List

To display a list of actions for a specific category, click on the category's icon in the *Library* list. See figure 3.11.

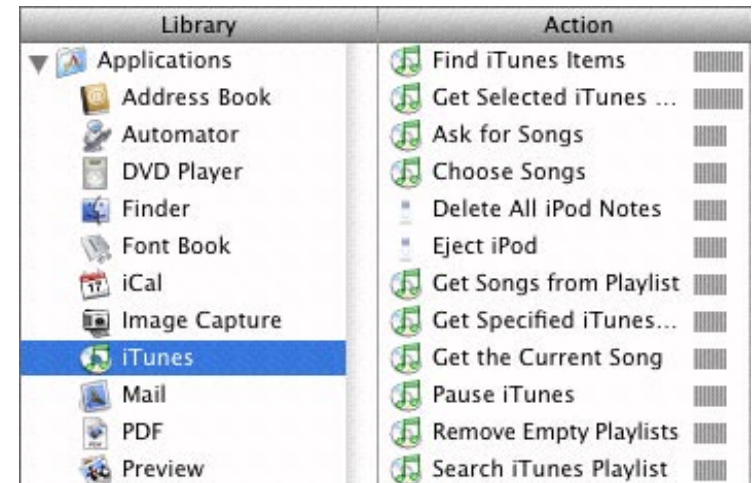


Figure 3.11 iTunes Actions

You may also narrow down the list of actions by entering a keyword into the search field in the workflow window's toolbar. For example, if you type the word *music* into the search field, then the *Action* list will display all the music-related actions. When a developer builds an action, they assign keywords to the action to allow for this type of searching. The words in each action's name are automatically added as keywords as well.

Clicking on a saved workflow in the *Library* list will display, within the *Action* list, a list of actions included in that workflow. See figure 3.12. You can test this by clicking on one of the example workflows included in the *Example Workflows* group in the *Library* list.



Chapter 3: Automator's Interface

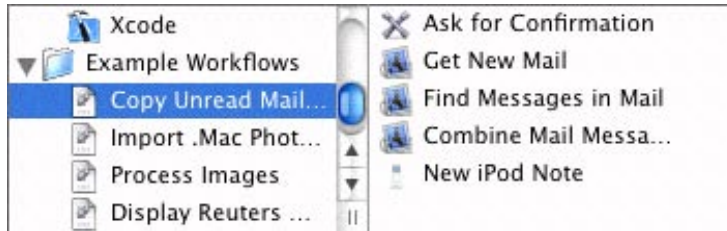


Figure 3.12 *Saved Workflow Actions*

In the bottom left of a workflow window, immediately following the + button, you will find an indicator specifying the number of actions currently displayed in the *Action* list. See figure 3.13.



Figure 3.13 *Action Count Indicator*

Description Area

The *Description* area is displayed in the lower left of the workflow window. See figure 3.14. Whenever you select an action in the *Action* list, or click on the title bar of an action in the *Workflow* view, the *Description* area will display a description of the selected action.



Burn a Disc

This action creates a CD or DVD containing the files and folders passed in from the previous action.

Requires: A drive capable of burning and a blank disc.

Input: (Files/Folders) Files and folders passed from the previous action.

Options: Name the disc, append today's date to the disc name, choose the device to use.

Result: (Files/Folders) Files and folders passed from the previous action.

Figure 3.14 *The Description Area*

An action's description will vary from action to action. However, it will typically contain a few lines of text explaining the function of the action. It will also contain other information, such as an overview of the type of input the action accepts, as well as the type of output generated by the action. Other information may be provided as well, including settings, requirements, and more. If you are having trouble determining what an action does, view its description.

Workflow View

The *Workflow* view appears on the right side of the workflow window. See figure 3.15. When you open a new workflow window, this view will be empty. It will become populated as you add actions to the workflow. If you open a saved workflow in Automator, then this *Workflow* view will display the detail for the actions



Chapter 3: Automator's Interface

included in the workflow.

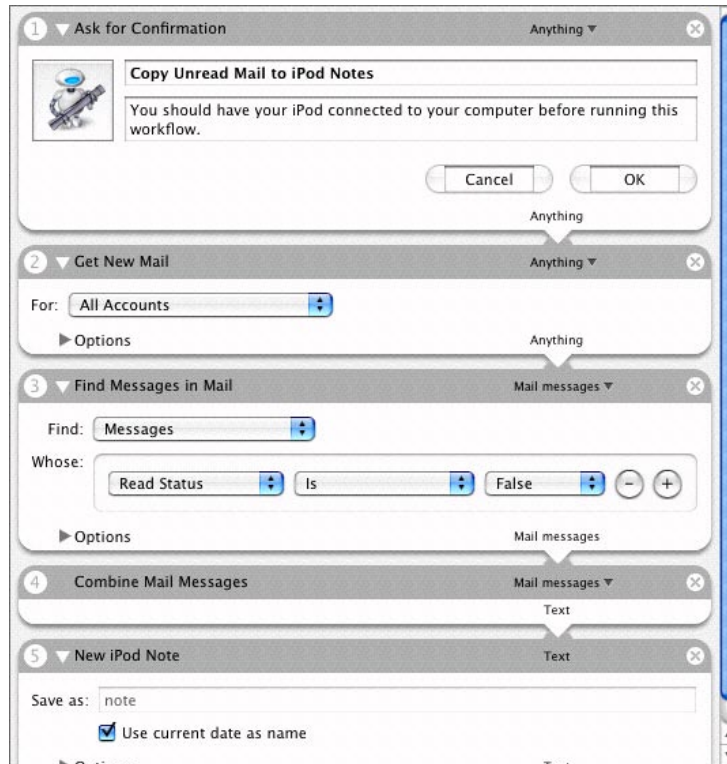


Figure 3.15 *The Workflow View*

The *Workflow* view displays a detailed view of each action in the workflow, along with its settings, input and output values, and more. We will begin working with actions and workflows in the next chapter, and we will explore the various aspects of the *Workflow* view in greater detail at that time.

Workflow Status

When running a workflow from within Automator, a status display can be found in the lower right corner of the workflow window. See figure 3.16. This status display will indicate which task is currently being performed. When processing is complete, the status display will indicate that the workflow execution has been completed. It will also indicate when a workflow fails.

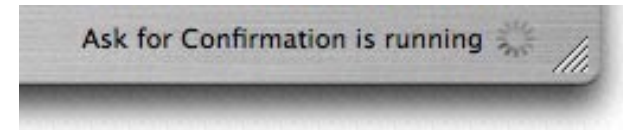


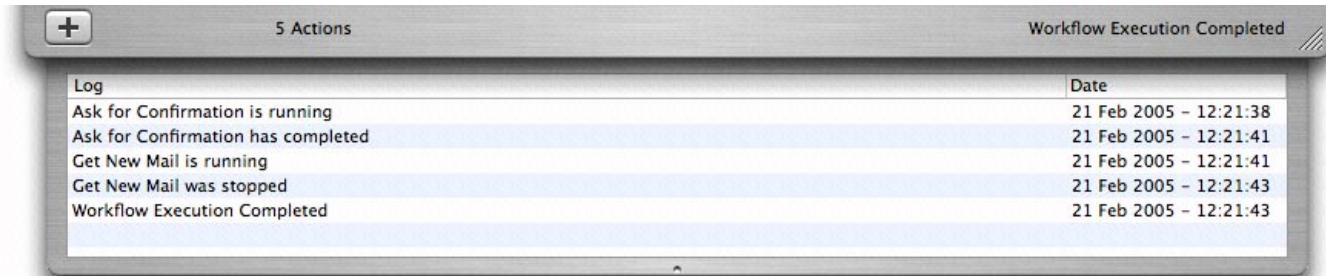
Figure 3.16 *Workflow Status Display*

Automator's Log

When running a workflow from within Automator, you may wish to view a list of all tasks that are being performed, to ensure that the workflow is performing in the desired manner. This can be done by opening Automator's *Log* drawer. See figure 3.17. To open the *Log* drawer, select *Show Log* from the *View* menu.



Chapter 3: Automator's Interface



5 Actions		Workflow Execution Completed
Log	Date	
Ask for Confirmation is running	21 Feb 2005 - 12:21:38	
Ask for Confirmation has completed	21 Feb 2005 - 12:21:41	
Get New Mail is running	21 Feb 2005 - 12:21:41	
Get New Mail was stopped	21 Feb 2005 - 12:21:43	
Workflow Execution Completed	21 Feb 2005 - 12:21:43	

Figure 3.17 Automator's Log Drawer

When a workflow is run, the *Log* drawer will become populated with a list of the tasks associated with that workflow, along with the date and time each task was performed. As we will discuss in chapter 5, the information contained within the *Log* drawer can be useful when troubleshooting problems in a workflow.

What's Next

Now that we have explored the primary aspects of Automator's interface, we are ready to begin building a workflow. In the next chapter, we will discuss, in detail, the steps involved in building a new workflow. Within this chapter, we will also explore additional aspects of Automator's interface. In particular, we will discuss an action's settings interface in the *Workflow* view.



Chapter 4 Constructing a Workflow

So far, we have discussed the basics of the Automator interface, but really haven't started using Automator yet. In this chapter, we will begin using Automator by walking through the steps involved in building a workflow. We will also discuss aspects of the interface not covered in the last chapter. The topics covered in this chapter and the next will prepare us for chapter 6, when we will put our Automator skills to the test, and build some example workflows.

Creating a New Workflow

When first launched, Automator will automatically create a new workflow window for you. If Automator is already running and you would like to create a new workflow, select *New* from the *File* menu. See figure 4.1.

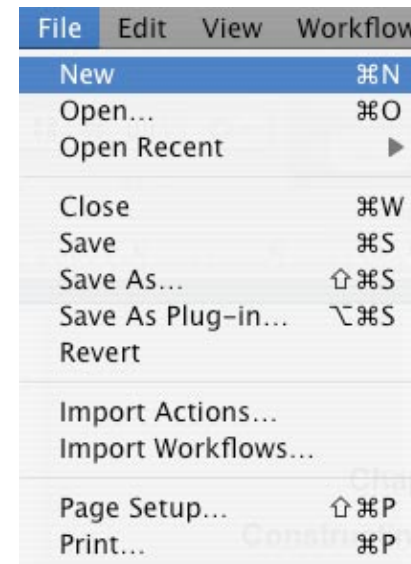


Figure 4.1 The New Command in the File Menu



Chapter 4:
**Constructing
a Workflow**

Adding Actions to a Workflow

When you create a new workflow, the *Workflow* view in the window will be empty. Your next step is to select actions appropriate for the process you wish to automate, add them to the *Workflow* view, and configure them.

To add an action to the *Workflow* view, first locate the desired action in the *Action* list. To find the appropriate action for the task you want to perform, you may need to click on a category icon in the *Library* list. For example, if you want to automate tasks in Mail, then click on the Mail category icon. If you cannot find an appropriate action, then you may want to try narrowing the list of actions, using the search field in the window's toolbar. Actions may be searched by name, category, or keyword.

While most action names are fairly descriptive, there may be an instance when you are not sure what a specific action will do. To further clarify the function of an action, click on the action, and view its description in the *Description* area.

Once you have located the desired action, select it, and drag it over into the *Workflow* view. The action's configuration interface will then be added to the workflow. See figure 4.2.

If your workflow already contains actions, then you may add a new action by dragging and dropping the action into the desired location in the workflow sequence. For example, you might drag and drop an action in-between two existing actions, or to the beginning or end of the workflow.

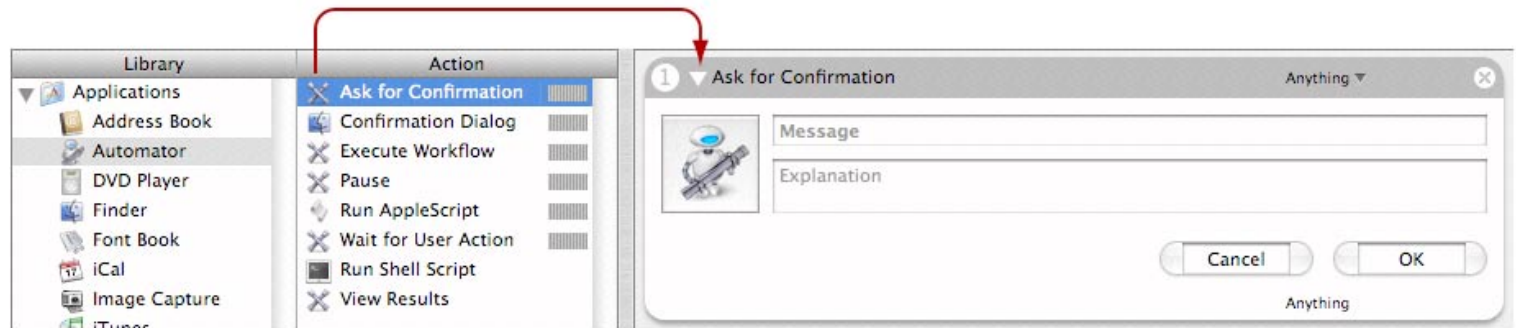


Figure 4.2 Adding an Action to a Workflow



Chapter 4:
**Constructing
a Workflow**

In some cases, when adding an action to a workflow, you may be prompted with a dialog, asking if you would like to proceed. This type of dialog typically occurs when an Action will perform a task that cannot be undone, when an Action has specific requirements, or when an Action will overwrite existing files. See figures 4.3 and 4.4.



Figure 4.3 *Example of an Action Confirmation Dialog*

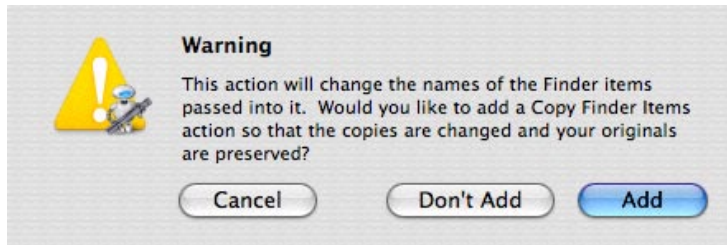


Figure 4.4 *Example of an Action Confirmation Dialog*

Configuring Action Settings

Once you've added an action to your workflow, you should check to see whether the action contains any modifiable settings. As mentioned in chapter 2, an action's settings allow you to control that action's behavior when the workflow is run.

Since every action performs a different type of task, each action's settings will be different. Because of this, when you begin working with a new action, you will need to become familiar with that action's settings.

As an example, the *Ask for Confirmation* action allows you to configure the text to be displayed to the user, as well as the names of the *Cancel* and *OK* buttons. See figure 4.5 A. The *Get New Mail* action allows you to specify the email account that should be checked for new messages. See figure 4.5 B. Modifying the settings for these, or any actions, will affect how the workflow behaves when it is run.



Chapter 4: Constructing a Workflow



Figure 4.5 *Modifying Action Settings*

In some cases, an action may not require any settings to be configured at all. For example, the *Update iPod* action contains no configuration settings. See figure 4.6.



Figure 4.6 *Example of an Action With No Settings*

Allowing a User to Specify Settings

Rather than forcing an action to use the same settings every time the workflow is run, you may wish to allow the user to adjust the action's settings themselves. This allows a workflow to be more flexible, so that it can be easily run by other users, or on other computers.



Chapter 4:
**Constructing
a Workflow**

Not all actions will let you provide users with the power to override settings during processing, and in some cases, you wouldn't want them to. However, those actions that do allow for this type of configuration will contain an *Options* disclosure triangle at the bottom of the action's interface. Clicking this triangle will expand the action's interface, displaying additional configuration options. See figure 4.7.

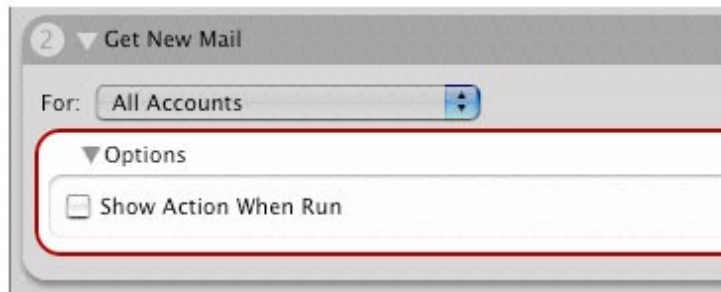


Figure 4.7 Action Options

Once expanded to display these additional options, the action's interface will include a *Show Action When Run* checkbox. By selecting this checkbox, the action's settings will be displayed to the user when the workflow is run, allowing adjustments to be made, if necessary. Figure 4.8 shows the settings window that would appear while processing the *Get New Mail* action shown in figure 4.7.

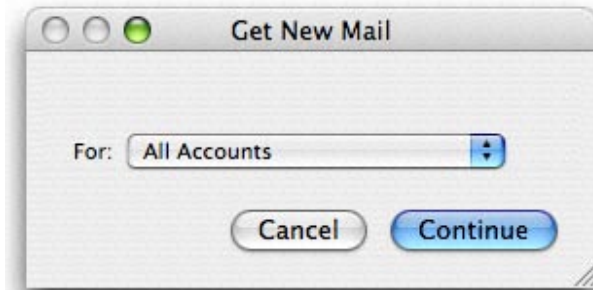


Figure 4.8 User Modifiable Settings for the *Get New Mail* Action

When allowing the user to modify an action's settings, it is important to note that, in some cases, not every single setting will be available to the user. To ensure that an action does not produce an error when run, be sure to specify all settings when configuring an action. If the user is prompted to modify the settings during processing, than any pre-configured settings will simply serve as the default settings.

In some cases, you may want to limit the settings that a user is allowed to modify. For example, as you can see in the *New iPod Note* action in figure 4.9, you can select *Show Entire Action* to display all settings to the user or *Show Selected Items* to show a limited set of items to the user.



Chapter 4:
**Constructing
a Workflow**

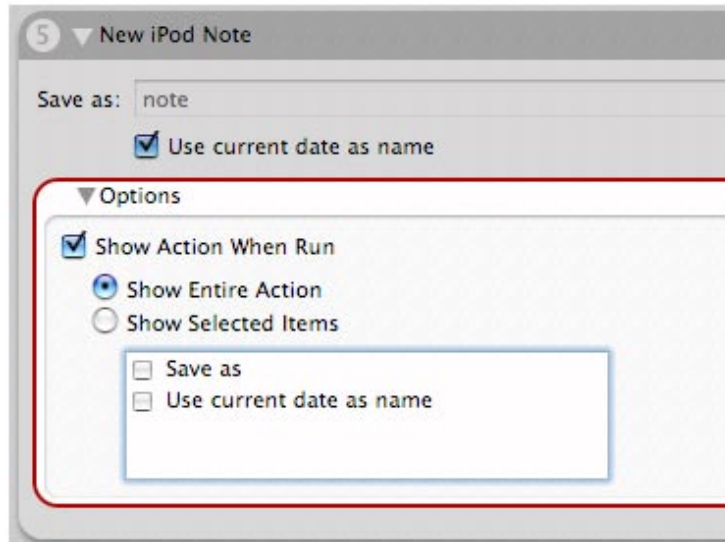


Figure 4.9 *Additional Action Options*

Working with Input and Output Values

When building a workflow, every action will have an input value, regardless of whether it is used by the action. In addition, every action will also have an output value. These input and output values allow actions to interact with one another in order to form a complete workflow from start to finish. An output value from one action is passed on as input to the next action in a workflow sequence.

Some actions can receive input in any format. However, other actions may require a specific type of input. The type of input required by an action is displayed in the upper right corner of the action's display. See figure 4.10 A. An action whose input type is set to *Anything* can be triggered at any point in a workflow, whereas, an action that requires a specific input type must follow an action that outputs that same specific type of value.



Chapter 4: Constructing a Workflow

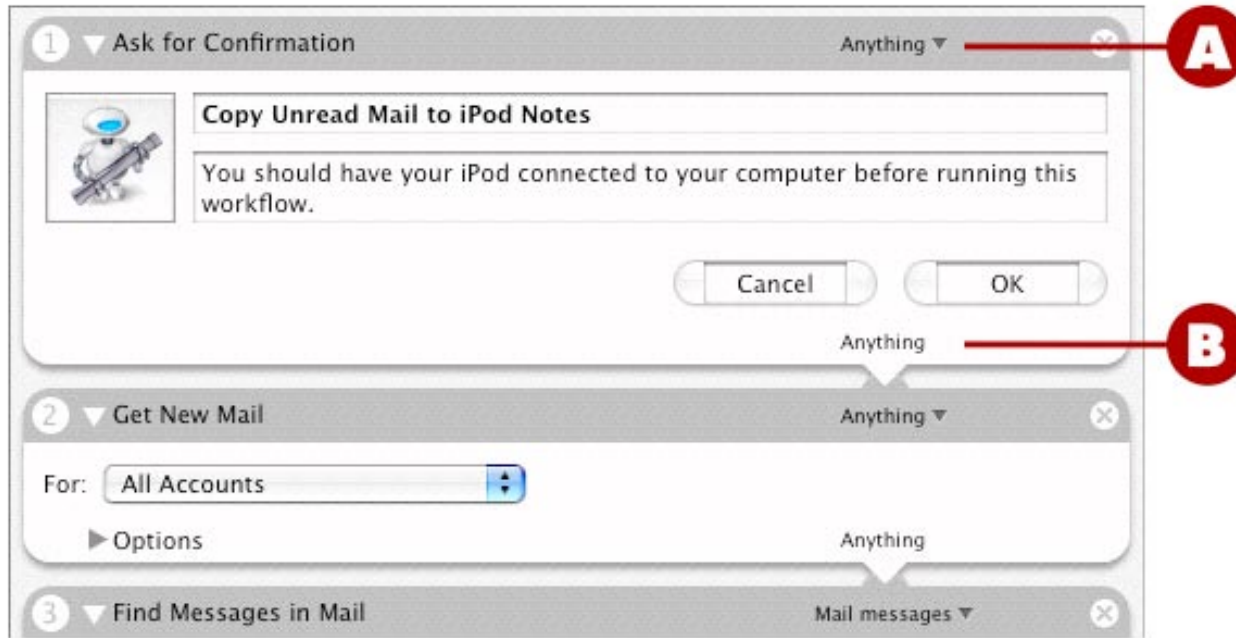


Figure 4.10 Action Input and Output Values

The output value of an action may be either the same type as the input value, or it may be a different type. The type of output produced by an action is displayed in the lower right corner of the action's display. See figure 4.10 B.

Typically, the values input to an action are processed or modified in some way and then passed down as the action's output. For example, the *Rotate Images* action will accept a list of image files as its input, rotate those images, and then pass the modified image files down as output.

Other times, a value that is completely unrelated to the

action's input may be passed as output. For example, the *Add Attachments to Front Message* action will accept file or folder paths as input, but will output a reference to the front outgoing email message in Mail.

Some actions can even be configured to ignore their input values all together, such as the *Get Specified Finder Items* action. This action will allow a user to specify file or folder paths to be passed down as output to the next action in the workflow. If configured to use its input value, then any input paths will be appended to the list of output paths. If configured to ignore its input value, then only the paths specified within the action will be passed as output. In chapter 6, we will construct



Chapter 4:
**Constructing
a Workflow**

a workflow that uses the *Get Specified Finder Items* action, and we will configure it to ignore its input value.

Input and Output Value Special Handling

Certain actions allow you to customize the action's input type. Actions that allow this customization will feature a small triangle to the right of the action's input type. Take a look at figure 4.11 A, and you will notice a small triangle just to the right of the word *Anything*. Click on that triangle and a popup menu will appear (see figure 4.11 B), allowing you to specify how the action's input value is treated. This popup menu allows you to configure the action to use the results from the previous action as input, or to ignore the results of the previous action.

Mismatched Input and Output Values

Automator provides visual clues that will help you to see when you have mismatched two actions in a workflow sequence. When an action requires an input value of a specific type, and the previous action does not provide an appropriate output type, the mismatched input and output values for the two actions will be displayed in **red** within the *Workflow* view. See figure 4.12.

As you configure a workflow, be on the lookout for mismatched input and output values, as it will indicate a problem in the workflow, which will more than likely keep your workflow from running successfully.

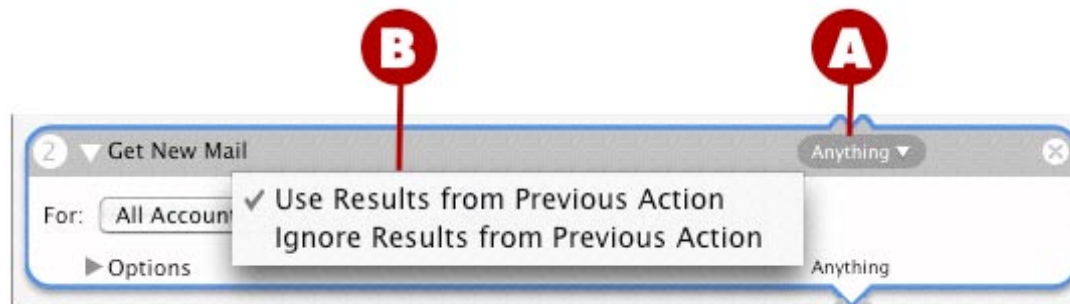


Figure 4.11 Action Input Settings

We will use this technique in an example workflow that we will construct in chapter 6.



Chapter 4:
**Constructing
a Workflow**

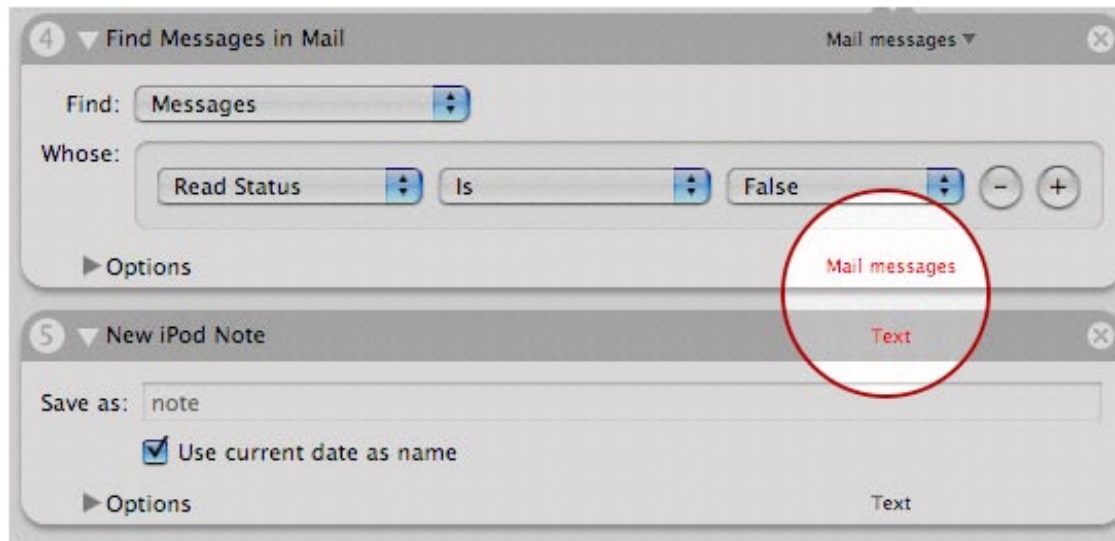


Figure 4.12 Mismatched Action Input and Output Values



Figure 4.13 Collapsing an Action



Chapter 4:
**Constructing
a Workflow**

Collapsing Actions in a Workflow

As your workflows become more complex, you will find that your workflows will become longer than will fit in the window without scrolling. Fortunately, Automator makes it easy to collapse individual actions.

To collapse an action, click the small disclosure triangle to the left of the action's name in the action interface's title bar. See figure 4.13. Doing so will condense the action, hiding the settings, but leaving the action interface still visible. To expand the action, click the triangle next to the action's name in the title bar again.

Deleting Actions from a Workflow

To delete an action from a workflow, click the X button in the upper right corner of the action's title bar. See figure 4.14 A.

You may also delete an action by clicking on the action's sequence number in the title bar of the action's interface (see figure 4.14 B), and selecting *Delete* from the popup menu. See figure 4.15. Holding down the *control* key, and clicking on the action's title bar will display this same popup menu as a contextual menu.

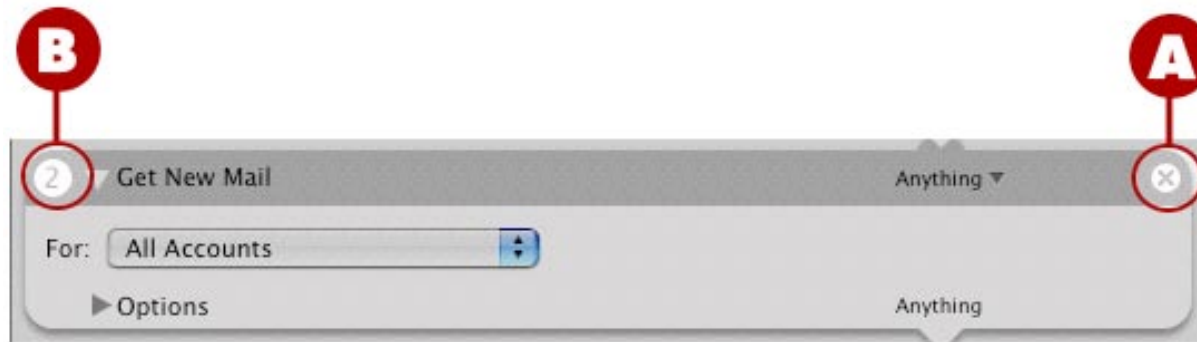


Figure 4.14 An Action Toolbar



Chapter 4:
**Constructing
a Workflow**

Move before "Ask for Confirmation"
Move after "Find Messages in Mail"
Move after "Combine Mail Messages"
Move after "New iPod Note"

Disable
Rename

Delete

Figure 4.15 *Deleting an Action From an Action's Popup Menu*

Actions may also be deleted by selecting an action in the workflow by clicking on its title bar, and then selecting *Delete* from the *Edit* menu. See figure 4.16.



Figure 4.16 *Deleting an Action from the Edit Menu*

A note of warning, though. When you delete an action from a workflow, you will not be prompted to confirm that this is actually what you want to do. Rather, the action will be deleted immediately, settings and all. So, before you choose to delete an action, make sure that you *really* want to delete it.



Chapter 4:
**Constructing
a Workflow**

Disabling Actions in a Workflow

Rather than completely deleting an action from a workflow, you also have the option of disabling an action. This might be useful for testing purposes, by allowing you to see how a workflow behaves without a certain action, but without deleting it entirely. If the workflow behaves appropriately without the action enabled, you can decide to delete the action at that time. One extremely helpful benefit of disabling an action is that, if you do change your mind, you can always re-enable the action, and you won't lose any of the settings that you may have configured.

To disable an action in a workflow, click on the action's sequence number in the action's title bar, and then select *Disable* from the popup menu. See figure 4.17.

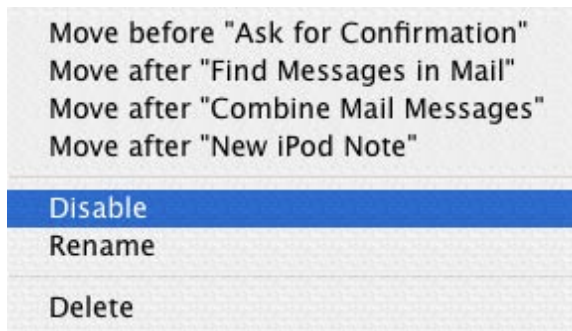


Figure 4.17 *Disabling an Action*

When an action has been disabled, it will appear to be grayed out, and the sequence number in its title bar will be changed to a -. See figure 4.18.

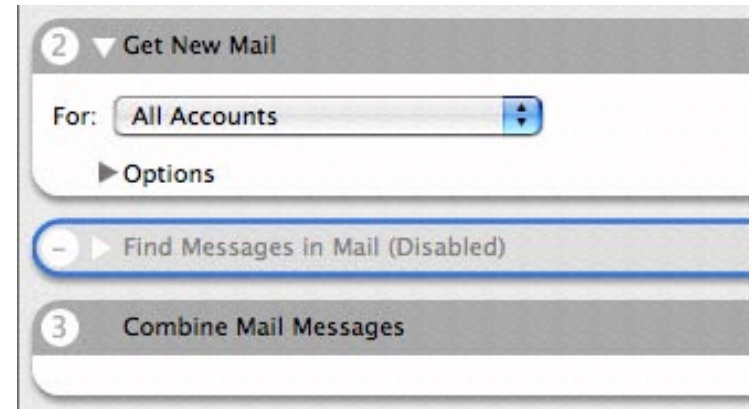


Figure 4.18 *A Disabled Action*

To re-enable an action that has been disabled, click on the - in the upper left of the action's title bar, and select *Enable* from the popup menu.



Chapter 4:
**Constructing
a Workflow**

Moving an Action in a Workflow

Actions that have been added to a workflow may also be rearranged. You will find that this is sometimes necessary as your workflow begins to take shape. To move an action in a workflow, click on the action's sequence number in the action's title bar, and select the desired location for the action from the popup menu. See figure 4.19.

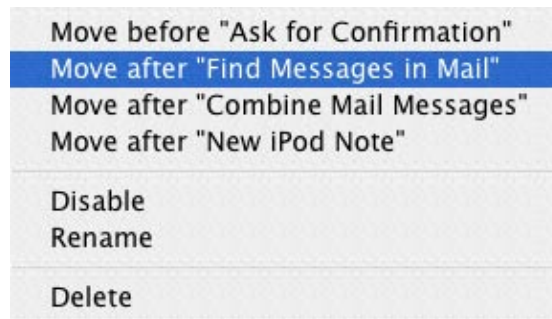


Figure 4.19 *Moving an Action*

Another way to move actions is to click on the action's title bar, and drag it to the desired location in the workflow. This may be helpful when working with a longer workflow, rather than looking through a lengthy popup menu. Actions may also be copied to the clipboard, and then pasted back to the end of a workflow.

What's Next

In this chapter, we have discussed various aspects of constructing a workflow, and configuring the actions that make up that workflow. In the next chapter, we will explore some of the things that you can do with a workflow, once it has been configured. Then, in chapter 6, we will walk through the process of building some example workflows.



Chapter 5 Utilizing a Workflow

Now that we have discussed the primary tasks involved in constructing a new workflow, let's discuss some of the things that you can do with a workflow, once it has been built.

Saving Workflows

The first thing that you will want to do once you have created a workflow is to save it. A workflow can be saved in a number of ways. It can be saved as a workflow file, as an application, or as a plug-in for another application. We will discuss each of these in detail in this chapter.

Once the workflow is saved as a workflow file or an application, it can easily be distributed to other users, to be run on their machines. Of course, this assumes that the recipient of the workflow has first installed all of the applications with which the workflow will interact, as well as all of the actions that are used within the workflow. For example, if you email a workflow that interacts with iMovie to someone, then the person you send it to must have iMovie installed on his or her machine in order for the workflow to function. The person must also be using a version of Mac OS X that supports Automator, and must have any actions utilized in the workflow installed on the machine.



Chapter 5: Utilizing a Workflow

Saving as a Workflow File

When saving a workflow, one option is to save the workflow as a workflow file. A saved workflow file will open in Automator when double clicked in the Finder. To save a workflow as a workflow file, select *Save As* from the *File* menu in Automator's menu bar. See figure 5.1.

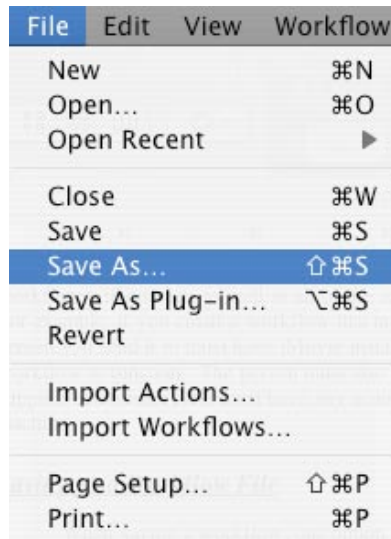


Figure 5.1 Save As Menu Item

Next, you will be prompted to specify an output folder and name for the saved workflow, as well as a file format. Select a file format of *Workflow* from the *File Format* popup menu in the save dialog. See figure 5.2.

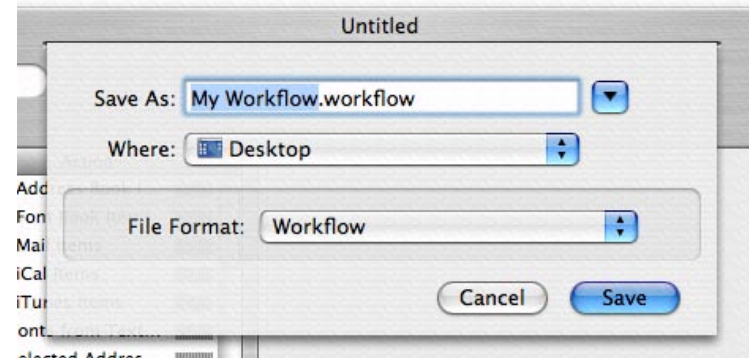


Figure 5.2 Saving a Workflow File

Once a workflow has been saved as a workflow file, it will appear in the Finder as a document icon containing the standard Automator robot figure. See figure 5.3. The workflow will also automatically be added to the *My Workflows* group in the *Library* list in the workflow window.



Figure 5.3 A Saved Workflow File

As mentioned before, saved workflow files can easily be distributed to other users, if desired. Once received by the other user, a workflow file may be imported into Automator on that user's machine. We will discuss importing later in this chapter.



Chapter 5: Utilizing a Workflow

Saving as an Application

When selecting *Save As* from the *File* menu in Automator's *File* menu, you will notice that the *File Format* popup menu in the save dialog also contains another file format. See figure 5.4. If desired, you may choose to save a workflow as an application.

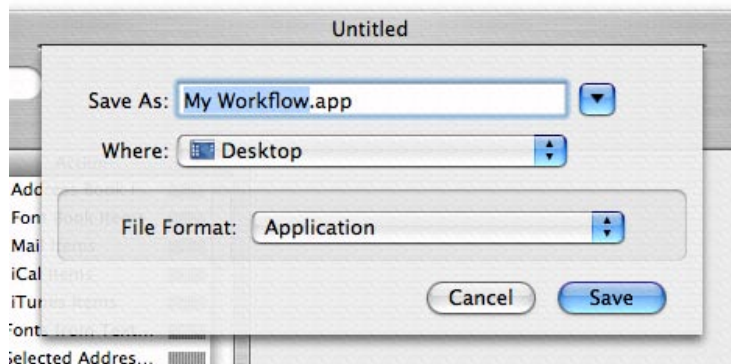


Figure 5.4 Saving a Workflow Application

A workflow that has been saved as an application will appear in the Finder as a robot icon atop a diamond-shaped platform. See figure 5.5.



Figure 5.5 A Saved Workflow Application

A workflow application file will behave like any other application in Mac OS X. It may be launched by double

clicking on it in the Finder, or by triggering it from the Dock. Once launched, a workflow saved as an application will execute in the same manner as if the workflow had been run from within Automator.

When a workflow application is launched, a status indicator will display the tasks being performed in the computer's menu bar. See figure 5.6. To the right of the workflow status indicator, a small red stop button may be clicked to stop the workflow during processing.

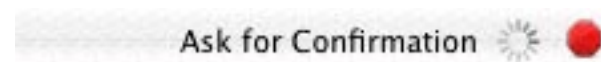


Figure 5.6 Workflow Status in the Menu Bar

A workflow that has been saved as an application may also be launched by dragging and dropping items onto the saved application's icon in the Finder. If the first action in the workflow will accept file and folder paths as its input value, then the paths to the dropped items will be passed as input to that first action for processing.

Workflow applications may be opened within Automator by selecting *Open...* from the *File* menu. However, workflow applications may not be imported into a workflow group folder in Automator's *Library* list. Like workflow files, workflow applications may be distributed to other users, to be run on their machines, so long as the users have installed any components required for the workflow to run successfully.



Chapter 5: Utilizing a Workflow

Saving as a Plug-in

While saving a workflow as an application may be sufficient in many cases, Automator also provides a way to save a workflow so that it can serve as a *plug-in* for another application, or for the system itself. Saving a workflow as a plug-in may allow the workflow to be triggered in a more efficient, or unique manner than is otherwise possible when saving the workflow as a workflow file or a workflow application. For example, you can create a plug-in workflow that is triggered by the Finder every time a file is added to a specified folder. You can also create a plug-in that is launched from the Finder's contextual menu. You'll see these and more examples as you make your way through the next few sections.

To save a workflow as a plug-in, select *Save as Plug-in* from the *File* menu in Automator's menu bar. See figure 5.7. You will be presented with a save dialog.

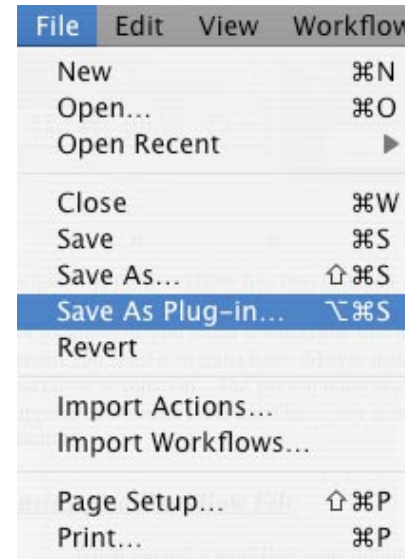


Figure 5.7 Save As Plug-in Menu Item

Notice that this save dialog is different from that of a standard save dialog. You may still enter a name for the saved plug-in. However, rather than specifying an output folder location, you will specify the type of plug-in that you would like to create from the *Plug-in for* popup menu. See figure 5.8. Once you have made a selection, click the *Save* button in the dialog, and Automator will automatically save the workflow and install it in the appropriate location on your hard drive for the type of plug-in that you specified.



Chapter 5: Utilizing a Workflow

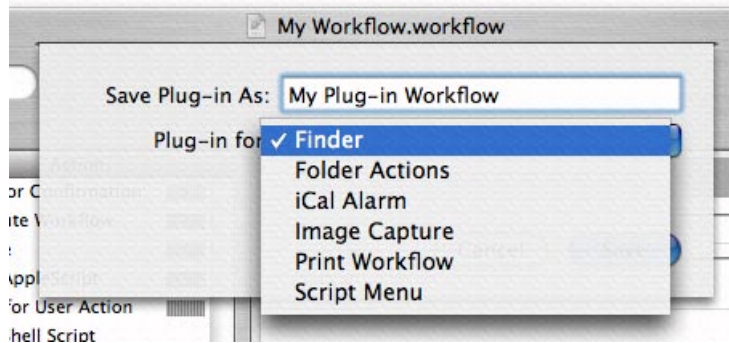


Figure 5.8 Plug-in Types

Automator currently supports saving a number of different types of plug-ins, which we will now discuss.

Finder Plug-ins

A workflow that has been saved as a Finder plug-in will appear in the Finder's contextual menu, allowing you to trigger it quickly from anywhere within the Finder.

To display the Finder's contextual menu, hold down the control key, and click on anything in the Finder. Once the contextual menu displays, you will find the saved workflow plug-in listed in the contextual menu under Automator. See figure 5.9.



Figure 5.9 An Installed Finder Plug-In

When triggering a workflow from the Finder's contextual menu, the paths to any selected items will be passed to the first action in the workflow's sequence. For example, if you trigger a workflow after control clicking on a folder, then the path to that folder will be passed to the first action in the workflow as input. Control clicking on an opened folder window will cause the path to the window's folder to be passed to the first action in the workflow. Control clicking on the desktop will cause the path to the desktop folder to be passed to the first action in the workflow.

Once saved, Finder plug-ins may be found in the *Library > Workflows > Applications > Finder* folder in your user folder.



Chapter 5: Utilizing a Workflow

Folder Action Plug-ins

Folder Actions are actually AppleScript files that can be attached to folders in the Mac OS X Finder. An AppleScript that has been created as a Folder Action may be configured to trigger when a specific type of action is taken on the folder to which the script is attached. For example, a Folder Action may be configured to trigger when items are placed into the attached folder, when items are removed from the attached folder, when the folder's window is opened, or when the folder's window is closed.

Automator gives you the option of attaching a workflow to a folder as a Folder Action, causing the workflow to be triggered whenever items are added to the attached folder. When you select this type of plug-in within the save dialog, additional save options will appear.

A second popup menu in the save dialog will allow you to specify the folder to which the workflow should be attached. See figure 5.10 A. This popup menu contains a list of several standard folders by default, from which you may choose, including the *Desktop*, your user folder, and your *Documents* folder. To specify a different folder that is not displayed in this list, select *Other* in the popup menu.

For Folder Actions to function in Mac OS X, they must be enabled system-wide. If Folder Actions have not yet been enabled on your machine, then enable them by selecting the *Enable Folder Actions* checkbox. See figure 5.10 B. Please note that this checkbox will only be shown in the save dialog if Folder Actions are disabled on your machine. Click the *Save* button to attach the workflow to the specified folder.

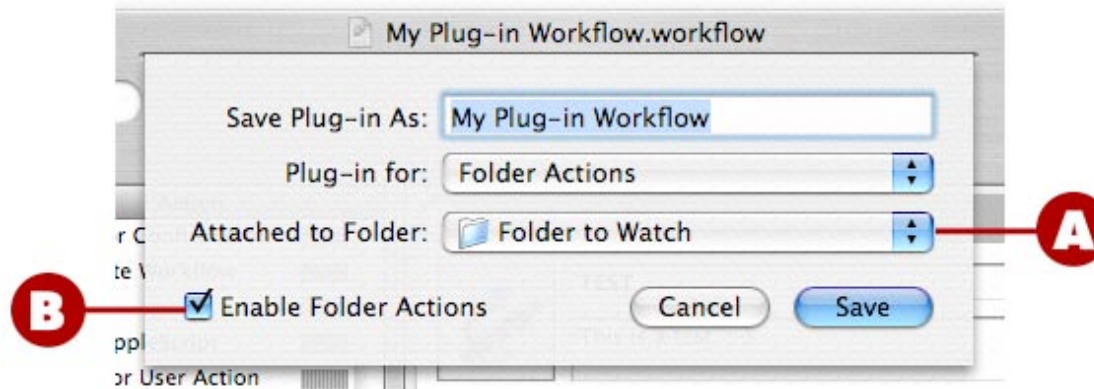


Figure 5.10 Saving a Folder Action Plug-in



Chapter 5: Utilizing a Workflow

Once attached to a folder, the workflow will be triggered whenever items are placed into the folder. Saving a workflow in this manner can allow you to create a folder that serves as a hot folder, allowing for some exciting automation possibilities. For example, you could attach a Folder Action workflow plug-in to your incoming fax folder, to be triggered whenever a new fax arrives.

It is worth noting that when you choose to save a workflow as a Folder Action plug-in, the workflow itself does not actually become attached to the folder. Rather, the workflow is saved as an application into a folder on your hard drive. Next, Automator dynamically creates an AppleScript file that will launch the saved workflow application, and it attaches this AppleScript file to the specified folder. When the Folder Action is triggered, the AppleScript will be triggered, which will launch the workflow and pass the newly added folder items to the workflow, in order to be processed.

After a workflow has been saved as a Folder Action plug-in, you may confirm that the Folder Action was successfully applied to a folder by locating and control clicking on the attached folder in the Finder. Doing so will produce the Finder's contextual menu. Within this menu, you will find *Remove a Folder Action* and *Edit a Folder Action* menu items. See figure 5.11. To confirm that your Folder Action was successfully attached to the folder, select one of these menu items, and verify that your workflow's name is displayed, with an extension of *.sctpt*.

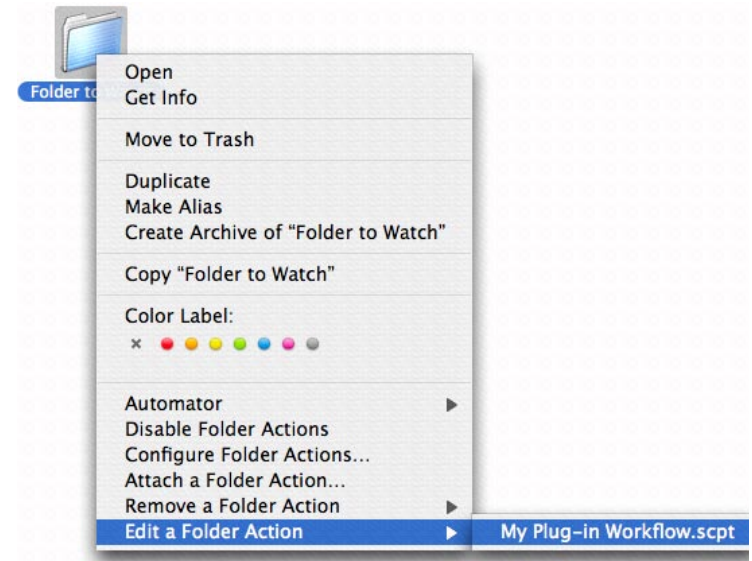


Figure 5.11 An Installed Folder Action Plug-in

The Finder's contextual menu contains a number of menu items related to Folder Actions. One menu item will allow you to enable or disable Folder Actions. This is a system-wide change, and if you disable Folder Actions on one folder, all other Folder Actions on your machine will be rendered inactive until Folder Actions are enabled again.

Another menu item in the Finder's contextual menu is labeled *Configure Folder Actions*. Selecting this menu item will automatically launch the Folder Actions Setup application on your machine, which is located in the *Applications > AppleScript* folder. The Folder Actions Setup application provides a central location for working with Folder Actions that have been configured



Chapter 5: Utilizing a Workflow

on your machine. See figure 5.12. From here, Folder Actions may be enabled, disabled, added, or removed. We will not cover the exact details regarding the usage of this application within this book. However, this application is fairly straightforward, and you are encouraged to explore it on your own.

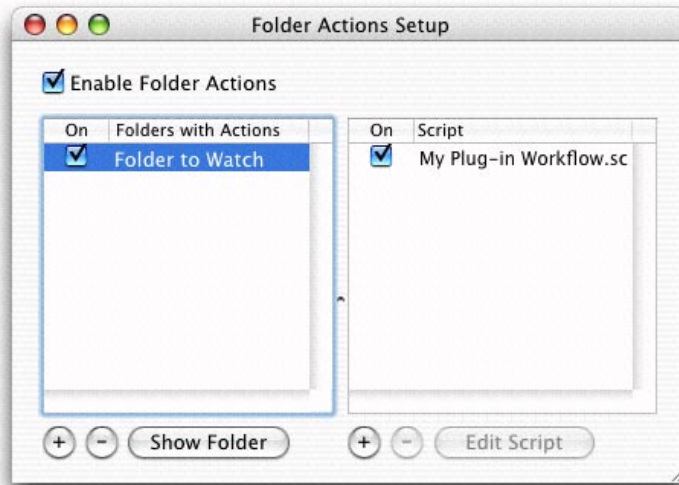


Figure 5.12 Folder Actions Setup Application

To prove that a Folder Action plug-in is actually an AppleScript file that triggers a workflow application, you may select the Folder Action's name under the *Edit a Folder Action* menu in the Finder's contextual menu. This will cause the selected Folder Action's AppleScript code to be opened and displayed in the Script Editor application. See figure 5.13. If you are at all familiar with AppleScript, you will see that this code simply passes the paths to any newly detected items to the

workflow, which is actually saved as an application.

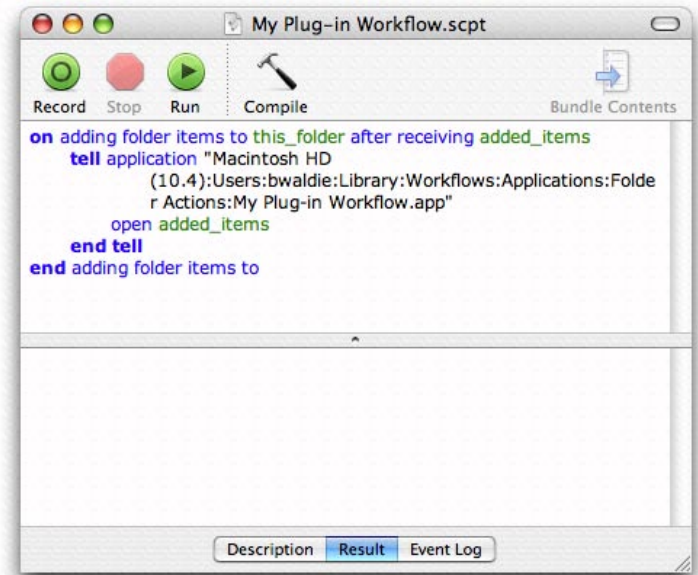


Figure 5.13 AppleScript Code for Triggering a Folder Action Plug-in

Once saved, Folder Action plug-ins may be found in the *Library > Workflows > Applications > Folder Actions* folder in your user folder.

iCal Alarm Plug-ins

Automator will also allow you to save a workflow as an iCal Alarm, allowing you to configure the workflow to trigger at a specific date and time. Once saved, you may even configure the alarm to trigger on a repeating schedule, such as every day at midnight.

Saving a workflow as an iCal Alarm will automatically



Chapter 5:
**Utilizing a
Workflow**

launch iCal and create a new event for the workflow. See figure 5.14. The newly created event will appear at the current time on the current date in your calendar. If you click on the event, you will notice that it has been configured with an alarm that will open the workflow, which Automator saved as an application into a folder on your hard drive.

Once the event has been created, you may adjust the date, time, and alarm settings for the event within iCal.

Once saved, iCal Alarm plug-ins may be found in the *Library > Workflows > Applications > iCal* folder in your user folder.

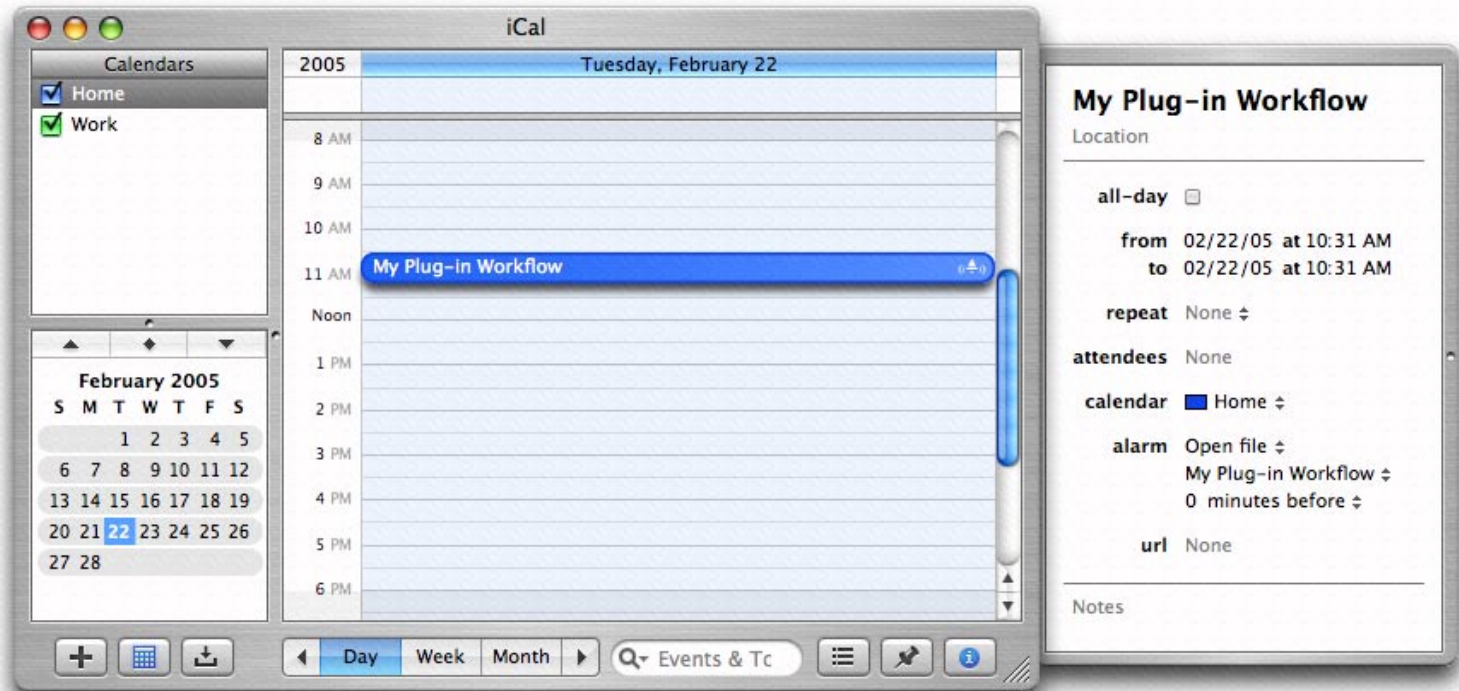


Figure 5.14 An Installed iCal Plug-in



Chapter 5:
**Utilizing a
Workflow**

Image Capture Plug-in

Another type of plug-in that Automator supports is a plug-in for the Image Capture application. Saving a plug-in of this nature will cause the workflow to appear in the *Automatic Task* popup menu in Image Capture when importing photos from a digital camera. See figure 5.15.

When importing photos, you may select the workflow from the *Automated Task* popup menu, and the workflow will be triggered once the download is complete. The paths to the downloaded images will be passed to the first action in the workflow sequence as input, allowing you to apply further processing to the downloaded photos.

Once saved, Image Capture plug-ins may be found in the *Library > Workflows > Applications > Image Capture* folder in your user folder.

Print Workflow Plug-in

You are probably already aware that you can convert any Mac OS X document to PDF format by selecting *Save as PDF* at the bottom of the print dialog. Automator allows you to save a workflow as a Print Workflow plug-in, further enhancing your PDF conversion abilities.



Figure 5.15 An Installed Image Capture Plug-in



Chapter 5: Utilizing a Workflow

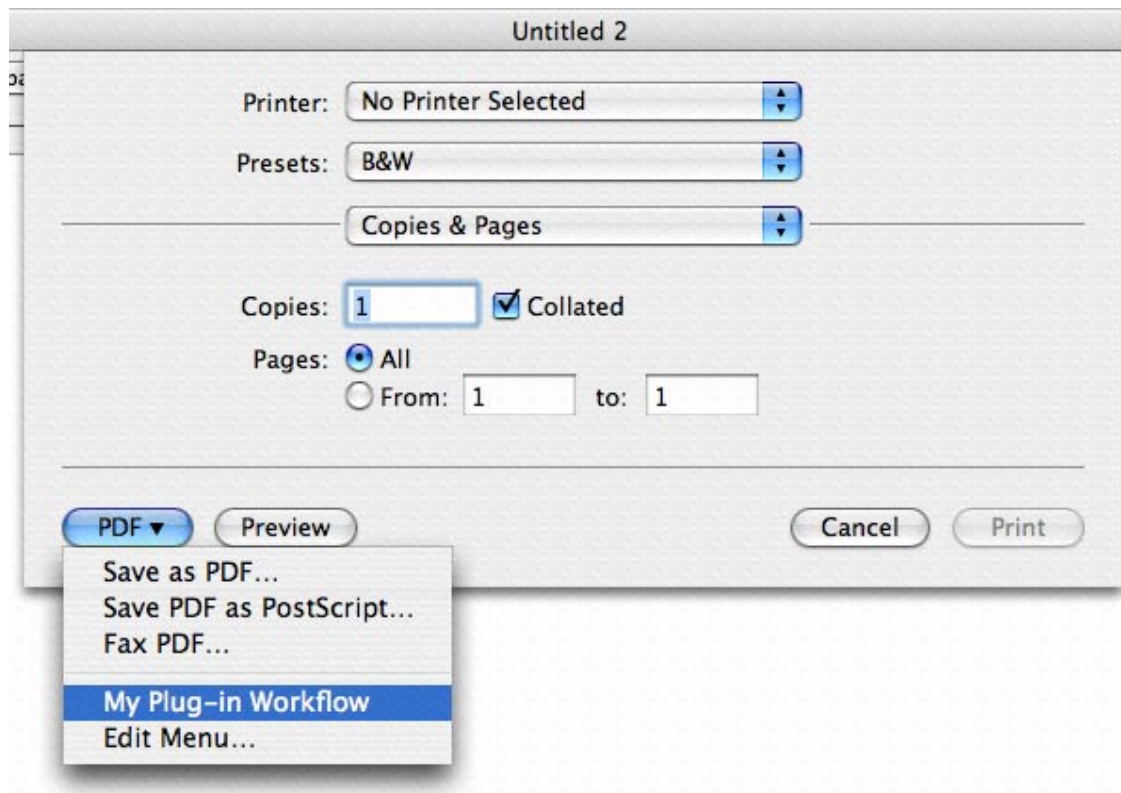


Figure 5.16 An Installed Print Workflow Plug-in

When you save a workflow as a Print Workflow plug-in, the workflow will appear at the bottom of the print dialog, beneath the *Save as PDF* option. See figure 5.16. Then, when printing, you may select the workflow to begin processing.

When a Print Workflow plug-in is triggered from the print dialog, the document that you are printing will be converted to PDF format, and the workflow will be triggered and passed the path to the newly generated

PDF. For example, you might create a workflow that will automatically create a new outgoing email message in Mail, and attach the newly generated PDF.

Once saved, Print Workflow plug-ins may be found in the *Library > PDF Services* folder in your user folder.

Script Menu Plug-in

Mac OS X contains a system-wide Script Menu, which may be configured to appear in the menu bar at all



Chapter 5: Utilizing a Workflow

times. The Script Menu can provide quick access to AppleScripts, PERL scripts, Shell scripts, and Automator workflows from within any application. This menu is disabled by default, and can be enabled manually with the AppleScript Utility application, located in the *Applications > AppleScript* folder. See figure 5.17.

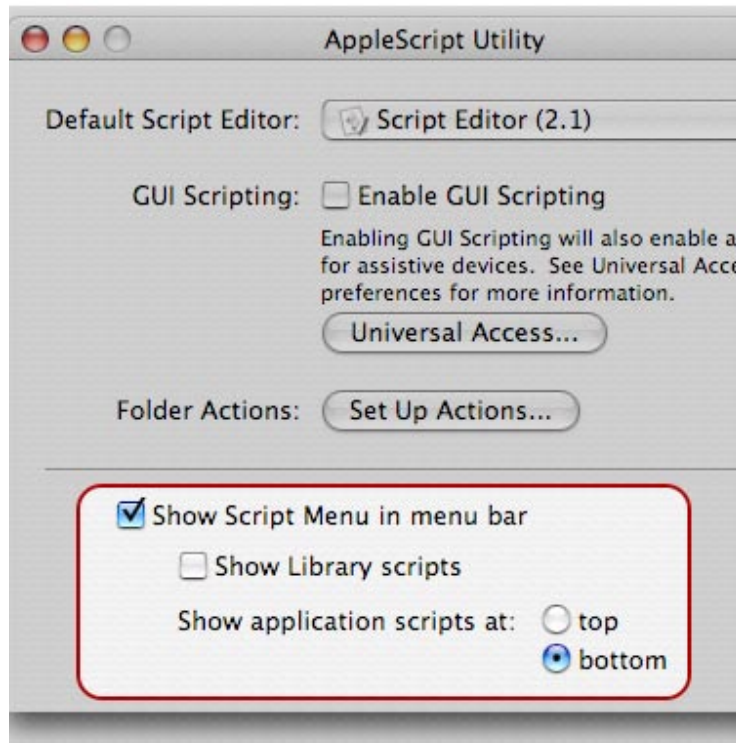


Figure 5.17 Enabling the Script Menu Manually in AppleScript Utility

When saving a Script Menu plug-in from Automator, the Script Menu will automatically become enabled for you. Once a workflow has been saved as a Script Menu plug-in, it will appear in the Script Menu, where you may select it at any time to trigger the workflow. See figure 5.18.

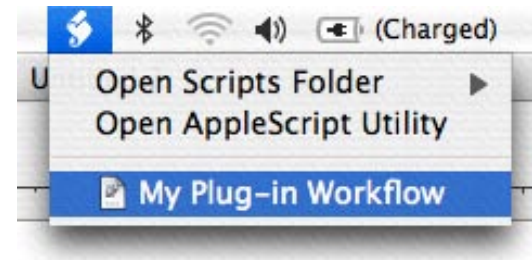


Figure 5.18 An Installed Script Menu Plug-in

Once saved, Script Menu plug-ins may be found in the *Library > Scripts* folder in your user folder.



Chapter 5:
**Utilizing a
Workflow**

Opening a Workflow

At times, you may want to open a saved workflow in Automator to adjust action settings, or to add new actions to the workflow. To open a workflow that has been saved as a workflow file, you may simply double click on the workflow file in the Finder. Doing so will launch Automator and display the workflow.

To open a workflow that has been saved as an application, you may drag and drop the workflow application onto the Automator icon in the Dock, or in the Finder.

You may also optionally select *Open* from the *File* menu, in order to open either a workflow file or a workflow application. See figure 5.19.

Please note that, in order to open a workflow, regardless of how it has been saved, you must have any applications with which the workflow interacts installed on your machine. You must also have any actions called upon in the workflow installed on your machine.

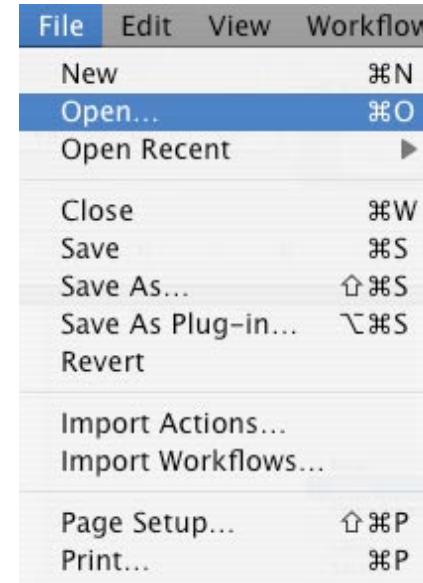


Figure 5.19 Open Workflow Menu Item



Chapter 5:
**Utilizing a
Workflow**

Printing a Workflow

As you develop a workflow, you may find it useful to print the workflow. Printing a workflow will allow you to place the entire workflow on paper right in front of you, which may be useful with lengthy workflows. With a workflow in front of you, you can review the actions and their settings all at once in order to ensure that things are configured properly. You can also consider potential areas for improvement, or additional actions that you may wish to add to the workflow. See figure 5.20.

To print a workflow, open the workflow and select *Print Workflow* from the *File* menu. See figure 5.21. You may also need to modify the page setup prior to printing, which can be done by selecting *Page Setup* from the *File* menu.



Figure 5.20 Example of a Printed Workflow



Chapter 5: Utilizing a Workflow



Figure 5.21 *Print Menu Item*

When printing a workflow, as with printing in any Mac OS X application, you also have the option to print to a PDF file. This may be useful as well. To print to PDF, select *Save as PDF* from the *PDF* popup at the bottom of the print dialog.

Running a Workflow

When creating a workflow, you may choose to run the workflow directly within Automator. This can be done by clicking the *Run* button in the workflow window's toolbar. See figure 5.22.



Figure 5.22 *Run Button*

You may also run an opened workflow by selecting *Run* from the *Workflow* menu. See figure 5.23.

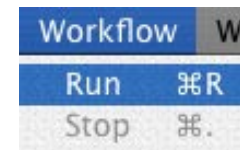


Figure 5.23 *Run Menu Item*

When you run a workflow using one of these methods, Automator will execute the workflow, beginning with the first action in the sequence, and proceeding through all subsequent actions. As an action processes within Automator, a spinning status indicator will appear at the bottom of the action's interface. See figure 5.24.



Chapter 5: Utilizing a Workflow

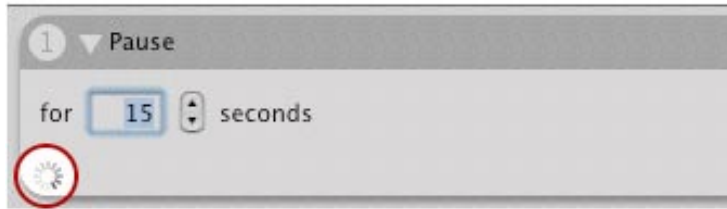


Figure 5.24 Action Run Status Indicator

Each time an action in the workflow is finished processing, a small green checkmark icon will appear in the lower left corner of the action. See figure 5.25. If an error occurs while an action is processing, then this green checkbox will be replaced by a red X. See figure 5.26.

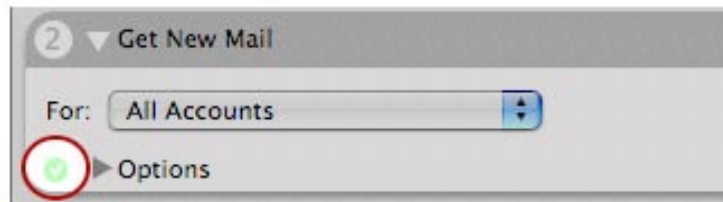


Figure 5.25 This action ran successfully

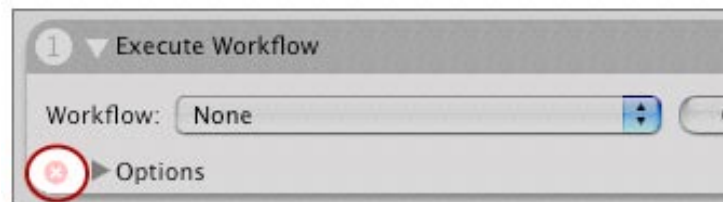


Figure 5.26 An error occurred when this action was run

As mentioned in chapter 3, the bottom of the window will also display a continuously updating status of the tasks occurring within the workflow. See figure 5.27.

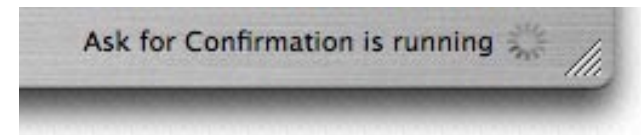


Figure 5.27 Workflow Status Indicator



Chapter 5:
**Utilizing a
Workflow**

Troubleshooting

As you run a workflow, you may need to do some troubleshooting if you do not get the expected results, or if an error occurs during execution. If an error does occur, then you may be presented with a dialog message. In some cases, this dialog may be descriptive enough to help you to determine the problem. See figure 5.28. In other cases, it may appear quite cryptic. See figure 5.29

The best way to conduct troubleshooting of a problematic workflow is to open the workflow and run it from within Automator. If the problematic workflow exists as a saved application, then you will need to open the workflow in Automator by dragging it onto the Automator application icon, or by selecting *Open* from Automator's *File* menu.



Figure 5.28 A Descriptive Error Message

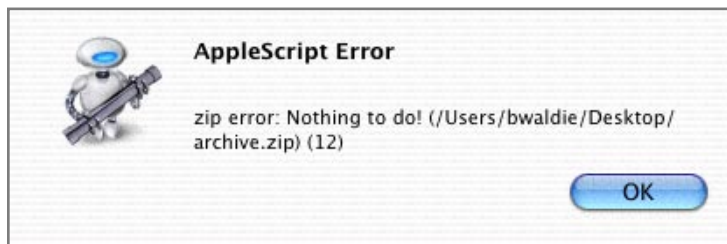


Figure 5.29 A Cryptic Error Message



Chapter 5:
**Utilizing a
Workflow**

Checking Action Settings

When troubleshooting a workflow, be sure to review all of the actions in your workflow, checking their settings and verifying that there are no mismatched input and output values between actions. See figure 5.30.

Mismatched input and output values may cause a workflow to fail during execution.

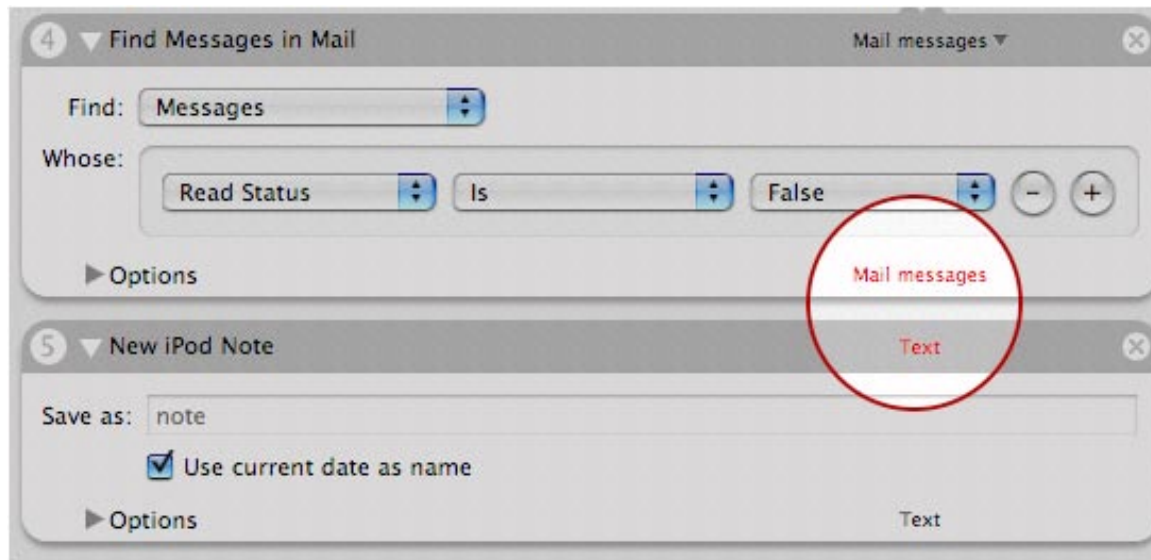


Figure 5.30 Mismatched Action Input and Output Values



Chapter 5:
**Utilizing a
Workflow**

Checking the Log

When testing a workflow by running it in Automator, it is always a good idea to open the *Log* drawer by selecting *Show Log* from the *View* menu. Once the *Log* drawer has been opened, run the workflow and monitor the *Log* drawer for potential problems. The *Log* drawer should display a listing of all events being performed by the actions in the workflow. See figure 5.31.

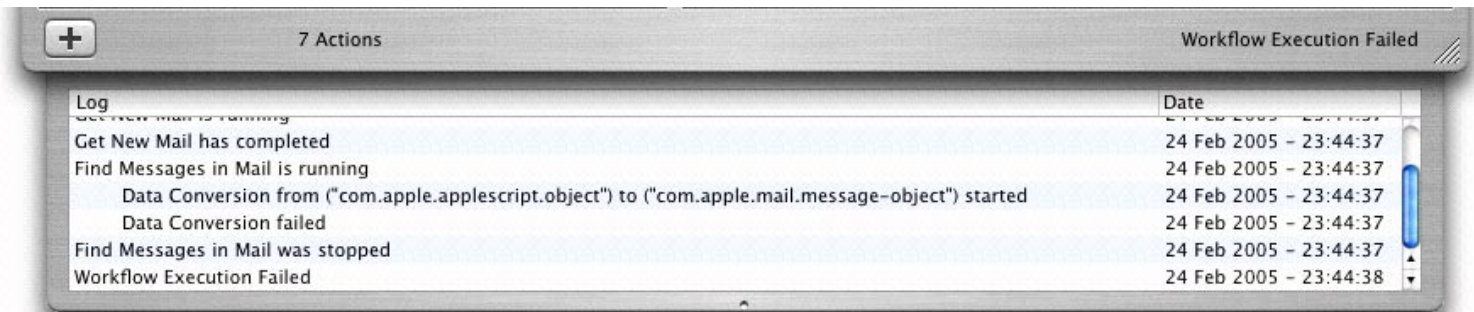


Figure 5.31 Troubleshooting Log

View Results Action

As another troubleshooting technique, Automator includes an action named *View Results*, which can be extremely useful for diagnosing problems in a workflow. This action can be found by clicking on the Automator category icon in the *Library* list in a workflow window. Simply drag this action into your workflow. No further configuration of the action is necessary. When the workflow is run within Automator, the *View Results* action will display the output value of the previous action. See figure 5.32.

For troubleshooting purposes, you may wish to add the *View Results* Action to the workflow after each action, to ensure that the actions are producing the desired results.



Chapter 5: Utilizing a Workflow

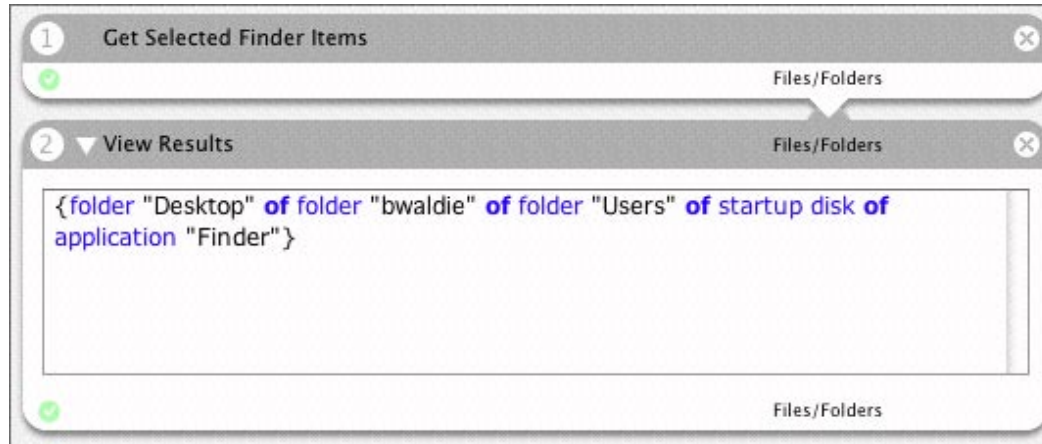


Figure 5.32 The View Results Action

Importing Actions

While the default actions included with Automator are excellent, and may be sufficient for almost everything that you need to do, at some point in the future you may have the need to add third-party actions. To import actions into Automator, select *Import Actions* from the *File* menu, as shown in figure 5.33.

You may also import actions by dragging and dropping them into the *Library* list in a workflow window.

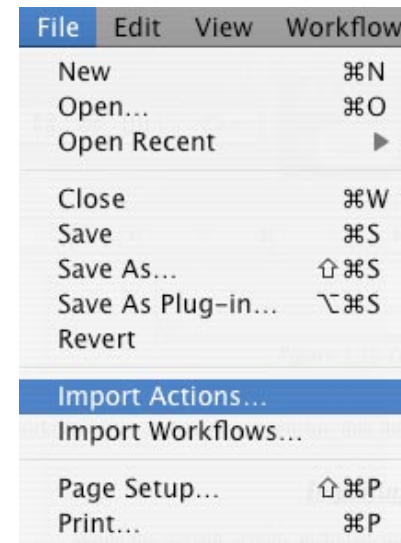


Figure 5.33 Import Actions Menu



Chapter 5: Utilizing a Workflow

Once you import an action, it will be copied into the current user's *Library > Automator* folder, making it available only to the current user. To install an action that can be accessed by all users, manually move it into the *Library > Automator* folder at the root level of your hard drive. Imported actions will immediately appear in the *Action* browser in Automator.

Some third-party actions may come with an installer, which will automatically install the actions for you, thus eliminating the need for you to import the actions.

Importing a Workflow

Automator will also allow you to import saved workflow files into the groups shown in the *Library* list. To import a workflow, select *Import Workflows* from the *File* menu. See figure 5.34. Importing a workflow in this manner will add it to the *My Workflows* group folder in the *Library* list.

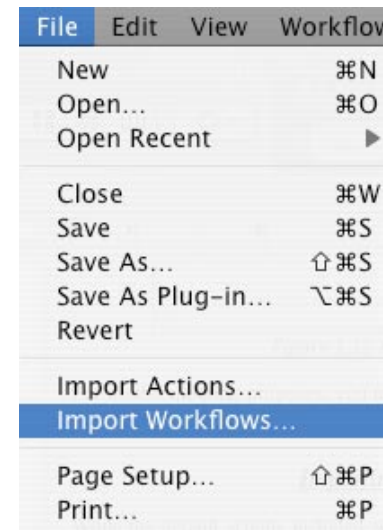


Figure 5.34 *Import Workflows Menu*

You may also import a workflow into a specific group in the *Library* list by dragging and dropping it from the Finder onto the desired group. As previously mentioned, only workflow files, and not workflow applications, may be imported.



Chapter 5:
**Utilizing a
Workflow**

Creating a Workflow from Finder Items

At times, you may wish to create a new workflow that will process specific files or folders. One way to do this is to open Automator, create a workflow, add the *Get Specified Finder Items* action to the workflow, and then add each individual file and folder to the action. This works fine. However, wouldn't it be great if there was an easier way to do this? Well, there is.

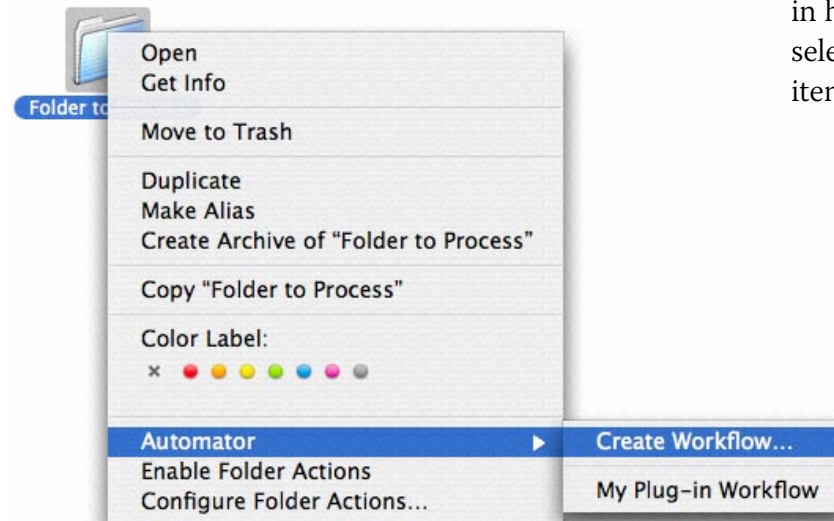


Figure 5.35 *Create Workflow From Finder Contextual Menu Option*

To quickly start a workflow like this, select the files and folders in the Finder that you want to process. Next, hold down the control key, and click on the selected items, displaying the Finder's contextual menu. Scroll

down to the *Automator* sub-menu, and select *Create Workflow*. See figure 5.35.

Automator will launch and automatically create a new workflow for you, complete with the *Get Specified Finder Items* action pre-populated with the list of files and folders you selected in the Finder. See figure 5.36.

Continue to add any additional actions to the workflow to process the specified files and folders.

The *Get Selected Finder Items* action may also come in handy. This action can be used to retrieve a list of selected items in the Finder, and pass the detected items to the next action in the workflow sequence.



Chapter 5:
**Utilizing a
Workflow**

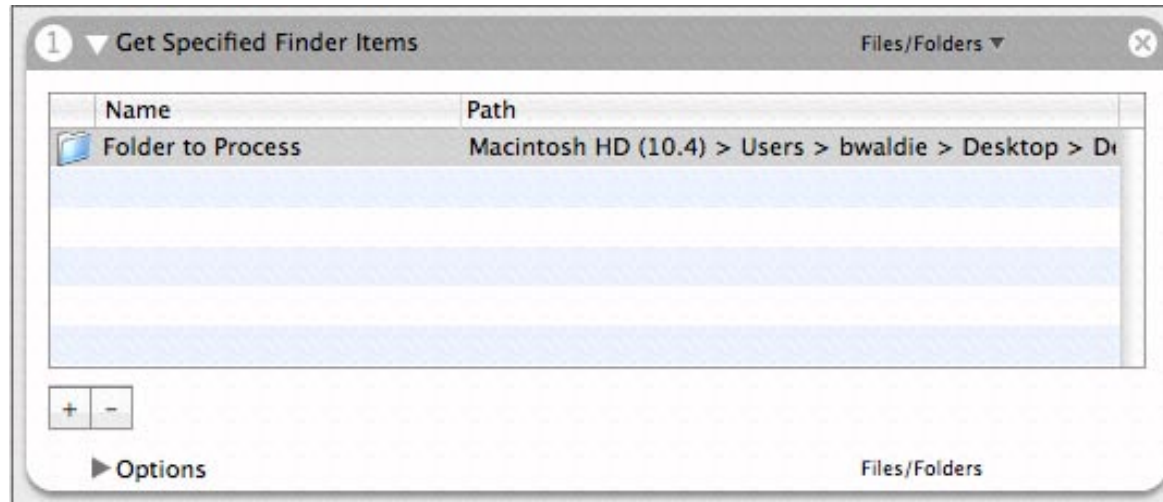


Figure 5.36 Action to Process Finder Items

What's Next

Now that we have explored the individual tasks involved in constructing a workflow, we can start building some. In the next chapter, we will construct two example workflows. By walking through this process, it will give you the opportunity to use what we have already discussed to help you to begin constructing your own Automator workflows.



Chapter 6 Building an Example Workflow

Now that we have discussed Automator's interface, and the primary aspects of constructing a workflow, let's walk through the process of actually building some example workflows. Throughout this chapter, you will see, first hand, the major steps involved in using Automator in order to automate a process.

Workflow Example 1 – Backup Safari Data

Planning the Workflow

The first step in building our workflow is to plan out what we want to do. For the purposes of this example, we are going to create a workflow that backs up our Safari data, including our bookmarks, form values, and history.

Outline the Workflow

Prior to building your workflow, it is always a good idea to outline the desired workflow, either in writing or in your head. This outline should list the tasks, as if they were being performed manually. By outlining the process, it will ensure that you are not leaving out any steps along the way, and that you know the proper order of the tasks to be performed.



Chapter 6: Building an Example Workflow

The following is a rough outline of the tasks involved in this example workflow:

- ▶ Locate a main backup folder on the *Desktop*, creating it if it doesn't already exist
- ▶ Locate the items to back up. In this case, we will back up Safari's bookmarks file, form values file, and history file. These files can be found in the *Library* > *Safari* folder in your user folder.
- ▶ Create a Finder *.zip* archive containing the files to be backed up, and save the archive into the main backup folder
- ▶ Rename the archived files to include the current date in their name.

Building the Workflow

Now that we have a plan, we can begin putting that plan into action by building the workflow.

First, create a new workflow window by either launching Automator, or by selecting *New* from the *File* menu. Next, begin adding actions into the workflow. All of the actions specified in these example workflows are included by default with Automator. If you have trouble locating one of the actions specified below, click on *Applications* in the *Library*, and type the name of the action into the search field in the window's toolbar.

Locating the Backup Folder

The first thing that we want our workflow to do is to locate the main backup folder. To do this, we will use the *New Folder* action. This action will allow you to specify the name and location of the folder you wish to create. The action has been written to attempt to create the folder only if it does not already exist.

Drag and drop the *New Folder* action into your workflow, and the action's settings interface will be displayed. In the *Name* field, specify the folder name *Safari Backups*. Next, select *Desktop* from the *Where* popup menu. See figure 6.1.

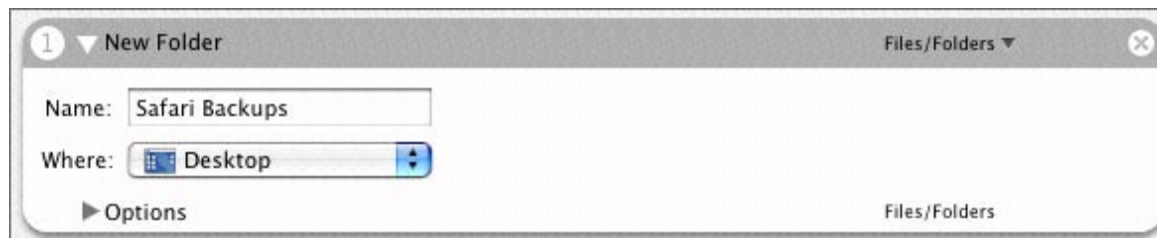


Figure 6.1 Properly Configured New Folder Action



Chapter 6: Building an Example Workflow

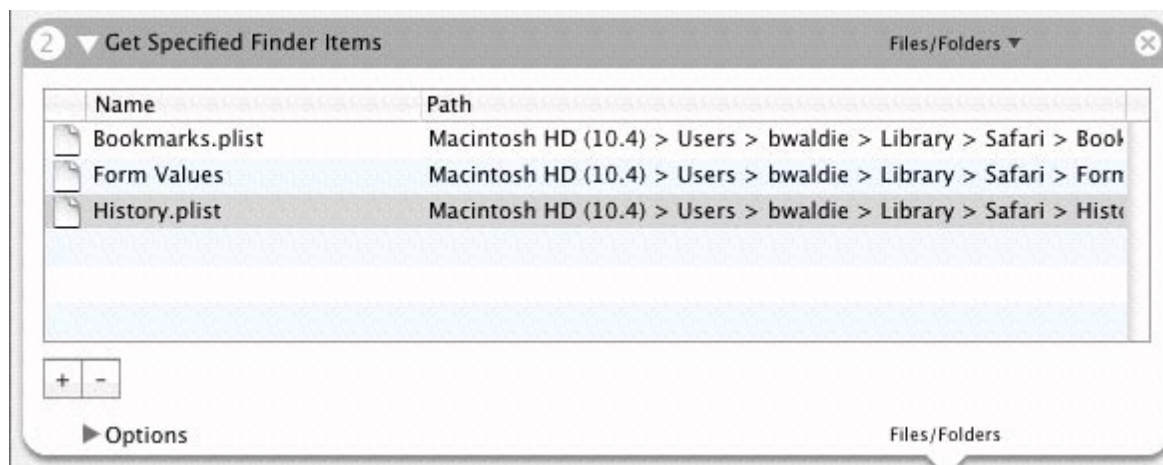


Figure 6.2 Properly Configured *Get Specified Finder Items* Action

Specifying the Items to Back Up

Next, we need to specify the items that we would like to be backed up by our workflow. This will be done by using the *Get Specified Finder Items* action. This action will allow you to specify any number of file paths, to be passed to the next action in the workflow sequence.

Add the *Get Specified Finder Items* action to the end of your workflow, and the action's settings interface will be displayed. This action's settings interface will contain an empty list of Finder items. To add a new item to the list, click the + button under the list field, and navigate to the desired item. Add the following items to the list, all of which should be located in the *Library > Safari* folder in your user folder. See figure 6.2.

- ▶ Bookmarks.plist
- ▶ Form Values
- ▶ History.plist

Because of how it was written, this action will automatically append any input paths it receives to its list of output paths. In our workflow, the previous action creates a new folder, and passes the path to that folder to the *Get Specified Finder Items* action. However, we don't want to back up that folder. We only want to back up the files that we have specified. Therefore, we need to configure the action to ignore the output of the previous action. To do this, select *Ignore Results from Previous Action* from the input value popup at the top of the action. See figure 6.3.



Chapter 6: Building an Example Workflow

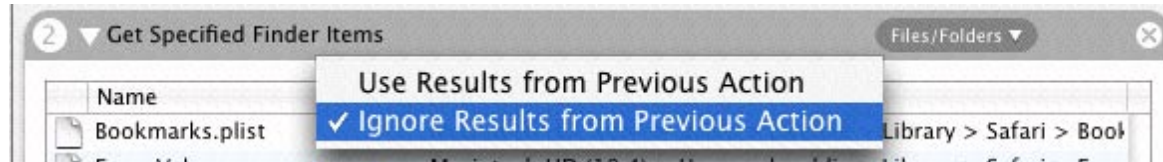


Figure 6.3 Ignore Input Value Configuration



Figure 6.4 Properly Configured Create Archive Action

Creating the Archive

Now that we have a list of files to be backed up, we can create the archive. This will be done using the *Create Archive* action. This action is configured to accept a list of file paths from the previous action. Within this action's settings interface, we may specify an output folder and an archive name.

Add the *Create Archive* action to the end of your workflow, and the action's settings interface will be displayed. Enter *Backup* into the *Save As* field in the action's settings interface. See figure 6.4.

Next, we need to specify an output folder for this action. For configuration purposes, navigate to your Desktop, and manually create a new folder named

Safari Backups. This folder needs to exist when configuring the workflow. However, once the workflow has been configured, then it can be deleted, as the first action in our workflow will automatically create it if it doesn't already exist. Once you have created this folder, then go back into your workflow in Automator, and select *Other* from the *Where* popup in the *Create Archive* action's settings interface. When prompted, navigate to the *Desktop*, and select the newly created *Safari Backups* folder. See figure 6.4.



Chapter 6: Building an Example Workflow



Figure 6.5 Properly Configured Add Date or Time to Finder Item Names Action

Appending the Date to the Archive

The *Create Archive* action will pass as its output the path to the newly created archive file, which will be located in our *Safari Backups* folder. Now, we need to append the date to the name of that archive. To do this, we will use the *Add Date or Time to Finder Item Names* action. This action will allow you to append date, time, or even other information, such as prefixes and suffixes, to the names of files passed from the previous action.

Drag and drop the *Add Date or Time to Finder Item Names* action to the end of your workflow, and the action's settings interface will be displayed. Select *Add Date or Time* from the popup in the upper left of the action, if it is not already selected for you. Next, select *Current* from the *Date/Time* popup, *Month Day Year* from the *Format* popup, and *Before Name* from the *Where* popup. Select *Dash* from the *Separator* popup

located under the *Format* popup, and select *Underscore* from the *Separator* popup located under the *Where* popup. Select the *Use Leading Zeros* checkbox. See figure 6.5.

Testing the Workflow

Now that you have built and configured the workflow, you should test it within Automator. To test the workflow, click the *Run* button in the workflow window's toolbar.

As the workflow runs, monitor the status indicators in each action. Verify that the green checkbox indicator appears as each action completes processing. While the workflow is running, monitor the status display in the bottom right corner of the workflow window. You may also want to open the *Log* drawer and monitor the



Chapter 6: Building an Example Workflow

events being performed by each action.

If the workflow has run successfully, then you should have a folder on your *Desktop*, named *Safari Backups*, and this folder should contain an archived file, containing your Safari data. See figure 6.6.



Figure 6.6 The Safari Backups Folder

Saving the Workflow

Next, we need to save our completed workflow. For this example, we will walk through saving the workflow in two different ways.

Saving the Workflow as an Application

The first way that we will save the workflow is as an application. To save the workflow in this manner, select *Save As* from the *File* menu in Automator. When prompted, enter *Backup Safari Data* as the name of the

workflow, and select a file format of *Application*. Save the workflow to the *Desktop*. See figure 6.7.

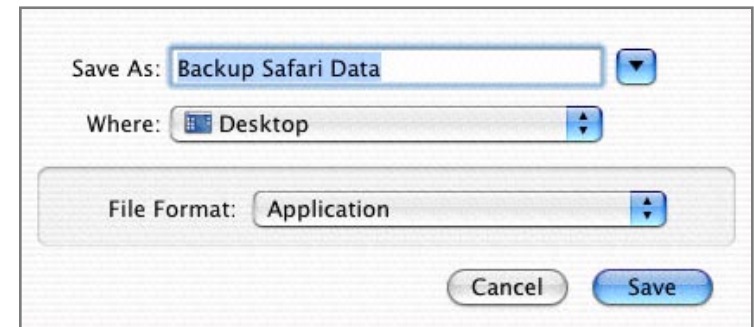


Figure 6.7 Saving a Workflow Application

Now that the workflow has been saved as an application on your *Desktop*, you may trigger the workflow by double clicking on its icon in the Finder. You may also drag the workflow into your Dock, where you may trigger it by clicking on it.

Scheduling the Workflow

Saving this example workflow as an application is fine. However, for this type of a process, it would be useful if we could configure the workflow to trigger at certain times. You may recall that, as we discussed in chapter 5, we can do this by saving the workflow as an *iCal Alarm* plug-in. To save the workflow as an *iCal Alarm*, select *Save As Plug-in* from the *File* menu in Automator. Enter a plug-in name of *Backup Safari Data*, and select *iCal Alarm* from the *Plug-in for* popup. See figure 6.8.



Chapter 6:
**Building an
Example
Workflow**

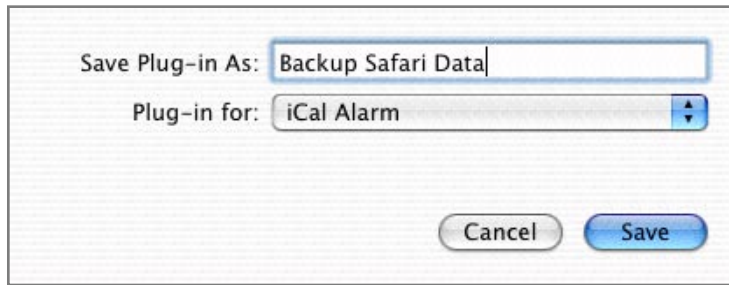


Figure 6.8 Saving an iCal Alarm Plug-in

Next, click the *Save* button. At this time, Automator should automatically launch iCal, and add the workflow to your calendar as a current alarmed event. See figure 6.9.

Now that the event has been created, we want to adjust the schedule of the event, so that it triggers at a specific time each day, such as at midnight. To do so, modify the event's date and time in iCal's event info panel, and specify a repeat period of *Every Day*. See figure 6.10.

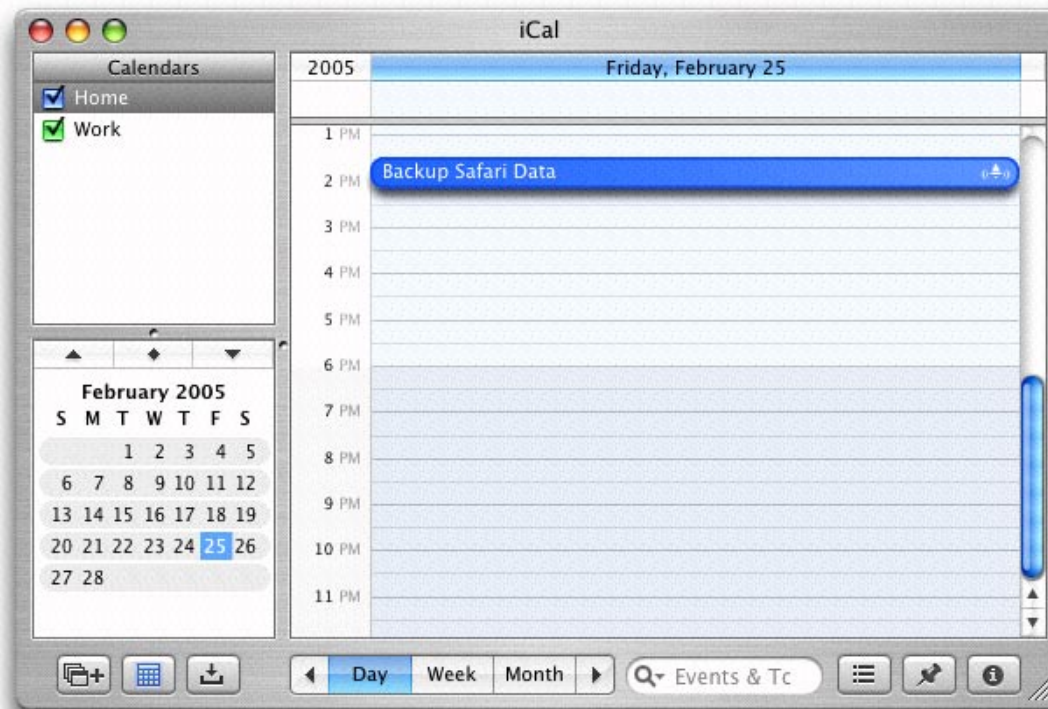


Figure 6.9 iCal Alarm Workflow



Chapter 6:
**Building an
Example
Workflow**

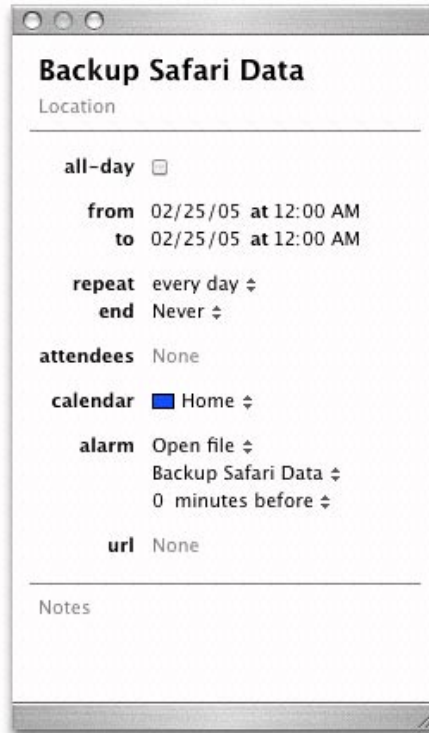


Figure 6.10 iCal Info Panel for an Alarm Workflow

Workflow Example 2 – Email Photo Contact Sheet

Planning the Workflow

This next example workflow is geared toward photographers. For this example, we are going to build a workflow that will trigger whenever we place photographs into a folder. Once triggered, the workflow will construct a contact sheet of the photos, in PDF format, generate a new email message, and attach the contact sheet to the email. This action will make use of the Mail application in Mac OS X. If you have not used Mail in the past, then you will be required to launch and configure it in order to use this example workflow. If you do not want to use Mail as your primary email application, then you can configure it for testing purposes by entering fake account information.

Outline the Workflow

As we discussed when planning our previous example, the first step is to outline the manual process that we want to automate. This manual workflow would consist of the following tasks:

- ▶ Trigger whenever photos are added into a folder
- ▶ Generate a contact sheet of photos in some application
- ▶ Convert that contact sheet to PDF format, and save it to the *Desktop*
- ▶ Create a new email message



Chapter 6:
**Building an
Example
Workflow**

- ▶ Enter a subject for the email message
- ▶ Enter body text for the email message
- ▶ Attach the contact sheet PDF file to the email message
- ▶ Delete the contact sheet PDF from the desktop

Building the Workflow

The first step is to create a new empty workflow window. Do this now. Next, we will begin to add actions into the workflow in order to begin automating the process.

Gathering Files to Process

While this workflow will be configured to process photos that are dropped into a folder, we cannot simulate this behavior for testing purposes within Automator itself. Therefore, we need to specify the photos to process. This will be done using the *Get Selected Finder Items* action, located under the Finder category in the *Library* list. See figure 6.11. Locate this action, and add it to the workflow.



Figure 6.11 *Get Selected Finder Items Action*

Once we are comfortable with the behavior of the workflow, then we will remove this action.

Building the PDF

The first task in the actual workflow will be to create the PDF contact sheet. While this could be done manually in a number of applications, Automator can handle this task with one specific action, called *New PDF Contact Sheet*. This action can be found under the *PDF* category in the *Library* list.

Add the *New PDF Contact Sheet* action to the workflow, and the settings for this action will be displayed for editing. The settings will allow you to specify a name and output folder for the saved contact sheet, a paper size, and the number of columns to appear in the contact sheet. Enter *Photo Samples* in the *Save as* field. Next, set the *Where* popup to save the contact sheet to the *Desktop*, set the *Paper size* popup to *US Letter*, and set the *Columns* popup to 3. See figure 6.12.



Chapter 6: Building an Example Workflow

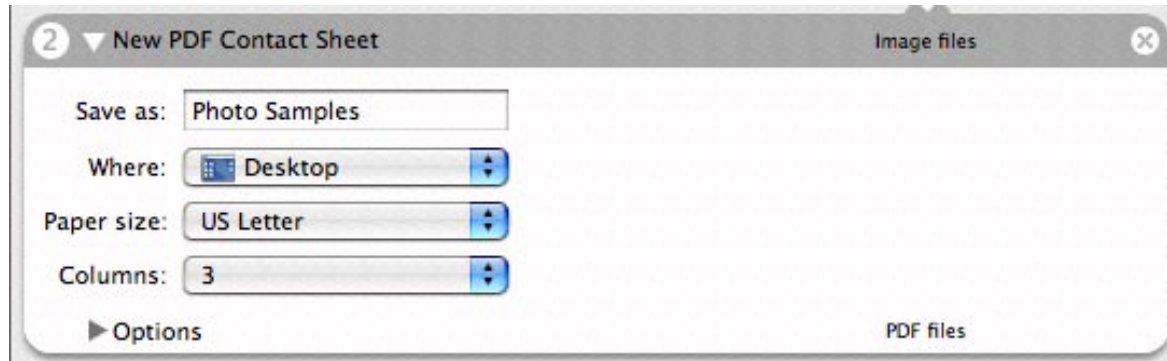


Figure 6.12 Properly Configured New PDF Contact Sheet Action

We now need to actually run this workflow in order to construct an initial PDF contact sheet. We need to do this because some of the actions that we will be adding to the workflow will require the PDF contact sheet to exist during configuration.

Navigate to the Finder, and locate and select one or more photos to process. Next, bring Automator back to the front, and trigger the workflow. If the workflow runs successfully, then a PDF contact sheet of the selected photos, named *Photo Samples*, should exist on your *Desktop*. Leave this PDF file on the *Desktop* for now, as we will need it again shortly.

Creating the Email

The next task in our workflow will be to generate the email message. To do this, we will use the *New Mail Message* action, located under the *Mail* category in the *Library* list.

Add this action to your workflow, and its settings interface will be displayed. Settings that may be modified for this action include the recipients of the message, the subject, body, and account. Fill in a default subject and body for the message. See figure 6.13.



Chapter 6:
**Building an
Example
Workflow**

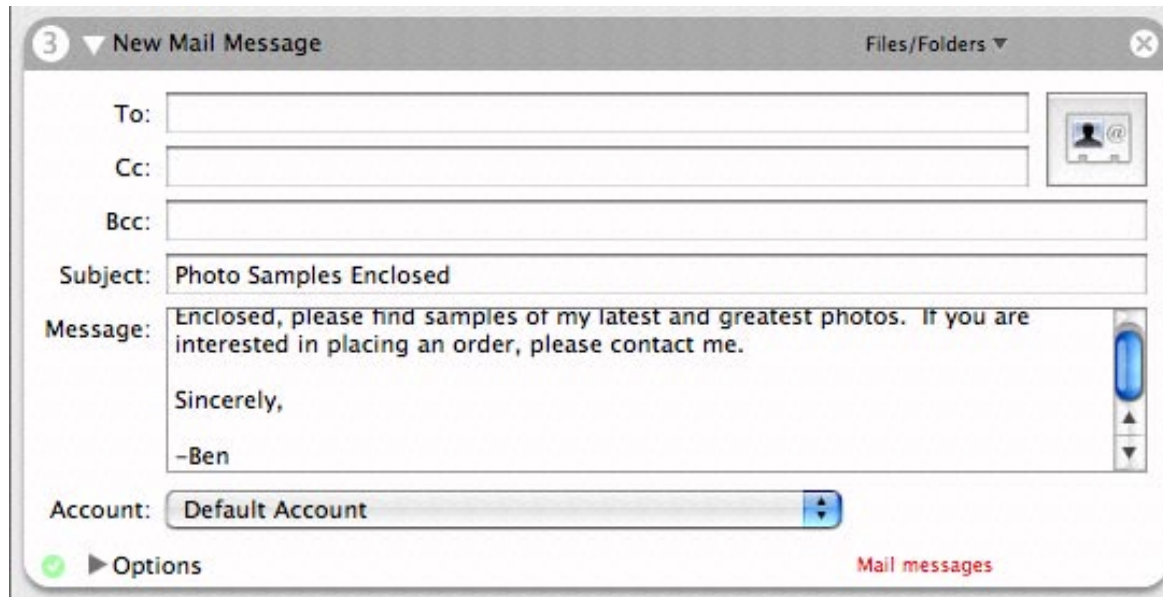


Figure 6.13 Properly Configured New Mail Message Action

Attaching the Contact Sheet

For the next part of the workflow, we will want to attach the contact sheet to the message. This will be done using the *Add Attachments to Front Message* action, located under the Mail category in the *Library* list. If you click on this action, you will see that it accepts file and folder paths as its input. However, our previous action, the *New Mail Message* action, does not output file and folder paths. Instead, it outputs references to Mail messages. This is where that initial PDF contact sheet that we created will come in handy, as we will add an intermediary action here to retrieve the PDF contact sheet's path, and pass it to the *Add Attachments to Front Message* action. For this task, we will use the *Get*

Specified Finder Items action. This action can be found under the Finder category of actions.

Add the *Get Specified Finder Items* action to the workflow. You may recall that we configured this action in our *Backup Safari Data* example, so we won't go over its settings again. Configure the list field of the action to contain the path to the PDF contact sheet that we created earlier. See figure 6.14.



Chapter 6:
**Building an
Example
Workflow**

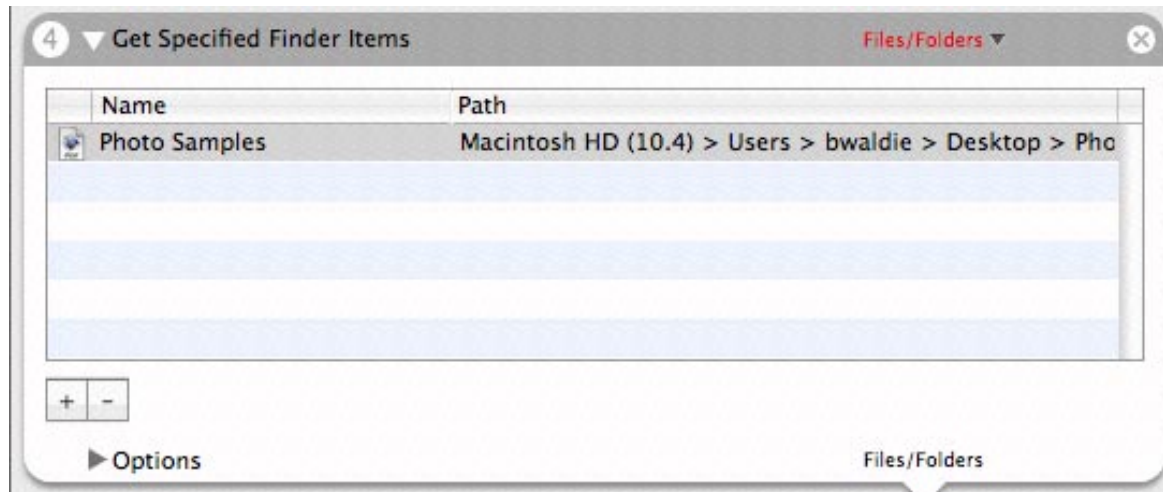


Figure 6.14 Properly Configured *Get Specified Finder Items* Action

Since the previous action in the workflow will return a reference to a Mail message, you may notice that the input value of the *Get Specified Finder Items* action appears in red, indicating two mismatched actions. In this case, it's not a problem, as the action doesn't process the incoming value, and outputs only file and folder paths.

Now, we can add the *Add Attachments to Front Message* action to the workflow. This action does not require any settings to be configured. See figure 6.15.



Figure 6.15 The *Add Attachments to Front Message* Action



Chapter 6:
**Building an
Example
Workflow**

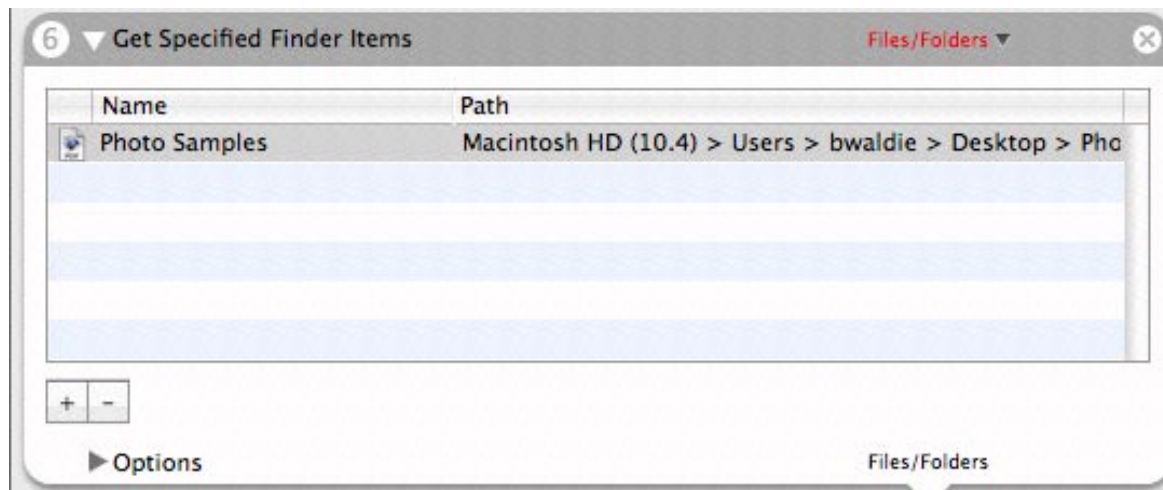


Figure 6.16 Properly Configured Get Specified Finder Items Action



Figure 6.17 The Move to Trash Action

Removing the Contact Sheet PDF From the Desktop

Finally, now that the PDF contact sheet will be attached to the message, we can delete the contact sheet from the *Desktop*. To do this, we will use the *Move to Trash* action, located under the Finder category of actions. However, since this action inputs file and folder paths, and the previous action outputs references to Mail messages, then we must specify again what we want to move to the trash. To do this, we will use the *Get*

Specified Finder Items action again, which will be configured exactly as before. You may add this action again, or you may simply copy and paste the previous instance of the action. See figure 6.16.

Now, we can add the *Move to Trash* action to the end of the workflow. This action does not require any settings to be configured. See figure 6.17.



Chapter 6:
**Building an
Example
Workflow**

Testing the Workflow

As always, you should conduct testing of the workflow before implementing it into a live workflow. Prior to testing this workflow, be sure to delete the *Photo Samples* PDF contact sheet that we created on the *Desktop*.

To test the workflow, select some photos in the Finder, and trigger the workflow from within Automator. If all goes well, then the workflow should generate a new email message, complete with an attached contact sheet of the photos you selected. See figure 6.18.

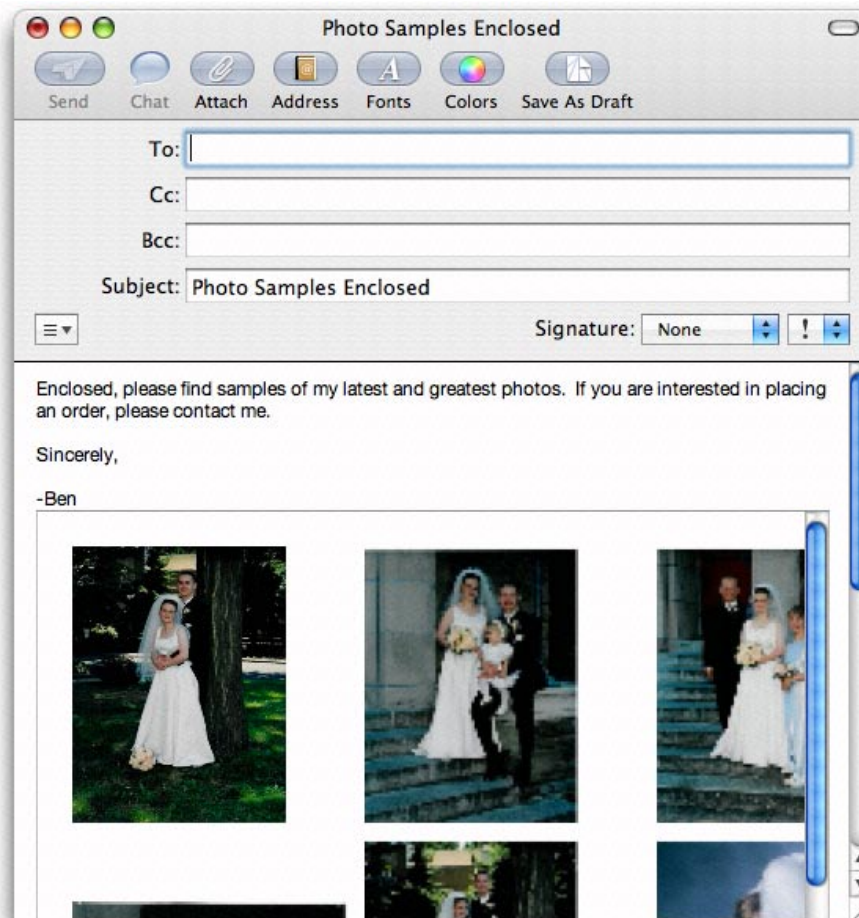


Figure 6.18 An Automator Generated Email



Chapter 6:
**Building an
Example
Workflow**

Saving the Workflow

Finally, if the workflow performs as desired, it is time to save the workflow. For this example, we will save the workflow as a Folder Action plug-in. To do this, first create a folder on your desktop, named *Photos*. Next, bring Automator back to the front. For testing purposes, we had configured our workflow to retrieve a list of selected items in the Finder. Since a Folder Action plug-in will automatically pass the paths of any detected items directly to the workflow, this first action is no longer necessary. Therefore, you may delete the *Get Selected Finder Items* action from the beginning of the workflow.

Next, select *Save as Plug-in* from the *File* menu. Enter a plug-in name of *Email Contact Sheet* and specify a plug-in type of Folder Action. Next, attach the Folder Action to the *Photos* folder that you created on the desktop.

That's all there is to it. To trigger the workflow, drag and drop photos into the *Photos* folder on your desktop.

What's Next

You have just created your first Automator workflows. These example workflows were designed to walk through the process of constructing a workflow from start to finish. Based on these examples, you should be starting to see how easy it is to configure a workflow and begin automating time consuming and repetitive tasks on your machine.

The next chapter serves as the final chapter in the *Using Automator* section of this book. In the next chapter, we will discuss some slightly more advanced ways that you can use Automator, including ways to trigger AppleScripts and Shell Scripts directly from within a workflow.



Chapter 7 Advanced Topics

Until this point, we have only scratched the surface of what is actually possible with Automator. While simple to use, Automator is a complex tool, and it offers a lot of possibilities for those looking for more robust types of automation.

During this chapter, we will discuss some more advanced actions that can be incorporated into your workflows, as well as other features of Automator that can provide you with a variety of ways to enhance and expand your automated workflows.

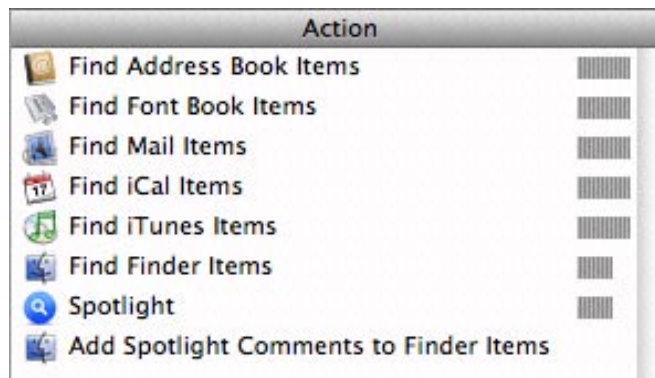


Figure 7.1 *Spotlight Actions*

Working with Spotlight

Automator is just one of many great tools in Mac OS X that can help you to become more efficient in your everyday routines. Other tools, like Spotlight, can also have a dramatic impact on how you use your computer. Because Automator actions can interact directly with the applications and processes on your machine, it can be used to take these other tools a step further.

Spotlight is a unique technology in Mac OS X that can be used to search your hard drive for virtually any type of data. What's more, Spotlight can be integrated with Automator to create some fairly robust workflows.

In the *Library* list in Automator, you will find a Spotlight category icon. Clicking on this category icon will display several actions to get you started with integrating Spotlight searches into your workflows. See figure 7.1.

Spotlight actions can allow your workflows to conduct dynamic searches for files, folders, and more. For example, you might create a workflow that searches your entire computer, using the *Find Finder Items* action, for anything that was modified on the current



Chapter 7:
**Advanced
Topics**

date, and then backs up the found items to a server. See figure 7.2.

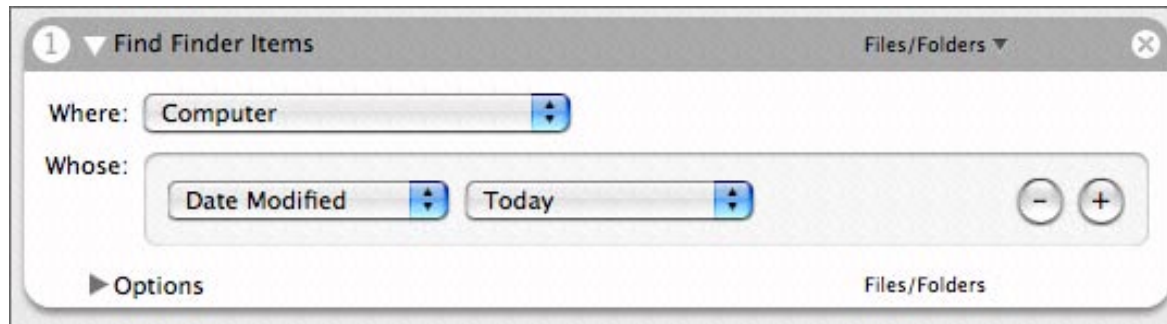


Figure 7.2 Find Finder Items Action

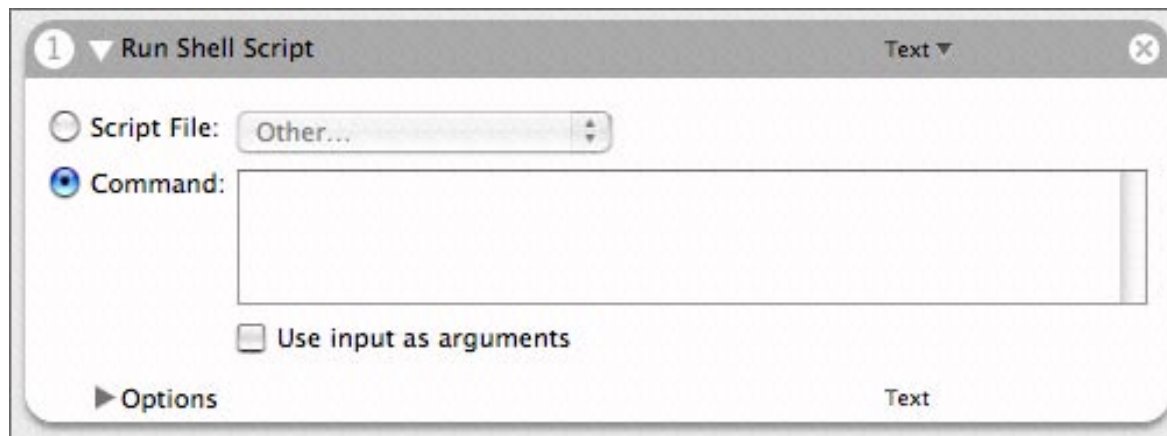


Figure 7.3 The Run Shell Script Action



Chapter 7:
**Advanced
Topics**

Triggering UNIX commands

One of the great things about Mac OS X is that it is built upon UNIX. While the UNIX underpinnings in Mac OS X are hidden from the average user, they are available for you to access, if you want them. Mac OS X offers a command line environment, where experienced users can run command line utilities, shell scripts, and more.

For those who are interested in integrating UNIX with Automator, you are in luck. Included with Automator is a *Run Shell Script* action. See figure 7.3. This action can be found under the Automator category in the *Library* panel. The *Run Shell Script* action will allow your workflow to run shell script files, or to trigger UNIX commands from directly within your workflow.

Working with AppleScript

For those who are AppleScript developers, you are not out of luck either. Automator offers a number of ways that you can utilize AppleScript to interact with Automator.

Triggering AppleScript commands

Like the *Run Shell Script* action, the Automator category of actions also includes an action for triggering *AppleScript* code, called *Run AppleScript*. This action may be added to a workflow in order to perform specific AppleScript-related tasks, such as interacting with applications in ways that Automator actions cannot. For example, you could use the

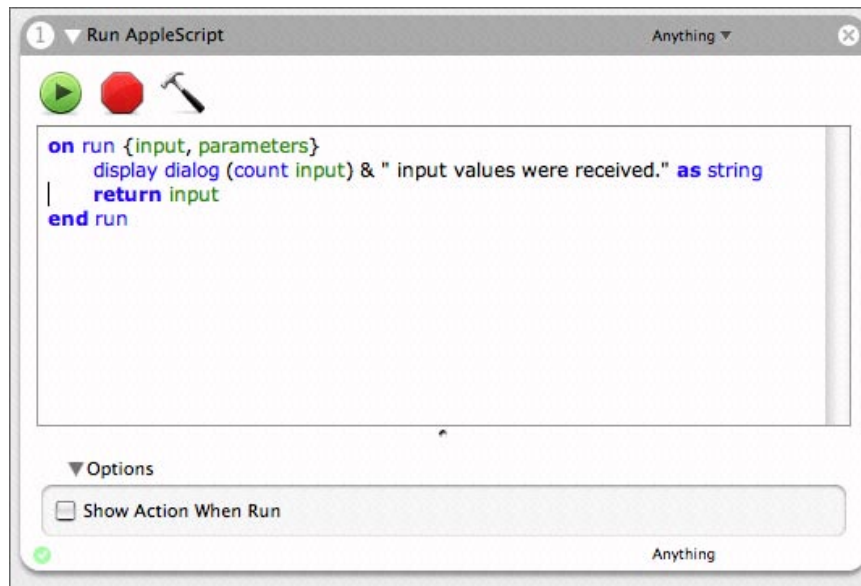


Figure 7.4 The Run AppleScript Action



Chapter 7:
**Advanced
Topics**

Run AppleScript action to use the *Database Events* background application to create a storage database to hold the output of a previous action. Then, later in your workflow, you could use another *Run AppleScript* action to retrieve this data for further processing. See figure 7.4 for an example of the *Run AppleScript* action.

AppleScripting Automator

Automator is also considered a scriptable application, meaning that it can be automated with AppleScript code. Automator contains a full AppleScript dictionary, allowing you to write code that will build and trigger workflows, and more. See figure 7.5.

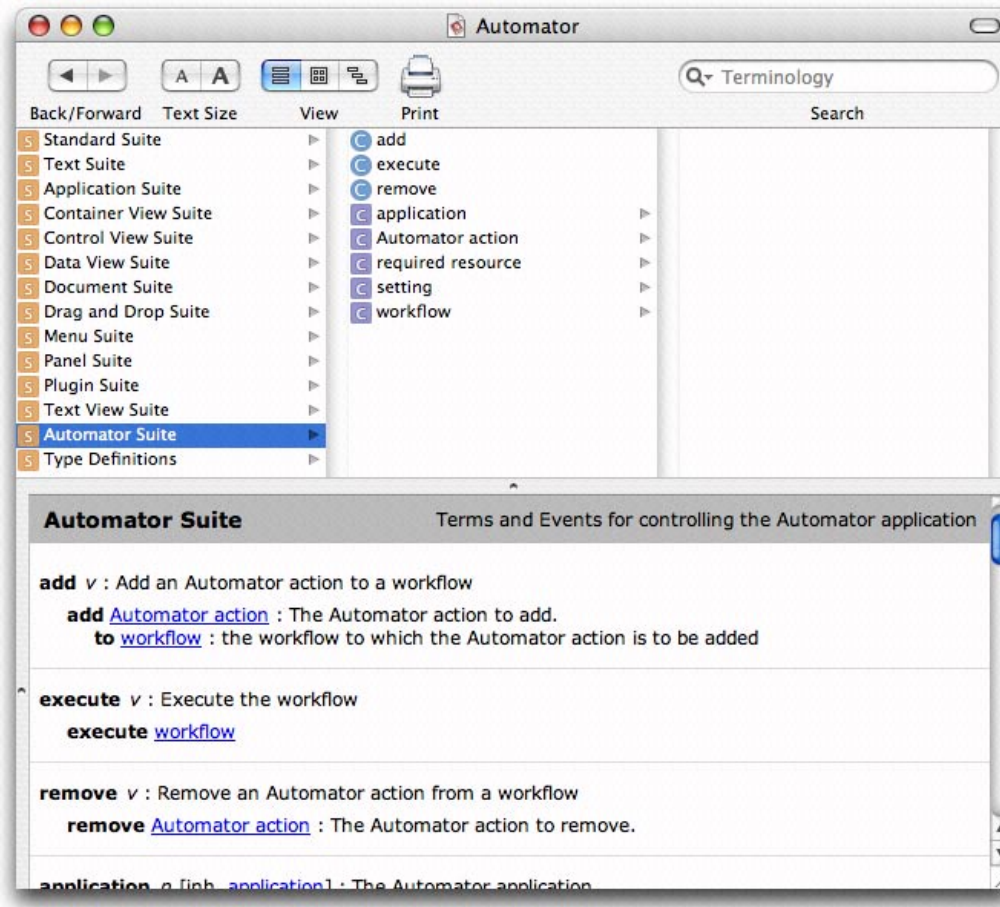


Figure 7.5 Automator's AppleScript Dictionary



Chapter 7:
**Advanced
Topics**

Developer-Related Actions

For developers using Automator, a number of actions are included under the Xcode category in the *Library* list. See figure 7.6. The actions included in this category can allow developers to automate a number of time-consuming tasks typically involved in the development process. For example, the *Build Xcode Project* action can be used to batch build and install specified Xcode projects, while the *Create Package* action can be used to batch create installation packages, using a number of custom default settings.

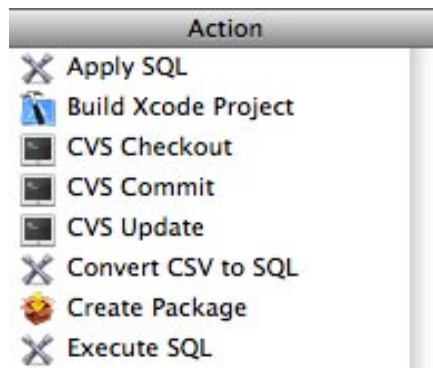


Figure 7.6 Developer-Related Actions

Providing Feedback

As you use Automator, be sure to consider providing feedback. Apple values customer input, so be sure to let Apple know if you have suggestions for new features or options, problems to report, sample actions to request, or anything else to say that's Automator-related. You can provide feedback about any Apple product through Apple's website at

<http://www.apple.com/feedback/>.

For an Automator-specific feedback website, visit <http://www.apple.com/feedback/automator.html>, or select *Provide Feedback* from the *Automator* menu in the menu bar in Automator. See figure 7.7.

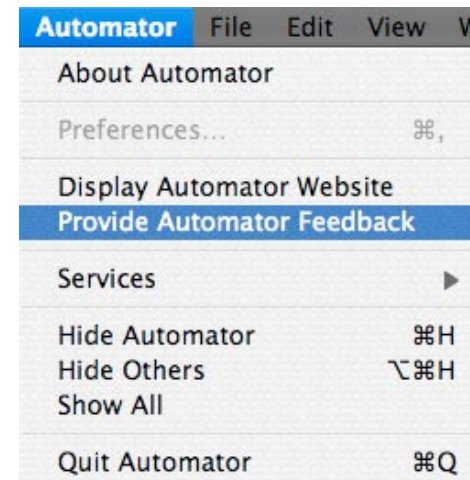


Figure 7.7 Provide Feedback Menu Item



Chapter 7:
**Advanced
Topics**

What's Next

The end of this chapter marks the end of the *Using Automator* section of this book. Throughout this first section, we have learned about using Automator as a tool in order to become more efficient. We have learned about its features and benefits, explored the Automator interface, built and triggered a workflow, and more.

By now, you should be starting to see Automator's amazing potential. You should also be thinking about how it can improve your digital lifestyle by automating those mundane time consuming and repetitive tasks that plague your life on a daily basis. Now, it is time for you to make those thoughts a reality. Don't wait. I encourage you to begin using Automator today.

If you are a developer, or if you want to learn even more about Automator, then stay tuned. In the next section, we are going to start looking at ways that you can develop your own customized Automator actions, using tools that you probably already have installed on your machine.



Section 2

Developing for Automator



Chapter 8 Introduction to Developing for Automator

o far, this book has focused on exploring aspects of the Automator application itself from a user's perspective, including locating actions, building workflows, and more. This chapter begins the developer section of the book. Throughout the remainder of the book, we will explore how you can begin to expand Automator's capabilities by developing your own actions, which can interact with Apple's applications, third-party applications, or the system.

Related Technologies Overview

Developing Automator actions involves a number of technologies. This chapter will provide brief introductions to those technologies before we get started with full-blown development of custom actions. Throughout this chapter, and the remainder of this book, the following technologies will be discussed:

- ▶ Xcode
- ▶ Interface Builder
- ▶ AppleScript
- ▶ Cocoa
- ▶ Objective-C

If you are already familiar with these technologies, feel free to skim this chapter or skip ahead to Chapter 9.

Introduction to Xcode

Xcode is a complete suite of integrated tools, libraries, and interfaces provided to developers by Apple for use in developing Mac OS X-compatible software. Using Xcode, developers can move through the complete process of constructing a software product, from



Chapter 8:
**Introduction
to Developing
for Automator**

concept and design through development, testing, and deployment.

The primary component of the Xcode tools package is the Xcode application, which provides an integrated development environment for constructing customized software products, including applications, system components, and command line tools. Xcode includes a fully featured code editor, a debugger, compilers for a number of languages, and a linker.

The basis for development in Xcode is the **project**, a collection of all the files that, together, come together to build the final product. A project may consist of source files, resources, settings, executables, interfaces, and any other components necessary for the construction of a software product. See figure 8.1.

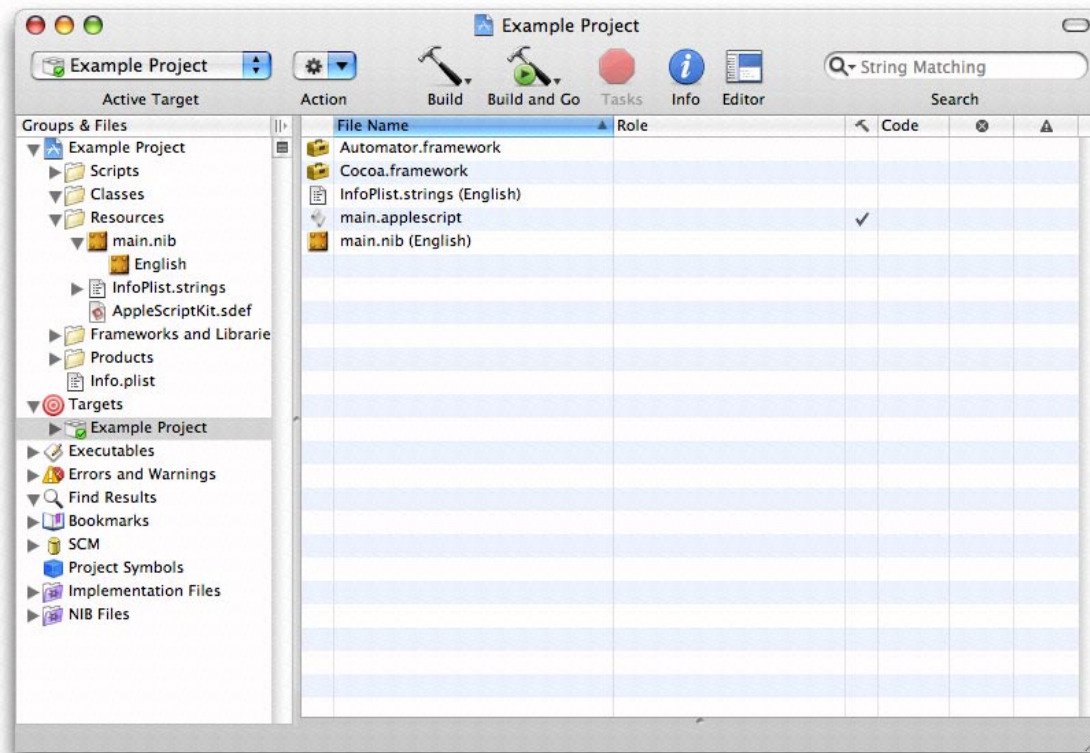


Figure 8.1 Xcode Project Environment



Chapter 8:
**Introduction
to Developing
for Automator**

The source code within an Xcode project may be written in virtually any language supported by the Mac, including AppleScript, C++, Java, and Objective-C. In fact, using Xcode, multiple languages may even be integrated together within a single project, in order to achieve a specialized result. For example, a project could contain Objective-C that interacts with system frameworks, while also containing AppleScript code that interacts with a specific application.

As we proceed through this book, the Xcode application will be used as the basis for development in the construction of custom Automator actions. We will be creating Automator action projects, which will contain a number of components that will be combined together during the build process in order to create a complete action.

Introduction to Interface Builder

Interface Builder is another application that is included in the Xcode tools package, and it is used to design and build custom user interfaces for software products. A project in Xcode may contain any number of interface elements, which may be edited using Interface Builder.

Creating user interfaces with Interface Builder is a relatively straightforward process, and does not require expert programming skills. Within the application, a floating palette contains a large variety of standard Mac OS X interface elements, including buttons, progress bars, text fields, popup buttons, and more. These interface elements may be dragged and dropped into interface windows or menu bars, and moved around as

desired in order to create a custom interface. See figure 8.2. Interface Builder will even assist in the design aspect of an interface by displaying guidelines that provide alignment and placement suggestions when arranging interface elements.

Once a custom interface has been designed in Interface Builder, the interface's elements may be linked back to the code within an Xcode project. By doing this, an interface can be configured to trigger specific code when user actions occur in the interface. For example, an interface might contain a button that is configured to trigger code when clicked. The values of elements in an interface may also be **bound** to code in a project, allowing those values to be accessed directly programmatically.

When developing an Automator action, you will probably want to design a custom interface for the action. This will allow the user to specify settings during configuration of the action, which will affect how the action will perform within a workflow. We will walk through the process of designing an interface for an Automator action in chapter 13.



Chapter 8:
**Introduction
to Developing
for Automator**

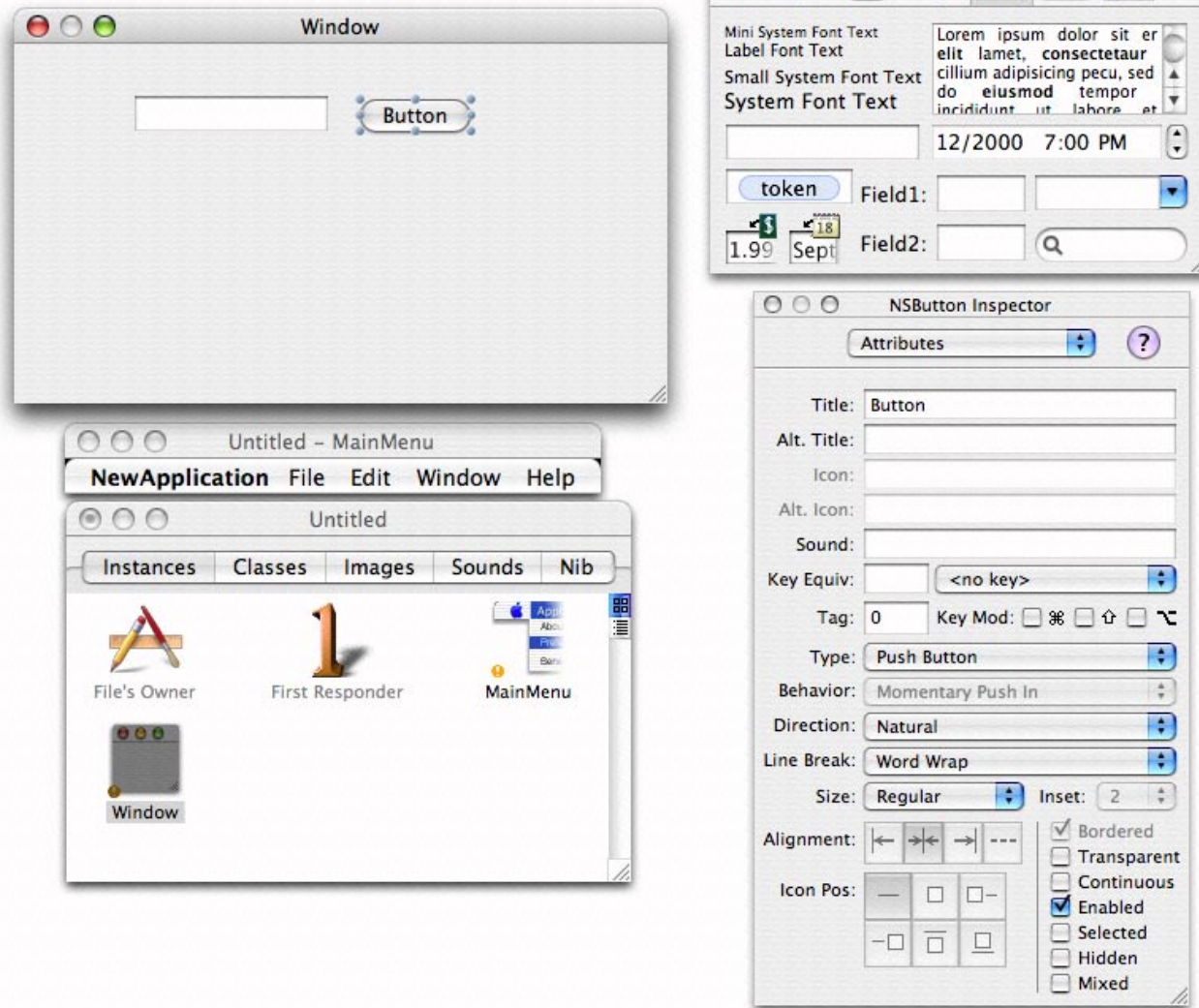


Figure 8.2 Interface Builder Environment



Chapter 8:
**Introduction
to Developing
for Automator**

Introduction to AppleScript

AppleScript is a **scripting language** that is built directly into Mac OS X, and is used to control existing applications, or the Mac OS itself.

In addition, AppleScript's ability to interact with many different applications can allow more advanced users to automate even the most complex and demanding workflows.

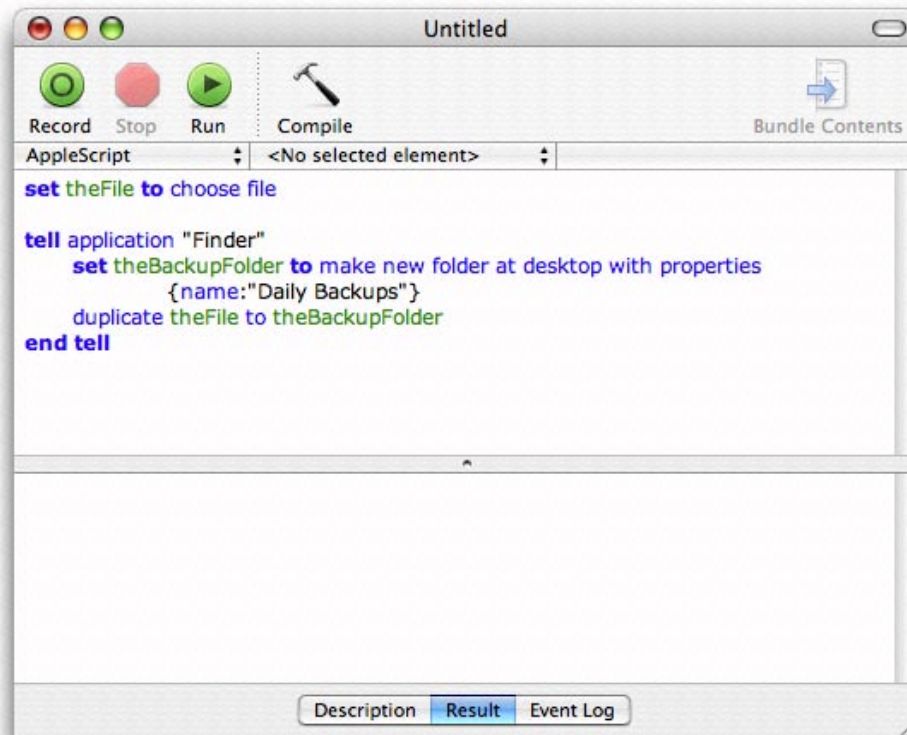


Figure 8.3 A Script Editor Document

By providing the ability to write and run scripts that control existing applications on the computer, AppleScript allows users to automate routine tasks such as backups, image processing, page layout, and more.



Chapter 8:
**Introduction
to Developing
for Automator**

In comparison to other languages, the learning curve for AppleScript is relatively manageable. AppleScript's English-like **syntax** provides even novice users with the ability to examine, navigate, and write simple scripts using an application such as the Script Editor, found in *Applications > AppleScript*. See figure 8.3. This application provides all of the basic functionality needed to create standard AppleScript files. Third party editors are also available, with added features and options for more advanced developers.

To learn more about AppleScript, there's no better place to start than with *Danny Goodman's AppleScript Handbook*, available in print and as an eBook from <http://www.spiderworks.com>.

Scriptable Applications

As previously mentioned, AppleScript is typically used to automate existing applications on a Mac. In order for an application to be automated with AppleScript, it must be **scriptable**, meaning that it possesses AppleScript terminology, and will respond to AppleScript commands. While not every application on the Mac is scriptable, many well-known applications are scriptable. In addition, AppleScript is becoming more and more popular within the Macintosh community, and more scriptable applications are being released on a regular basis. Automator should also provide encouragement for more software developers to make their applications scriptable, as AppleScript may be used as the driving force behind an Automator action that targets a specific application.

If an application is scriptable, it will possess an AppleScript dictionary, which will contain all of the AppleScript terminology that the application understands. To determine if an application is scriptable, you will need to determine if it has an AppleScript dictionary. To do this, select *Open* from the *File* menu in Script Editor, navigate to and select the desired application, and click the *Open* button. If the application is scriptable, its dictionary will be displayed in a new Script Editor window. See figure 8.4.

One thing to pay close attention to with regard to any scriptable application is that with every update to the application, the application's AppleScript terminology may change. This may occur when new features and options are introduced into the application. As you begin automating applications with AppleScript, be sure to test your scripts thoroughly before introducing new application versions, as some code changes may be required. This same rule applies to AppleScript code that interacts with the operating system. While not every software update will require changes to be made to existing AppleScript code, it can happen.



Chapter 8:
**Introduction
to Developing
for Automator**

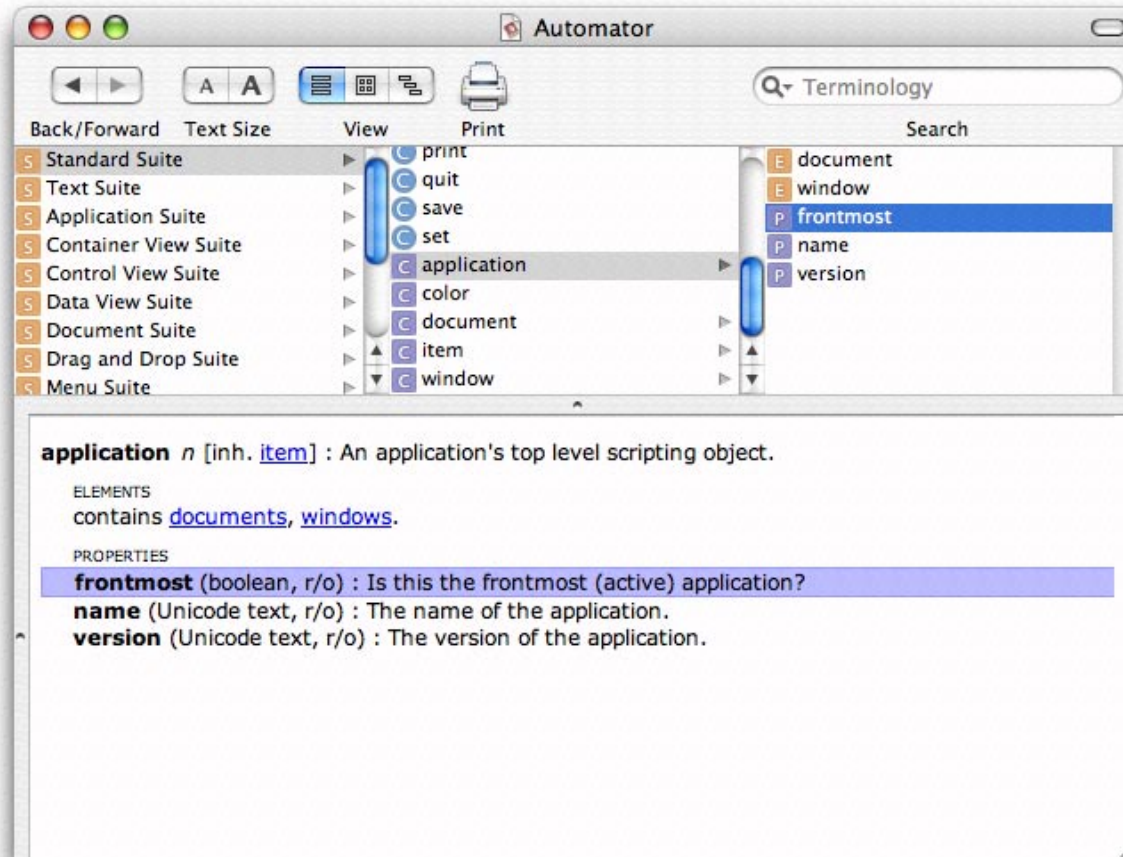


Figure 8.4 Automator's AppleScript Dictionary

Recordable Applications

Some scriptable applications are also **recordable**. This means that, using an AppleScript editor such as the Script Editor, which can be found in the *Applications* > *AppleScript* folder in Mac OS X, you can actually record many manual tasks within the application as

AppleScript code.

Recording can be an excellent way to learn the proper syntax for scripting an application. However, there are some limitations to recording. A recorded script will not contain if/then logic, repeat loops, variables,



Chapter 8:
**Introduction
to Developing
for Automator**

or error handling. Because of this, recording is fine for performing simple, straightforward tasks, but is probably not the best choice for complex automation. Of course, a recorded script may be manually edited after it has been created, if desired, in order to enhance its functionality.

It is also important to note that not every scriptable application is recordable. In fact, unfortunately, precious few scriptable applications are recordable. The way to determine if an application is recordable is to begin recording in your script editor, and then go into the application you want to automate and perform some manual tasks. If AppleScript code is automatically generated in your script editor as you perform these tasks, then the application is recordable. The Finder is a recordable application in Mac OS X.

Attachable Applications

Another level of AppleScript support found in some applications is the ability to trigger AppleScripts from directly within the application itself. An application that provides this ability is considered to be **attachable**.

Some attachable applications allow AppleScripts to be triggered from built-in script palettes or menus. Mac OS X also contains a system-wide script menu, which can allow you to trigger scripts from within virtually any application. As we mentioned in chapter 5, an Automator workflow may be saved as a script menu plug-in, allowing it to be triggered from this menu as well.

AppleScript Studio

AppleScript Studio, contrary to the way the name sounds, is not actually an application itself. Rather, it is a feature set of Xcode and Interface Builder that allows developers to construct fully native Mac OS X Cocoa applications that use AppleScript to interact with the Mac OS or with applications. Using AppleScript Studio, developers can build applications, driven by AppleScript, that use the standard Mac OS X interface.

An AppleScript Studio application's interface can be configured to trigger AppleScript code tied to the application's interface. For example, clicking a button in an AppleScript Studio interface could trigger an AppleScript that performs a specific task.

Later in the book, we'll show you how to use AppleScript Studio to construct AppleScript-based Automator actions.

Additional information about AppleScript Studio, including a complete tutorial and language reference can be found in the *AppleScript Studio Programming Guide* and *AppleScript Studio Terminology Reference* documentation. As we will discuss in chapter 18, these documents are provided by Apple in the *Apple Developer Connection Reference Library*.

Introduction to Cocoa

Cocoa is an integrated suite of **object-oriented** software components, used for running and developing fully featured Mac OS X applications. Cocoa possesses libraries of **classes** and **methods** that may be accessed and reused by developers in order to create feature-rich



Chapter 8:
**Introduction
to Developing
for Automator**

Mac OS X applications.

The classes and methods employed by Cocoa are packaged into multiple frameworks, including two primary frameworks of core classes, the Application Kit framework and the Foundation framework. Additional Cocoa frameworks are also accessible to developers, including the Address Book framework, Core Audio framework, Core Image framework, Core Video framework, Web Kit framework, and more. Using Objective-C and Java, developers can access these classes and methods in order to interact directly with the core technologies included in Mac OS X.

By accessing these frameworks, developers have the ability to rapidly build robust applications, while writing surprisingly little code. While only Objective-C and Java are able to directly interact with the Cocoa environment, these languages may be integrated with other languages, including ANSI C, AppleScript, and C++.

Introduction to Objective-C

Objective-C is an object-oriented programming language, based on the ANSI C language. It serves as a set of extensions to ANSI C, allowing the two languages to be used in conjunction with one another, if desired.

In Mac OS X, most Cocoa frameworks are written in Objective-C. Because of this, the libraries of methods and classes within those frameworks may be freely accessed by developers using Objective-C.

The learning curve for Objective-C is significantly greater than that of AppleScript. However, it is

considered to be a simple programming language, and is relatively easy to learn in comparison to other programming languages.

Additional information about Objective-C, including instruction, tutorials, and language references can be found in the *Objective-C Programming Guide*, the *Introduction to Developing Cocoa Objective-C Applications* document, and the *Framework References*. As we will discuss in chapter 18, these documents are available from Apple via the *Apple Developer Connection Reference Library*.

If you are new to programming, then an excellent place to start is Dave Mark's *Learn C on the Macintosh (Mac OS X Edition)*, followed by *Learn Objective-C (Mac OS X Edition)*, by Mark Dalrymple and Scott Knaster. Both books are available from SpiderWorks, <http://www.spiderworks.com>.



Chapter 8:
**Introduction
to Developing
for Automator**

Types of Automator Actions

Apple's Xcode tools come with everything that you, as a developer, will need in order to build your own custom Automator actions. To help get you started, Apple provides two project templates for action development within Xcode, one for creating an AppleScript-based action and one for creating a Cocoa Objective-C-based action. These action project templates have already been pre-configured with many settings and components, reducing the time that would otherwise be necessary to develop an action. As we proceed through the remainder of this book, we will be working with the action project templates provided by Apple.

If you are planning to develop an action that will interact with scriptable applications in Mac OS X, then you will want to use Apple's AppleScript action project template. If you are planning to develop an action that will interact with core Mac OS X frameworks, such as networking and communications, then you will want to use Apple's Cocoa action project template.

While the action templates provided by Apple are based on AppleScript and Objective-C separately, as with any Xcode project, these templates can be expanded to incorporate any languages supported by Xcode. For example, an AppleScript action could be expanded to interact with Objective-C code, and an Objective-C action could be expanded to interact with AppleScript code. By combining multiple languages, an action could potentially be written to take advantage of virtually any aspect of Mac OS X, from the applications, to the system frameworks.

What You Need to Get Started

In chapter 1, we briefly mentioned what you need to get started with this book, some of which bears repeating. As previously mentioned, the remainder of this book is geared toward developers that already have experience in developing either AppleScript or Objective-C based applications, using the Mac OS X developer tools. Brief introductions to these technologies were provided in this chapter. However, the fundamentals of these technologies will not be covered in this book.

If you are not familiar with the technologies listed in this chapter, then you should consider consulting additional resources for continued learning. Some suggested resources are provided in chapter 18.

What's Next

The purpose of this chapter was to provide some background information about developing for Automator, prior to actually getting started with development. For AppleScript developers, this may mean a slightly better understanding of the concepts of Objective-C and the Cocoa frameworks in Mac OS X. For Objective-C developers, this may mean some greater insight into the possibilities of inter-application communication with the use of AppleScript. In the next chapter, we will begin to discuss actions, and how they are handled within the system and by the Automator application.



Chapter 9 How Actions Work

Before we get started creating an Automator action, it is important to understand how an action works from a developer's perspective. This will help to provide some important background information about why actions are handled by Automator in the way that they are.

What is an Action?

In Mac OS X, an Automator action is known as a **bundle**. Actually, an action is known as a specific type of a bundle, called a **loadable bundle**. We will explore loadable bundles in a moment. First, let's expand on the concept of a bundle.

Bundle Overview

A bundle in Mac OS X is a specific type of directory structure that is used to group together executable code and resources associated with that code. Essentially, a bundle provides a central, organized hierarchy for a common set of code and resources. You might simply think of a bundle as a folder containing code and related resources.

Sometimes, a bundle is saved as a **package**, which means that it appears to a user as a single file, rather than as a directory structure. This method of saving bundles as packages provides benefits for both developers and users alike. It allows the actual code and resources within the bundle to be hidden from the user's view, virtually eliminating the possibility of the



Chapter 9: How Actions Work

user inadvertently doing any damage to the bundle's components. It also eliminates the potential confusion associated with multiple bundle components.

Many times, packaged bundles are saved as applications. By saving a packaged bundle as an application, it allows a user to trigger the bundle's executable code by double clicking on one single file, as they would any other application. All of the bundle's resources and components are still there, but they remain hidden from view. Automator is an example of a packaged bundle that has been saved as an application. To the average user, Automator simply appears as an application. However, if you show its contents by control clicking on it in the Finder, and selecting *Show Package Contents* from the Finder's contextual menu (see figure 9.1), you will see that it actually contains a complete directory structure of resources. See figure 9.2.

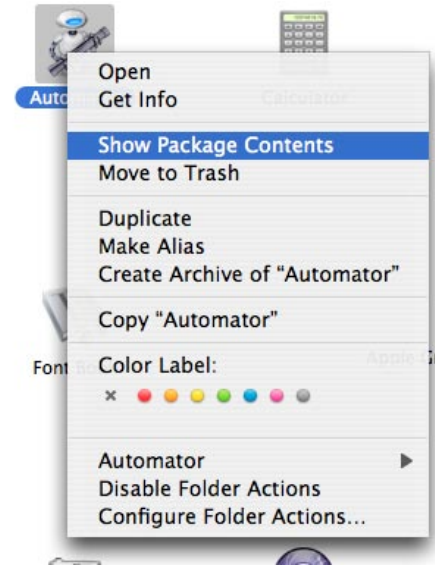


Figure 9.1 Showing Automator's Package Contents



Figure 9.2 Automator's Packaged Bundle Directory Structure



Chapter 9: How Actions Work

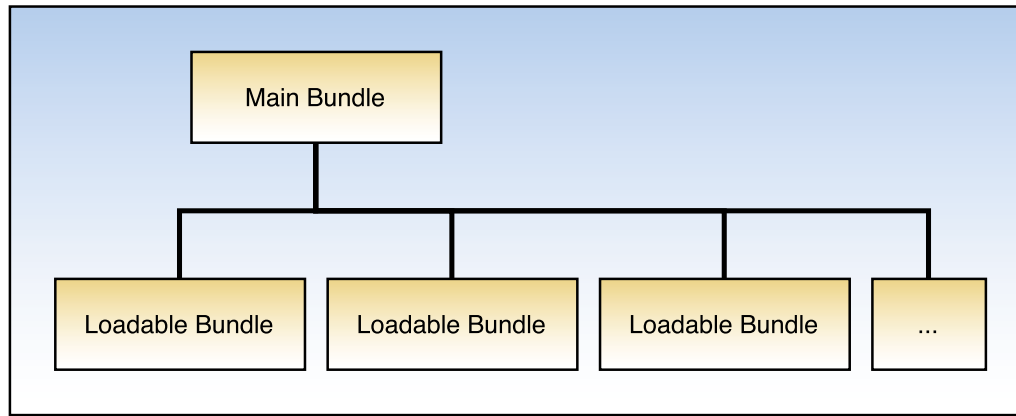


Figure 9.3 Overview of a Loadable Bundle Architecture

Loadable Bundle Overview

Now, on to loadable bundles. A loadable bundle is a specific type of bundle that is configured to be read and loaded by another bundle. See figure 9.3. A loadable bundle may contain executable code, but it may not execute that code on its own. However, once loaded, the main bundle can trigger the executable code located within the loadable bundle. You might think of a loadable bundle kind of like a plug-in for a bundle.

As with bundles saved as packaged applications, there are advantages to loadable bundles. For one, implementing a loadable bundle architecture reduces the amount of code and resources that need to be implemented within a single bundle. Instead, the main bundle only needs to include basic functionality, and new bundles can be plugged in and loaded as needed to add additional functionality. In this type

of architecture, a main bundle can be configured to load additional bundles and execute their code only as needed. Since the main bundle contains less code, it can typically process that code much faster than would otherwise be possible. In addition, compile times would be reduced, as they would be split up among multiple bundles.

How Actions Fit In

As previously mentioned, an action is actually a loadable bundle. Automator is the main bundle, and only includes basic functionality itself. The primary purpose of Automator is to detect and load the bundles of actions, and allow users to organize those actions together to create workflows. It is this type of architecture that allows Automator to be expanded



Chapter 9:
**How Actions
Work**

virtually indefinitely by installing new actions.

Action Handling Within Automator

It is important to note that Automator does not fully load action bundles when it is launched. Instead, it is configured to load action components only when those actions are needed by the user. Once it does load an action, that action will remain loaded within Automator until the application is quit.

When first launched, Automator scans various locations on the computer, and retrieves a list of installed actions. It then extracts the *info.plist* file from each action's bundle, and reads information from the extracted file that is needed in order to display the action in Automator's interface window. This necessary information includes the action's name, category, keywords, description, warnings, required components, etc. Each action also possesses a unique bundle identifier, which is used by Automator to keep track of which action is which. Because of this, when creating your own custom actions, it will be important to ensure that you assign unique bundle identifiers to each of your actions. This will be covered in chapter 12.

At the same time that information about the action is extracted, Automator will also load any Mach-O code found in Objective-C-based actions, and read the action's main *nib* file. In AppleScript-based actions, the *nib* file is not actually read until the action is dragged into a new workflow by a user for the first time.

For those that are new to Cocoa development, Mach-O (Mach object) is the native executable format for binary code in Mac OS X. Also, as a reminder, a *nib* file is the file that Interface Builder uses to store a collection of interface elements built for a specific project.

Action Handling in a New Workflow

When a user begins creating a workflow, the user selects the desired action in Automator's interface, and drags it into a workflow view. At this time, Automator accesses the bundle of the added action, and the view (NSView) instance from the action's main *nib* file is displayed. If a view instance is not present in the *nib*, then the action will still appear in the workflow view, though without an area for modifiable settings. Instead, it will only contain a title bar, and a footer, showing the action's name, and input and output values.

Once an action has been added to a workflow view, the user may modify the settings of the action as desired, add additional actions to the workflow view, rearrange the actions, etc. The user may also run the workflow from within Automator. When this is done, Automator triggers the main processing code in the action, passing it any input from the previous action in the workflow, if present. Additionally, any settings specified by the user in the action's interface are passed to the main processing code of the action as parameters. Once processing is complete, Automator then passes the output of the action to the next action in the workflow sequence.



Chapter 9:
**How Actions
Work**

Action Handling in a Saved Workflow

When a workflow is saved, either as a workflow file or as an application, Automator retrieves the settings specified by the user for any actions included in the workflow. These settings, along with information identifying the actions, are archived into the workflow file. When the workflow file is opened again in Automator, or triggered by the user, the archived action information is retrieved from the workflow, and the actions are located, loaded, and displayed or run.

Because the actual action files are not archived within the saved workflow, any actions utilized in the workflow must be installed on the machine running the workflow. So, if you create a workflow and deliver it to someone else, then that person's computer must have any actions utilized within the workflow installed on their computer.

NOTE: If you install a new version of an action into your system, then you should delete and reapply that action in any saved workflows. This is due to the fact that certain information pertaining to the action that may have been archived within the workflow may no longer be valid.

Threading

When developing for Automator, it is also important to understand that Automator implements a threading architecture when a workflow is triggered. When a workflow begins executing, it does so on a secondary thread. As it begins to process through the actions in the workflow, it analyzes each action, and determines how it should be handled. Objective-C-based actions continue to be run on the secondary thread. However, AppleScript-based actions are run on a main thread instead. By triggering AppleScript actions in this manner, a workflow can allow an AppleScript-based action to interact with external resources, such as scripting additions and scriptable applications. In addition, this method also allows a user to stop or cancel an AppleScript action during processing.



Chapter 9:
**How Actions
Work**

Where Actions are Stored

As mentioned, when Automator is launched, it scans various locations in Mac OS X for actions to be listed in its interface. During this scan, the following primary folders are checked for actions:

- ▶ *System > Library > Automator* – This folder is reserved for Automator actions provided by Apple. Users and developers should not install third-party actions into this folder. Actions in this folder are available to all users on the computer. See figure 9.4.
- ▶ *Library > Automator* – Users and developers may install non-Apple actions into this folder. Any actions that are placed into this folder will be available to all users on the computer.
- ▶ *Users > UserName > Library > Automator* – Users and developers may install non-Apple actions into this folder. Any actions placed into this folder will be available to only the current user.

In addition to scanning the folders specified above, Automator will also scan through the bundles of any registered applications, searching for actions. The Finder automatically registers applications within Mac OS X as it detects them. Within registered application bundles, developers may choose to include actions within a *Contents > Library > Automator* directory. If Automator detects actions in this folder within a registered application, then those actions will be displayed in Automator's interface.

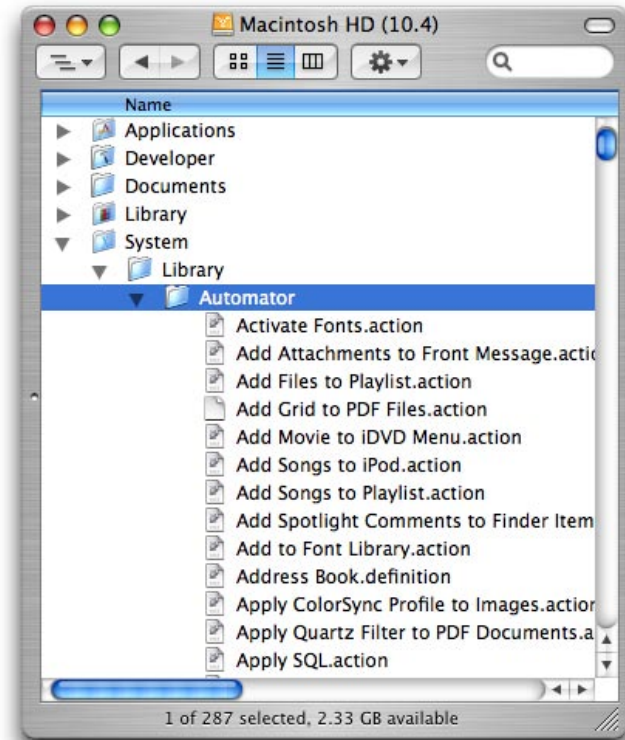


Figure 9.4 *The Location of Apple Actions*



Chapter 9:

How Actions Work

What's Next

Now that we have talked a little about actions, and how they are handled within Automator and the system, we can now begin to discuss the creation of an action.

In the next chapter, we will discuss the first stage in the action creation process, planning the action.

Planning should always be the first step in developing any software product, and an Automator action is no different. Once we have discussed planning actions, then we will begin discussing the process of building actions in Xcode.



Chapter 10 Planning an Action

As with the development of any software product, the first step in developing an Automator action should be to create a plan. Planning should always be an essential part of any software development process, so developing an Automator action should be no different. Taking the time to conduct a planning phase prior to beginning development on a project will help to ensure that development can progress smoothly.

An organized documented project plan can become a tremendous asset when developing any software product. For one, it will allow you to step back and take a look at the finished product as a whole, before actually writing any code. This can help you to determine both major and minor potential problem areas, which may require adjusting to the overall plan. Take my word for it. It is far easier to make adjustments to a software product during a planning phase, than to do so once development is complete.

Another benefit of creating a project plan is that the project plan itself can become the basis for your software development efforts, serving as an outline of the desired functionality. Your project and code can then be built to conform to this outline, helping you to develop your project in a more efficient and organized manner.

Throughout this chapter, we will explore the primary areas for consideration when planning an Automator action project.



Chapter 10:
**Planning
an Action**

Action Functionality

When preparing to develop an Automator action, you should begin by determining the primary functionality of the action. The first rule here is to *keep it simple!* Your action should not be written to perform an extremely complex task. Rather, your action should be kept as generic as possible. Its purpose should be to perform a single specific task. This is important.

By creating actions that adhere to this rule, you will ensure that users will have the greatest flexibility when using the action in their own unique workflows. A generic action should be able to be easily adapted and integrated into a variety of workflow scenarios, some of which you may not even be considering yet.

To gain a better understanding of what might constitute a flexible action, let's take a look at some of the standard actions that Apple provides with Automator.

- ▶ *Get Selected Finder Items* – This action retrieves a list of selected items in the Finder, whether files or folders. The list of retrieved Finder items is then returned as the action's result.
- ▶ *Open Finder Items* – This action opens a passed set of Finder item paths in either their default applications, or in an application specified via the action's settings. This action accepts a list of Finder paths as input, and outputs the same paths as its result.

- ▶ *New Mail Message* – This action creates a new outgoing email message in Mail. Using the action's settings interface, the user may pre-configure recipients, a subject, and a body for the message. This action will accept as its input a list of Finder item paths. If item paths are passed to the action, then they will be attached to the message. The result of this action is a reference to the new outgoing message in Mail.
- ▶ *Add Attachments to Front Message* – This action adds a passed set of Finder item paths to the front outgoing message in Mail. This action accepts as its input a set of Finder item paths, and generates as its result a reference to the front outgoing message in Mail.

Looking at these actions provided by Apple, you can see that each action is responsible for a single primary task, such as retrieving a list of Finder items, generating an outgoing email message, or adding attachments to an opened email message. For additional examples of this concept, you should explore some of the other actions provided by Apple that are included with Automator.

As you begin to develop your own actions, be sure to adhere to this rule, as it will be a benefit to anyone who will use the action.



Chapter 10:
**Planning
an Action**

Action Input and Output

Another step in the planning process of an Automator action is to determine the type of input that the action will handle, as well as the type of output result that the action will generate. This consideration should be primarily based on the type of processing that the action will perform. Although, you should also try to anticipate the types of workflow environments in which users will run the action. Ask yourself what other actions might be included in a workflow prior to your action? What actions might be included after your action? These types of questions, in addition to helping you determine the proper types of input and output values for the action, may also help you to determine additional actions to create.

When anticipating the input value of an action, you should plan to accept as input a list, or array, of values. Since most actions will generate a list of values as their result, following this rule will ensure that your action handles the type of data provided as input. You should also plan for your action to process through the entire list of input values.

If your action will not require that any input be provided, and it will not generate a specific type of output, then you should configure the action to output its input value as its result. This way, the data can be moved on to another action for continued processing.

Action Settings

When planning an action, don't hard code everything that the action will do. Consider making certain aspects of the action's behavior modifiable by the user, if it is possible to do so. Based on settings specified by the user, an action may be able to take different courses of action. Making aspects of the action's behavior modifiable will provide users with greater flexibility in using the action within a variety of workflow situations. If you are developing commercial actions, this will add greater value to your actions, making them more appealing to potential buyers.



Chapter 10:
**Planning
an Action**

Action Naming

Determining an appropriate name for your action is also an important part of the process of planning an action. Automator will display hundreds of actions in a list within its interface. Therefore, when browsing through these actions, the user will see the names for the actions, and look for one appropriate for the type of task that they want to perform.

When you determine a name for your action, try to select a name that is as descriptive as possible. If your action's name accurately describes the functionality of the action, then it will stand out among the other actions, and will be easily located within Automator.

Since an action's name should be descriptive, do not worry too much about the length of the name. If it is necessary to use a lengthy name in order to accurately describe the functionality of your action, go right ahead. If your action will process multiple types of data, also be sure to pluralize your action. This will help to eliminate potential confusion by users. For example, the action named *Add Attachments to Front Message* clearly states that it will allow the user to add multiple attachments to the front email message.

As another rule, when determining a name for your action, do not select the same name as another action that is listed under the same application category. Since you are probably not reinventing the wheel by building a duplicate of an existing action, your action will be unique in one way or another. So, select a unique name that accurately reflects the functionality of your custom action.

What's Next

Again, the planning process is an important one when developing any software product. Be sure, as you begin to develop your own Automator actions, that you take the time to plan out the action's functionality. If you do so, then the development of your action will become that much easier. In addition, the result of your development efforts will be a more polished, more reliable action that performs exactly as expected.

Once you have planned your custom action, it is time to begin development on the action project itself. In the next chapter, we will discuss creating an Automator action project within Xcode. We will then move on, with several chapters outlining the tasks involved in bringing the completed action to fruition. As you begin to put these tasks into practice, the time you have put into planning your action will become a key aspect of, and a driving force behind development itself.



Chapter 11 Building an Action Project

nce you have planned out the various aspects of your custom action, it is time to begin the development process. The process of developing an Automator action is multi-tiered. In this chapter, we will discuss the initial stage of action development, creating an action project in Xcode. In the following chapters, we will move to configuring the action's properties, as well as interface development and other tasks involved in the action construction process.

Creating the Project

To begin development on an Automator action project, launch Xcode, and select *New Project* from the *File* menu. You will then be presented with the *New Project Assistant* dialog window. See figure 11.1.

Selecting a Template

In the initial *New Project Assistant* window, your first task will be to select the type of project template upon which you will build your action. Apple has included two pre-built templates for creating Automator action projects within Xcode, one for constructing an AppleScript-based action, and one for a Cocoa Objective-C-based action. Within the *New Project Assistant* window, templates are grouped into categories. Automator actions can be found under the category labeled *Action*.

These templates provided by Apple are designed to provide you with a core project structure that includes everything you need for the construction of your custom Automator action. Of course, like any project in Xcode, you may choose to expand upon the initial



Chapter 11:
**Building
an Action
Project**

template structure, by adding resources and integrating code from other languages. If you plan to create an action that is a hybrid project, consisting of multiple languages, then it is recommended that you get started by selecting one of the action templates provided by Apple, and then expand the project by adding additional code as needed.

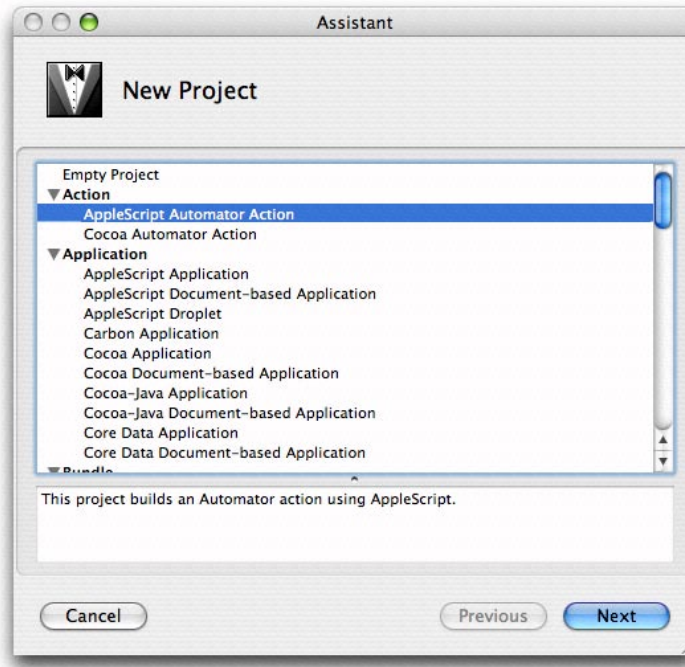


Figure 11.1 The Initial Xcode New Project Assistant Window

Once you have selected an action template, click the *Next* button in the *New Project Assistant* dialog window to continue with the project creation process.

Specifying an Action Name and Directory

As the next step in building your action, you will be prompted to specify a name for the new project. See figure 11.2. This is where some of the planning that we discussed will come in handy. Be sure to take into consideration the suggestions made in chapter 10 regarding action naming.

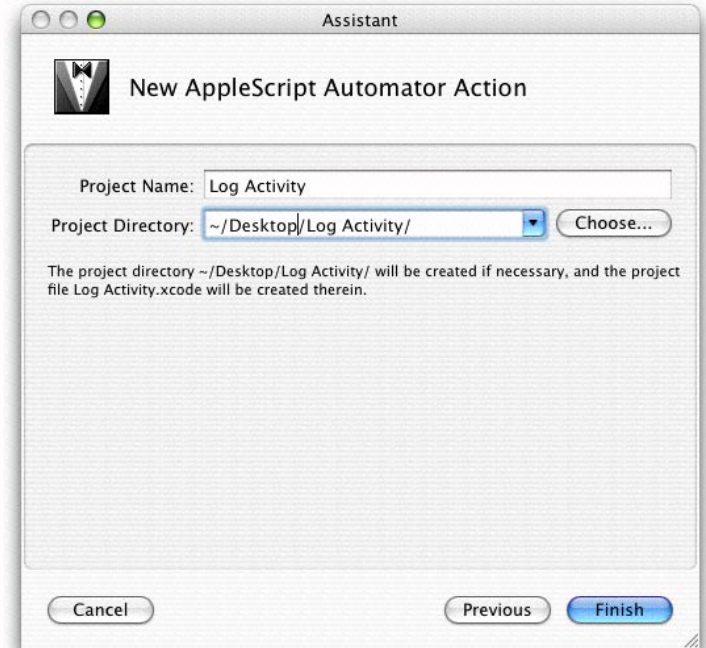


Figure 11.2 The Xcode New Project Window

The naming dialog will also allow you to specify an output directory for your project. Specify this information by typing a directory path into the *Project Directory* field, or click the *Choose* button to navigate to



Chapter 11: Building an Action Project

and select a directory to be automatically populated.

Once you have specified the name and directory for your action project, click the *Finish* button, and the action project will be created, and displayed in a new Xcode window. See figure 11.3.

Action Template Components

A new Automator action project will consist of a number of different components, most of which can be found in both AppleScript and Cocoa action projects alike. In addition to these standard components, you may add other components, if necessary. For example, you may want to include document templates, images, or other types of components within your action bundle.

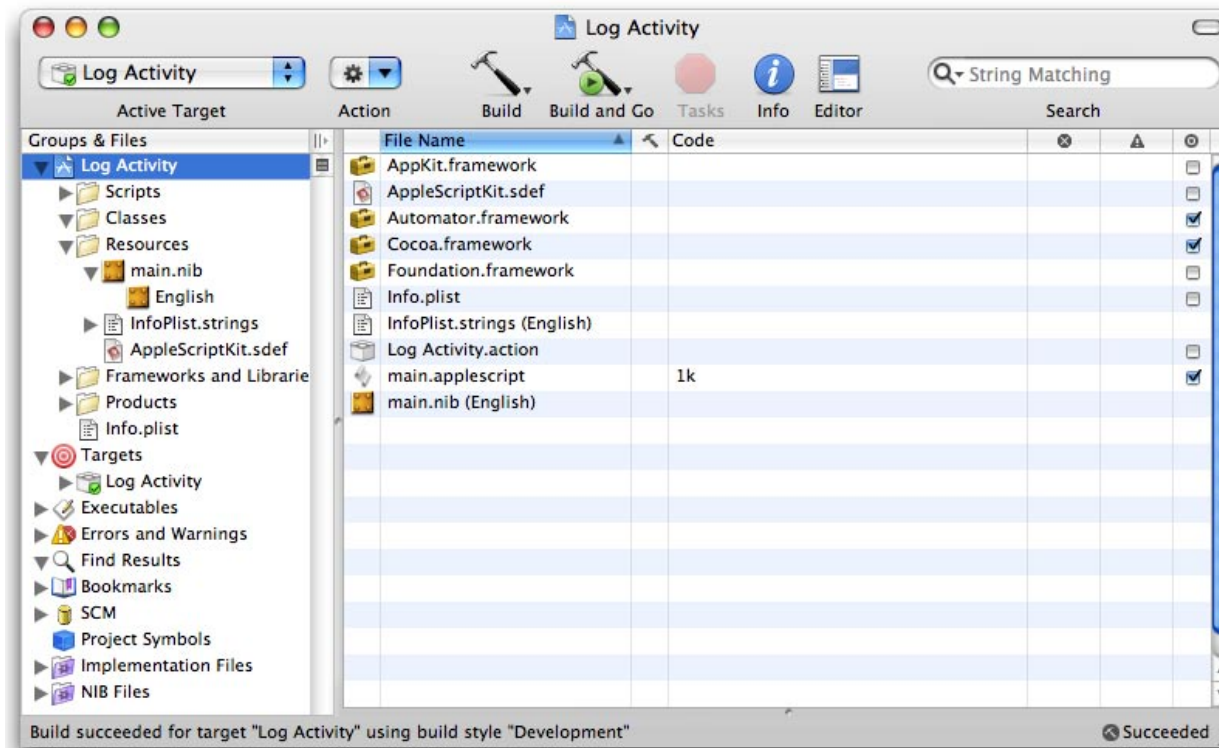


Figure 11.3 A New AppleScript-Based Action Project



Chapter 11:
**Building
an Action
Project**

Common Action Template Components

All Automator actions that have been built on one of the templates provided by Apple will include the following common components:

- ▶ *Frameworks* – By default, a new Automator action project will already be configured with several frameworks, including the Cocoa framework, the Application Kit and Foundation frameworks, and the Automator framework.
- ▶ *Information Property List File* – An information property list file, or *info.plist* file, is included in the project. This file contains information about the action that is necessary in order for the action to be loaded within Automator and run in a workflow. Information specified within this file includes the action's input and output requirements, description text, icon file, and unique bundle identifier.
- ▶ *Localized Strings* – An English version of an *InfoPlist.strings* file is included in the project. This file contains localized versions of certain strings contained within the *info.plist* file. Additional versions of this file may be created in other languages as needed.
- ▶ *Main Interface Nib File* – A main Interface Builder file, or *main.nib* file, is included in the action project. This file is pre-configured with an initial view (NSView) instance for the action, as well as several settings.

AppleScript Action Specific Components

In addition to the components included in every Automator action project, an AppleScript action project will also include the following additional components:

- ▶ *Scripting Definition File* – Like other AppleScript Studio projects, an AppleScript-based Automator action project will include an *AppleScriptKit.sdef* file. This file contains the AppleScript Studio AppleScript terminology necessary for the project to compile and run.
- ▶ *Main AppleScript File* – A single AppleScript file, named *main.applescript*, is included by default in a new AppleScript-based Automator action. This file has been pre-configured to contain an **on run** handler, which will be used for executing the action code, when triggered.

Cocoa Action Specific Components

In addition to the components included in every Automator action project, a Cocoa action project will also include the following additional components:

- ▶ *Objective-C Template Files* – A Cocoa action project will include, by default, *projectName.h*, *projectName.m*, and *projectName_Prefix.pch* Objective-C template files. These files have been pre-configured with default code to be triggered when the action is run.



Chapter 11:

Building an Action Project

What's Next

Once you have built your initial Automator action project, you will need to make a number of necessary configurations in order for the action to function properly. In the next chapter, we will discuss modifications that will be made to the information property list file for the action. These modifications will affect how the action is viewed within Automator, as well as how it is handled within a workflow.



Chapter 12 Configuring an Action's Property List File

Once you have created an initial Automator action project, it is time to begin configuring the action for usage. In order for an action to function properly, its information property list file, also called an *info.plist* file, must be properly configured. An information property list file is a structured XML file that contains data keys and values.

When you create a new action project in Xcode, an *info.plist* file is automatically included in your project. This file is read by Automator on launch, and is used in order to display information about the action within Automator's interface, such as its name, description, and icon. In addition, this file is also used to control certain aspects of the action's behavior both during configuration and during runtime, such as the types of input the action can accept, as well as the types of output it provides.

Editing Properties

If you are not familiar with the process of editing information property list files, it is fairly straightforward. By default, an action's *info.plist* file will already be pre-configured to contain all of the keys that you will need to include in the file, along with default values for those keys. When making changes to this property list file, you will be able to simply modify the default values for these existing keys.

Since an *info.plist* file is XML based, it may be edited in a standard text editor application, in Xcode, or in a property list editor.

Editing in Xcode

If desired, the *info.plist* file for your action project may be edited as text from directly within Xcode. To do this, double click on the *info.plist* file within the *Groups & Files* list for your project. Xcode will open the XML of the *info.plist* file for manual editing. See figure 12.1.



Chapter 12: Configuring an Action's Property List File

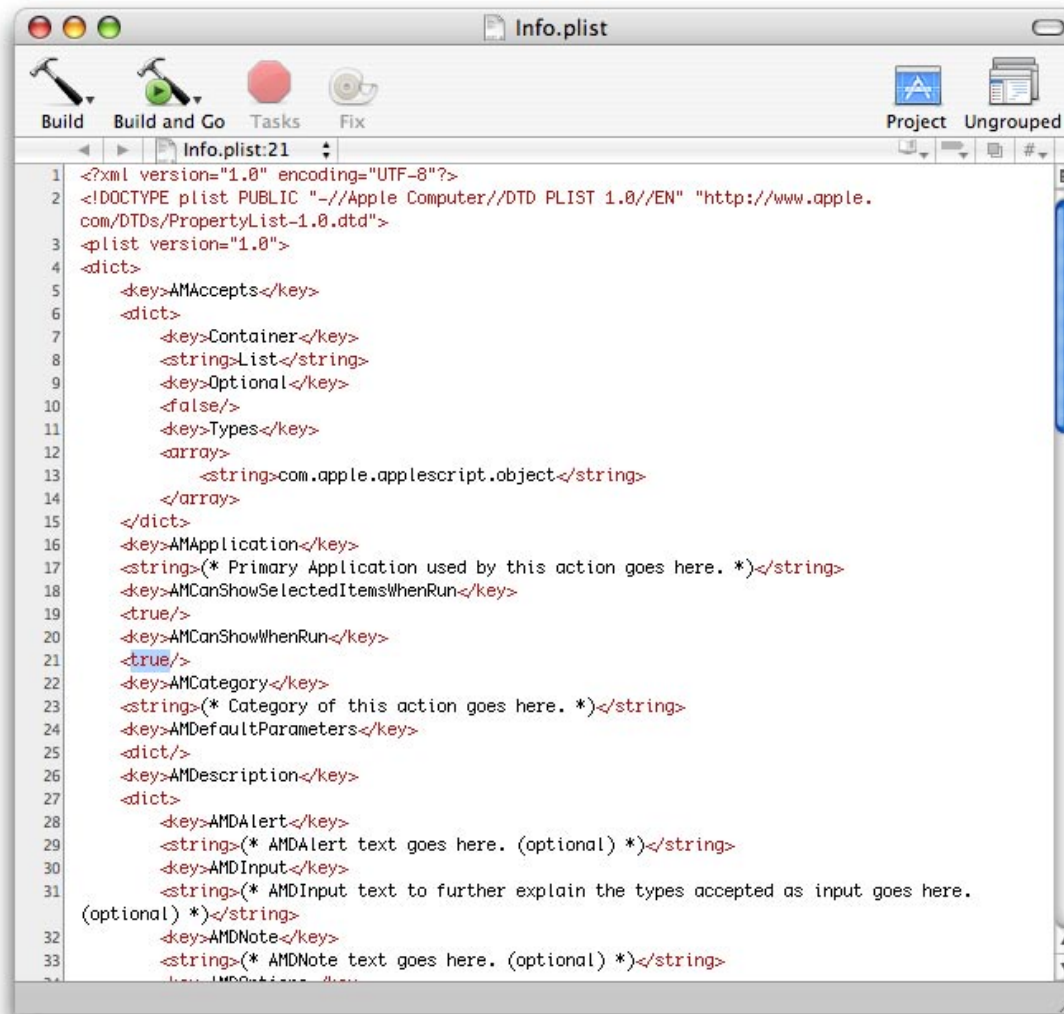


Figure 12.1 Editing an info.plist File within Xcode



Chapter 12:
**Configuring
an Action's
Property
List File**

Editing in a Third-Party Editor

If you are not comfortable editing the XML of *info.plist* file on your own, you may choose to use a property list editor to do the editing. A property list editor will allow you to edit the keys and values contained within a property list file without the need to edit the actual XML.

Apple includes an application, appropriately named Property List Editor, with the Xcode tools. This application can be found in the *Developer > Applications > Utilities* folder. Other third-party tools are also available for editing plist files, such as PlistEdit Pro (<http://homepage.mac.com/bwebster/plisteditpro.html>).

By default, if you have installed the Xcode tools, double clicking on an *info.plist* file in the Finder will launch Property List Editor, and display the *info.plist* file for editing. If you prefer to use a different application for editing plist files, you may modify this default behavior by selecting a plist file in the Finder, and opening its information window (*command + i*). Next, select the desired application from the popup button under *Open with*. See figure 12.2. If the desired application is not included in the list, select *Other...*, and navigate to and select the application manually. To apply this change to all *info.plist* files on your computer, click the *Change All* button after selecting the desired application in the information window.

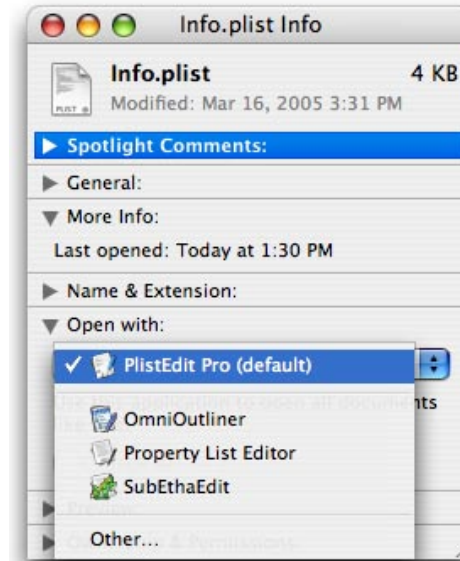


Figure 12.2 *Modifying the Default Application for plist Files*

To open an *info.plist* file in a property list editor from within Xcode, control click on the *info.plist* file in the *Groups & Files* list for your project. Next, select *Open with Finder* from the displayed contextual menu. See figure 12.3. This will cause the Finder to open the *info.plist* file within its default application.



Chapter 12:
**Configuring
an Action's
Property
List File**

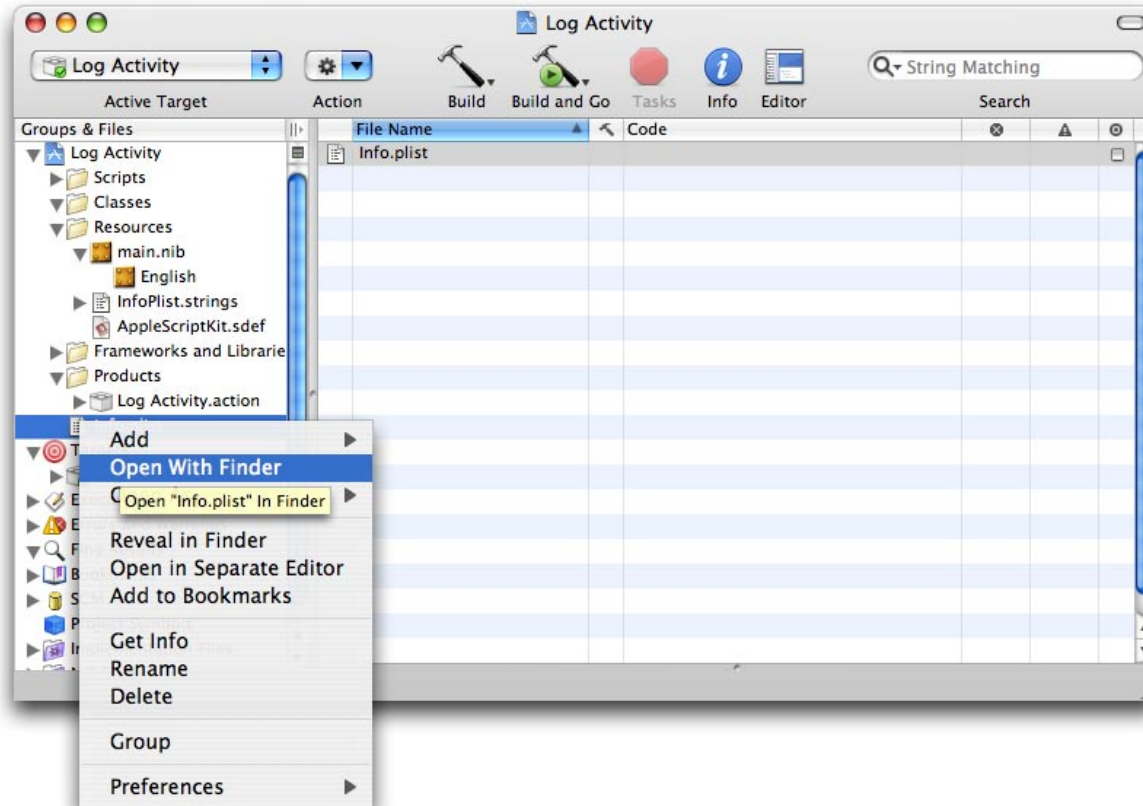


Figure 12.3 *Opening a plist file in a Third Party Editor from Within Xcode*

Once opened in a property list editor such as Property List Editor or PlistEdit Pro, a table view will provide a quick and easy way to modify existing keys and values, or add new keys and values. Figure 12.4 shows an *info.plist* file that has been opened in PlistEdit Pro.



Chapter 12:
**Configuring
an Action's
Property
List File**

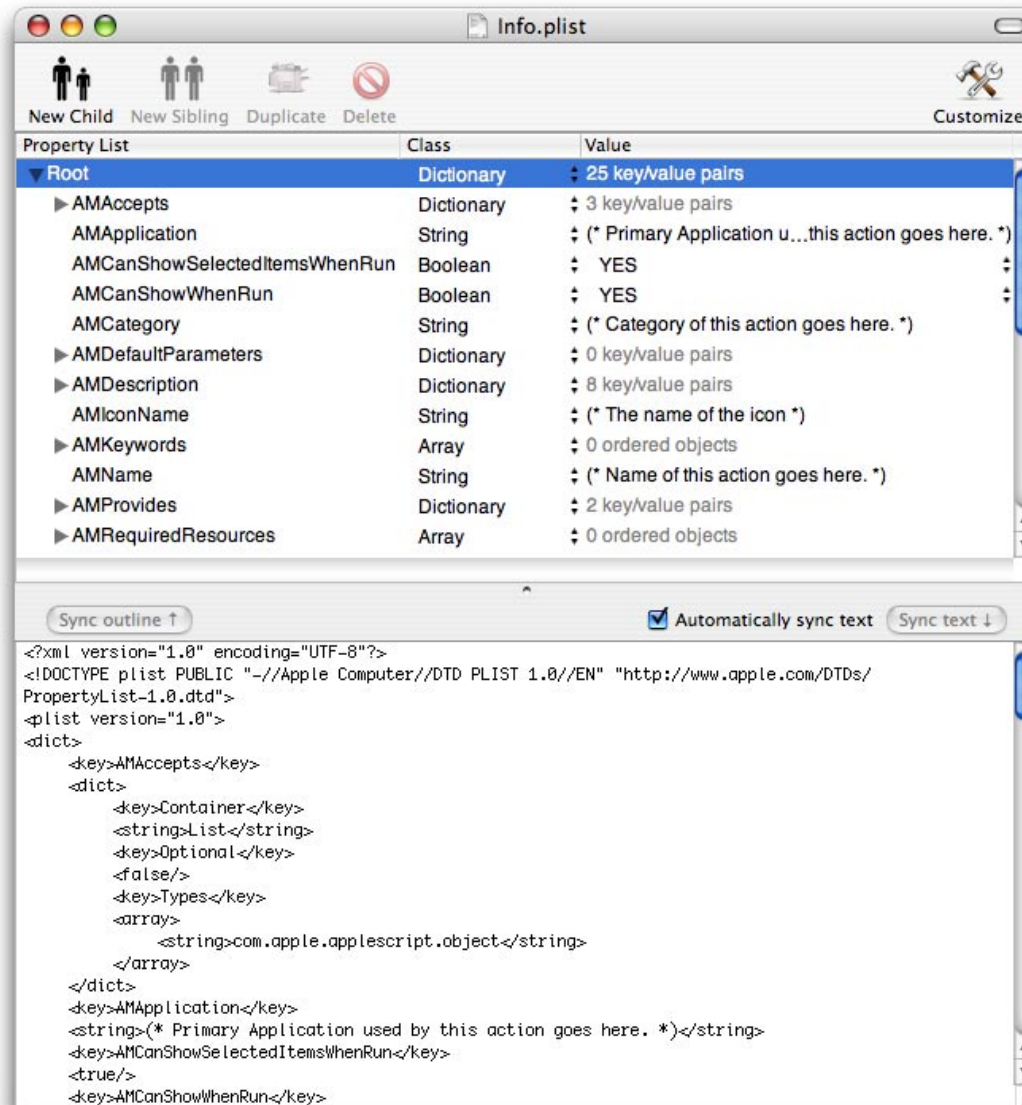


Figure 12.4 Editing a plist File in PlistEdit Pro



Chapter 12:
**Configuring
an Action's
Property
List File**

Configuring General Action Properties

The keys and values in an action's *info.plist* file will affect how an action is handled when run in a workflow, as well as how it is displayed within Automator's interface. Throughout the remainder of this chapter, we will discuss these keys and values, which will be organized into different groups. First, we will discuss some general keys and values, which pertain to an action as a whole. These keys must be present within the *info.plist* file for the action to be properly loaded by Automator. As we proceed through this chapter, example XML syntax will be provided for each key value pair. At the end of the chapter, a complete sample of a configured *info.plist* file will be provided. As you proceed with editing the *info.plist* file, you will notice that all Automator-related keys will begin with a prefix of *AM*.

Specifying the Action's Name

The **AMName** key should be set to a string value indicating the name of the action, as it will be listed in the *Action* list within Automator's interface. For example, setting the value of this key to *Log Activity* would cause the action to appear in the *Action* list as *Log Activity*. The value of this key is also used as search criteria when a user performs an action search in Automator.

The following is an example of a properly configured **AMName** key and value:

```
<key>AMName</key>  
<string>Log Activity</string>
```

Specifying the Action's Application

The **AMApplication** key should be set to a string value indicating the name of the application with which the action will interact. This value will determine in which application category the action will be listed in the *Library* list within Automator's interface. For example, setting the value of this key to *TextEdit* would cause the action to appear under the application category *TextEdit*, along with any other installed TextEdit actions. The value of this key is also used as search criteria when performing a search in Automator.

The following is an example of a properly configured **AMApplication** key value:

```
<key>AMApplication</key>  
<string>TextEdit</string>
```

In some cases, you may want an action to be displayed under multiple categories within Automator. This can be done by setting the value of the **AMApplication** key to an array of string values. For example, the following key value combination would cause an action to appear in both the TextEdit and Automator categories.



Chapter 12: Configuring an Action's Property List File

```
<key>AMApplication</key>
<array>
  <string>TextEdit</string>
  <string>Automator</string>
</array>
```

If you specify the name of a valid application in the **AMApplication** key, then the application's icon will be displayed alongside the category's name in the *Library* list within Automator. If you specify a category that is not a valid application name, then a generic icon will be displayed next to the category's name in the *Library* list. See figure 12.5.

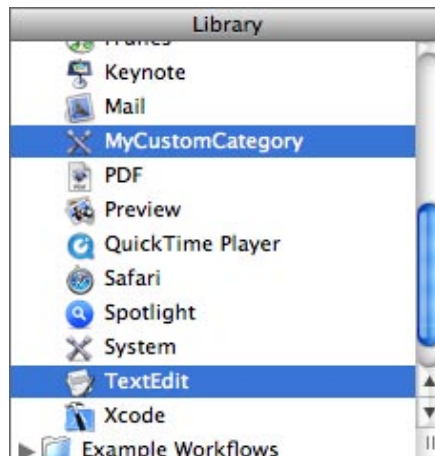


Figure 12.5 Application Category Icons

Specifying the Action's Category

The **AMActionCategory** key should be set to a string value indicating into which type of category the action fits. Automator uses this key value in order to group

related actions together. For example, the category of an action that performs activity logging might fit into a category of *Logs*, which includes other similar logging actions.

The following is an example of a properly configure **AMActionCategory** key value:

```
<key>AMCategory</key>
<string>Logs</string>
```

Specifying an Action's Keywords

The **AMKeywords** key should contain an array of string values indicating keywords that may be used in order to locate the action when performing searches within Automator. For example, an action named *Log Activity*, that performs activity logging, might contain the words “Generate” and “Logging” as keywords. By default, words that are included in the name of an action will be included as search criteria. Therefore, it is not necessary to specify these words as keywords.

The following is an example of a properly configured **AMKeywords** key value:

```
<key>AMKeywords</key>
<array>
  <string>Generate</string>
  <string>Logging</string>
</array>
```



Chapter 12:
**Configuring
an Action's
Property
List File**

Specifying an Action's Bundle Identifier

In order for an action to be registered as a unique loaded bundle within Automator, you must be sure to configure the action's **CFBundleIdentifier** key value. The value specified for this key must be a unique string value for each action, so that no actions conflict with one another. As a general rule, the following formula may be used in order to determine a unique **CFBundleIdentifier** key value.

```
com.«Company Name or User Name».  
Automator.«Action Name»
```

The following is an example of a properly configured **CFBundleIdentifier** key value:

```
<key>CFBundleIdentifier</key>  
<string>com.spiderworks.Automator.Log  
Activity</string>
```

Configuring an Action's Icon

When creating a custom action, by default, Automator will use a generic icon to represent the action within the *Action* list in Automator's interface. See figure 12.6.



Figure 12.6 *A Generic Action Icon*

However, unless your action will perform a generic task, your action should be configured with an icon relative to its function. This may be done by referencing an image file or an icon file in the **AMIconName** key value.

Using an Apple Application Icon

By default, Automator's application bundle contains a number of existing icons for many Apple applications in Mac OS X, including Address Book, iPhoto, iTunes, and more. These icons can be found in the *Contents > Resources* directory within Automator's bundle.



Chapter 12: Configuring an Action's Property List File

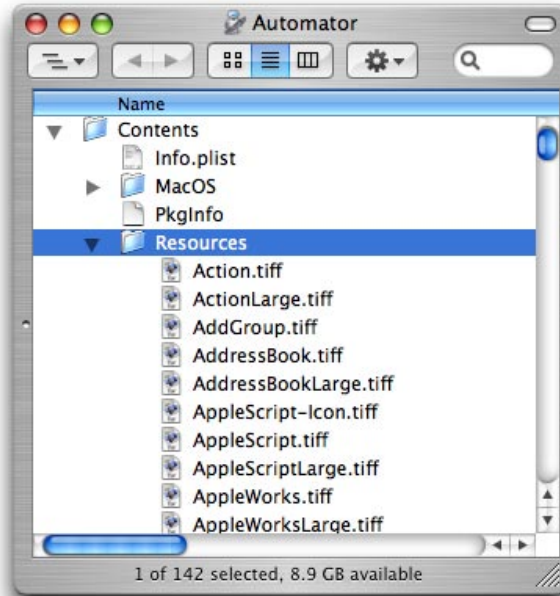


Figure 12.7 Automator's Existing Available Icons

To use any of the icons contained within the Automator bundle, enter the name of the icon, with or without the *icns* or *tiff* extension, as a string into the **AMIconName** key value.

The following is an example of a properly configured **AMIconName** key value for displaying an icon from within the Automator bundle. This example will display the TextEdit icon for the action within Automator's *Action* list.

```
<key>AMIconName</key>  
<string>TextEdit</string>
```

Using a System Icon

When configuring an action's icon, you may also choose to use any system icon that is contained within the *CoreTypes* bundle file. This bundle is located in the *System > Library > CoreServices* folder in Mac OS X. Within this bundle, a lengthy list of icons can be found in the *Contents > Resources* directory. To use any of the icons contained within this directory, enter the name of the icon, with or without the *icns* extension, as a string into the **AMIconName** key value.

The following is an example of a properly configured **AMIconName** key value for displaying an icon from within the *CoreTypes* bundle. This example will display the icon for an executable binary file for the action within Automator.

```
<key>AMIconName</key>  
<string>ExecutableBinaryIcon</string>
```

Using a Custom or Third-Party Application Icon

It is also possible to display a custom icon for your action, or the icon of a third-party application. To do this, you will need to generate the icon as either an *icns* file, or in TIFF format. If you generate the icon in TIFF format, then create two separate TIFF files, one large (32 x 32) and one small (16 x 16), and name them *Large.tiff* and *Small.tiff* respectively.

Next, import the icon or TIFF files into the action project by dragging them into the project's *Groups & Files* list in Xcode. When prompted, choose to *Copy*



Chapter 12: Configuring an Action's Property List File

items into destination group's folder (if needed). See figure 12.8. This will ensure that the files are copied into your project folder, rather than simply referenced from elsewhere. Figure 12.9 shows a project containing an imported *icns* file.

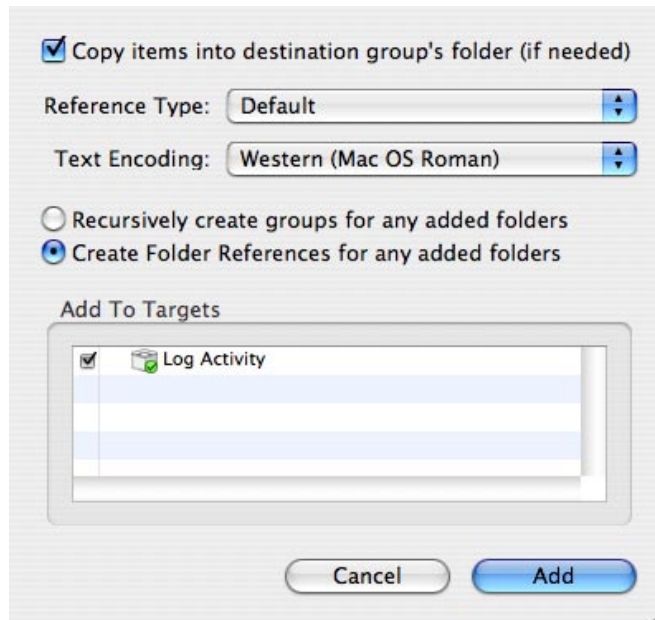


Figure 12.8 Adding a Custom Icon to a Project

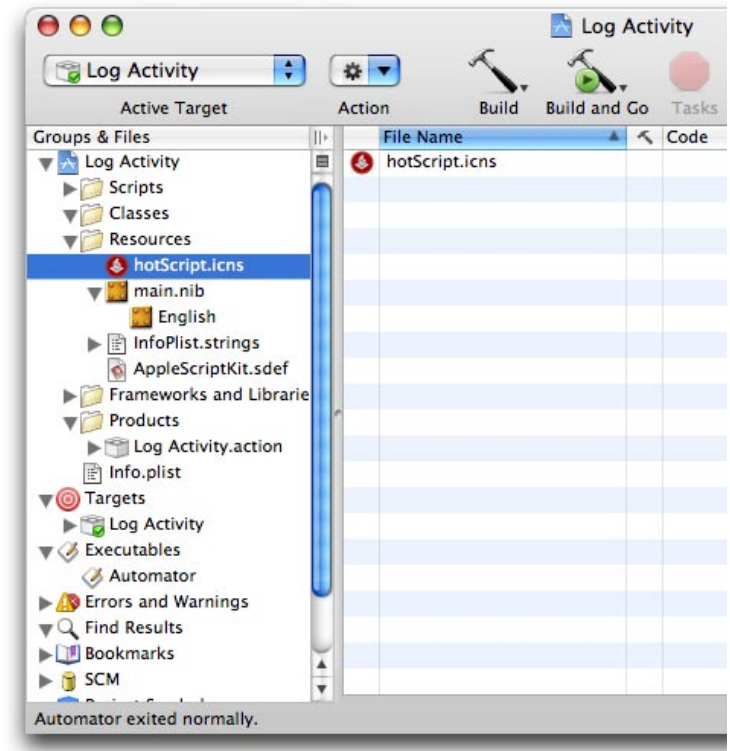


Figure 12.9 A Custom Icon Added to a Project

To configure the action to display an icon that has been imported into the project, enter the name of the icon as a string into the **AMIconName** key value. For icon files, be sure to include the *icns* extension. For TIFF files, you do not need to include the *tiff* extension, if you do not want to do so.

The following is an example of a properly configured **AMIconName** key value for displaying an icon that has been imported into an action project.



Chapter 12:
**Configuring
an Action's
Property
List File**

```
<key>AMIconName</key>  
<string>hotScript.icns</string>
```

If TIFF files were imported into the action project, then enter the name of the large TIFF file. For example:

```
<key>AMIconName</key>  
<string>Large</string>
```

Automator will automatically utilize the small TIFF file in order to display the small version of the action's icon, when necessary.

Configuring an Action's Description

The next set of property list keys and values we will discuss pertain to the action's description. In Automator, when a user selects an action in the *Action* list or in a workflow, a description of the action is displayed in the lower left corner of the Automator window. The contents of this description area are derived from the **AMDescription** key in the *info.plist* file. The value for this key consists of an XML dictionary of additional keys and values, which will be displayed as elements within the description area in Automator's interface. Not all of these keys are required. Those that are not necessary or required should be removed from the *info.plist* file so that they do not appear as empty values in the action's displayed description.

Please note that the values entered for the keys below will not affect the behavior of the action. These keys are used only for the purpose of displaying the action's description.

Summary Field

A brief summary of the action should be entered as a string value for the **AMDSummary** key. This value will be displayed in the description area in Automator, immediately beneath the action's name and icon. This key value is required.

The following is an example of a properly configured **AMDSummary** key value:



Chapter 12: Configuring an Action's Property List File

```
<key>AMDSummary</key>
<string>This action will generate an entry in an
activity log.</string>
```

Input Field

The **AMDInput** key is used in order to specify the type of input that the action will accept. The type of input the action accepts will be displayed in the description area in Automator, preceded by the label *Input*. The value for this key should be a string.

By default, Automator will automatically display the type of input that an action is configured to accept via the **AMAccepts** property, described later in this chapter. Therefore, it is not necessary to enter text for this key value unless you wish to be more specific in describing the action's input to the user. If it is not necessary to be more specific, then this key and value may be removed from the *info.plist* file, or the key value may be set to an empty string, and Automator will continue to display the input type for the action.

The following is an example of a properly configured **AMDInput** key value containing text to be displayed:

```
<key>AMDInput</key>
<string>Data from application X</string>
```

The following is an example of a properly configured **AMDInput** key value that does not contain any text to be displayed:

```
<key>AMDInput</key>
<string></string>
```

Result Field

The **AMDResult** key is used in order to specify the type of value that the action will output as its result. The type of output the action generates will be displayed in the description area in Automator, preceded by the label *Result*. The value for this key should be a string.

By default, Automator will automatically display the type of output that an action is configured to generate via the **AMProvides** property, described later in this chapter. Therefore, it is only necessary to include this key and value if you wish to be more specific in describing the action's output to the user. If it is not necessary to be more specific, then this key and value may be removed from the *info.plist* file, or the value may be set to an empty string.

The following is an example of a properly configured **AMDResult** key value that contains text to be displayed:

```
<key>AMDResult</key>
<string>Application X Documents</string>
```

The following is an example of a properly configured **AMDResult** key value that does not contain any text to be displayed:

```
<key>AMDResult</key>
<string></string>
```



Chapter 12: Configuring an Action's Property List File

Options Field

The **AMDOptions** key is used in order to provide information about the settings that are available for modification by the user with the action's interface, once added to a workflow. The value for this key should be a string. If an action has no settings to be configured by the user, then this key and value should be removed from the *info.plist* file.

The value entered for this key should be used in order to provide a *brief* overview of the action's settings, or to provide additional details about the settings that cannot be derived from the action's interface itself. The value specified for this key will be displayed in the description area in Automator, preceded by the label *Options*.

The following is an example of a properly configured **AMDOptions** key value:

```
<key>AMDOptions</key>
<string>Workflow name, log name, location, and
whether to include the time in each activity
log entry.</string>
```

Requirements Field

The **AMDRequires** key is used in order to provide information about any specific requirements that must be met in order for the action to successfully execute. For example, you might use this field in order to specify that a document in a specified application must already be opened prior to triggering the action. The value specified for this key will be displayed in the description area in Automator, preceded by the label *Requires*. The

value for this key should be a string. If an action has no specific requirements, then this key and value should be removed from the *info.plist* file.

The following is an example of a properly configured **AMDRequires** key value:

```
<key>AMDRequires</key>
<string>An opened Application X
document.</string>
```

Note Field

The **AMDNote** key is used in order to provide additional information to the user that does not fit under another description key. The value specified for this key will be displayed in the description area in Automator, preceded by the label *Note*. The value for this key should be a string. If an action has no note to display, then this key and value should be removed from the *info.plist* file.

The following is an example of a properly configured **AMDNote** key value:

```
<key>AMDNote</key>
<string>The original document will not be
modified by this action.</string>
```

Alert Field

The **AMDAlert** key is used in order to alert the user to the consequences of running the action. For example, you might use this field in order to indicate that a task being performed by the action will be permanent, and



Chapter 12: Configuring an Action's Property List File

may not be undone. The value specified for this key will be displayed in the description area in Automator, preceded by the label *Warning*. The value for this key should be a string. If an action does not need an alert message, then this key and value should be removed from the *info.plist* file.

Please note that this key and value only affects the description area of the action. It will not actually cause an alert dialog to appear. The method for displaying an alert dialog when configuring a workflow will be explained later in this chapter.

The following is an example of a properly configured **AMDAAlert** key value:

```
<key>AMDAAlert</key>
<string>The original document will be modified.
This may not be undone.</string>
```

Related Actions Field

The **AMDRelatedActions** key is used in order to provide the user with a list of any other actions that may be related to the task that will be performed by the current action. For example, if the action will manipulate documents in Preview, then you may want to let the user know that the *Open Images in Preview* action would be a good action to run in a workflow prior to the current action. The value specified for this key will be displayed in the description area in Automator, preceded by the label *Related Actions*. If an action does not have any related actions, then this key and value should be removed from the *info.plist* file.

If a single action will be listed, then this key should contain a string value indicating the bundle identifier for the action to be listed. If multiple actions should be listed, then this key should contain an array of string values indicating the bundle identifiers for the actions to be listed. The bundle identifier for an action can be determined by opening the *info.plist* file stored in the action's bundle, and determining the value of the **CFBundleIdentifier** property.

The following is an example of a properly configured **AMDRelatedActions** key value for displaying a single related action:

```
<key>AMDRelatedActions</key>
<string>com.apple.Automator.
OpenImagesInPreview</string>
```

The above example value would be displayed in the description area in Automator as:

Related Actions: Open Images in Preview



Chapter 12:

Configuring an Action's Property List File

The following is an example of a properly configured **AMDRelatedActions** key value for multiple related actions:

```
<key>AMDRelatedActions</key>
<array>
  <string>com.apple.Automator.CopyFinderItems
</string>
  <string>com.apple.Automator.
    OpenImagesInPreview</string>
</array>
```

The above example value would be displayed in the description area in Automator as:

Related Actions: Copy Finder Items, Open Images in Preview

Configuring Action Input and Output Values

In order for an action to receive input and generate output, the action's information property list file must be updated to contain an array of acceptable input and output types. To do this, you will need to modify the **AMAccepts** and **AMProvides** key values by inserting what are known as **type identifiers** (also called **uniform type identifiers** or **UTI's**). A type identifier represents a type of Automator data that may be passed between actions. For example, a type identifier might represent files and folders, text, URLs, application documents, etc. We will discuss configuring the **AMAccepts** and **AMProvides** key values shortly. However, let's first discuss the type identifiers that Automator will understand.

Input/Output Type Identifiers

Automator has been written to understand a number of *built-in* type identifiers, which are specified below. This list of type identifiers includes AppleScript and Cocoa type identifiers, as well as a public type identifier, that may be used to generically refer to files and folders. See figures 12.10, 12.11, and 12.12. The list also includes type identifiers that are very specific, such as one for an iPhoto album, one for an iTunes playlist, etc. See figure 12.13. A small set of type identifiers is also available for Microsoft PowerPoint. See figure 12.14. In the future, this list of built-in type identifiers will most likely grow to include other third-party applications as well.



Chapter 12:
**Configuring
an Action's
Property
List File**

Type Identifier	Data Type
com.apple.applescript.object	Generic AppleScript Object
com.apple.applescript.alias-object	AppleScript Alias
com.apple.applescript.alias-object.image	AppleScript Image Alias
com.apple.applescript.alias-object.movie	AppleScript Movie Alias
com.apple.applescript.alias-object.pdf	AppleScript PDF Alias
com.apple.applescript.data-object	AppleScript Data Reference
com.apple.applescript.text-object	AppleScript Text Reference
com.apple.applescript.url-object	AppleScript URL Reference

Figure 12.10 AppleScript Type Identifiers

Type Identifier	Data Type
com.apple.cocoa.path	String Path
com.apple.cocoa.string	String
com.apple.cocoa.url	URL

Figure 12.11 Cocoa Type Identifiers

Type Identifier	Data Type
public.item	File/Folder

Figure 12.12 Public Type Identifiers



Chapter 12:
**Configuring
an Action's
Property
List File**

Type Identifier	Data Type
com.apple.addressbook.group-object	Address Book Group Reference
com.apple.addressbook.item-object	Address Book Item Reference
com.apple.addressbook.person-object	Address Book Person Reference
com.apple.finder.file-or-folder-object	Finder File or Folder Reference
com.apple.ical.calendar-object	iCal Calendar Reference
com.apple.ical.event-object	iCal Event Reference
com.apple.ical.item-object	iCal Item Reference
com.apple.ical.todo-object	iCal To Do Reference
com.apple.idvd.menu-object	iDVD Menu Reference
com.apple.idvd.slideshow-object	iDVD Slideshow Reference
com.apple.iphoto.album-object	iPhoto Album Reference
com.apple.iphoto.item-object	iPhoto Item Reference
com.apple.iphoto.photo-object	iPhoto Photo Reference
com.apple.itunes.item-object	iTunes Item Reference
com.apple.itunes.playlist-object	iTunes Playlist Reference
com.apple.itunes.source-object	iTunes Source File Reference
com.apple.itunes.track-object	iTunes Track Reference
com.apple.mail.account-object	Mail Account Reference
com.apple.mail.item-object	Mail Item Reference
com.apple.mail.mailbox-object	Mail Mailbox Reference
com.apple.mail.message-object	Mail Message Reference
com.apple.safari.document-object	Safari Document Reference
com.apple.spotlight.item	Spotlight Item Reference
com.apple.textedit.document-object	TextEdit Document Reference

Figure 12.13 Application Specific AppleScript Type Identifiers

Type Identifier	Data Type
com.microsoft.powerpoint.presentation-object	PowerPoint Presentation Reference
com.microsoft.powerpoint.slide-object	PowerPoint Slide Reference

Figure 12.14 Microsoft Specific AppleScript Type Identifiers



Chapter 12: Configuring an Action's Property List File

These charts of built-in type identifiers may also be found in *Appendix B*. In addition to handling built-in type identifiers, Automator can load and understand type identifiers for third-party applications. In order to do this, each third party application must install, into one of the *Library > Automator* folders, a definition file outlining the type identifiers supported by that application.

Configuring Input

By configuring an action to only accept certain types of input, you can help to prevent the action from running during unanticipated situations within a user's workflow. Rather than needing to add code to handle an improper type of data, Automator will automatically detect the input type mismatch, and produce an error if the action is not passed the proper type of data from a previous action.

The **AMAccepts** key represents an XML dictionary that contains the following keys:

- ▶ **Container** – This key contains a string value indicating in which format input is accepted. The default value for this key is set to *List*, and you will probably not need to change this value if your action will be configured to process a list of input values.
- ▶ **Optional** – This key consists of a Boolean value indicating whether the action can be configured to ignore its input. By default, this value is set to **false**, indicating that input is required. If you change this value to **true**, then users will be able to

configure your action to ignore its input.

- ▶ **Types** – This key consists of an array of string values for the type identifiers that indicate the types of input that the action can process.

The following is an example of a properly configured **AMAccepts** key:

```
<key>AMAccepts</key>
<dict>
  <key>Container</key>
  <string>List</string>
  <key>Optional</key>
  <false/>
  <key>Types</key>
  <array>
    <string>com.apple.applescript.object
    </string>
    <string>com.apple.finder.file-or-
    folder-object</string>
  </array>
</dict>
```

In this example, the action will accept as input any generic AppleScript value, as well as Finder item paths. The action will also not allow the user to ignore the input for the action.

Configuring Output

Configuring the output of an action will help to ensure that the action is properly paired in a workflow with an action that will accept its result.

Like the **AMAccepts** key, the **AMProvides** key represents an XML dictionary of keys and values



Chapter 12: Configuring an Action's Property List File

describing the type of output generated by the action. This XML dictionary contains the following keys:

- ▶ **Container** – This key contains a string value indicating in which format output is provided. The default value for this key is set to *List*, and you will probably not need to change this value if your action will be configured to produce a list of output values.
- ▶ **Types** – This key consists of an array of string values for the type identifiers that indicate the types of output that the action will provide.

The following is an example of a properly configured **AMProvides** key:

```
<key>AMProvides</key>
<dict>
  <key>Container</key>
  <string>List</string>
  <key>Types</key>
  <array>
    <string>com.apple.finder.file-or-
      folder-object</string>
  </array>
</dict>
```

In this example, the action will provide finder item paths as output for the next action in the workflow sequence.

Adjusting Action Behavior

As we discussed in the first section of this book, it is possible for users to configure certain actions in a workflow to display their settings interfaces during processing. This can allow the person running the workflow to adjust the behavior of the action during processing. The ability for an action to provide users with this flexibility is based upon values that you specify for keys in the action's *info.plist* file.

Show Action When Run Option

The **AMCanShowWhenRun** key represents a Boolean value indicating whether the action may be configured by the user to display its settings interface during processing. The default value for the **AMCanShowWhenRun** key is set to **true**, indicating that this behavior will be enabled. To disable this behavior, set the value of the **AMCanShowWhenRun** property to **false**. The following are examples of properly configured **AMCanShowWhenRun** properties.

```
<key>AMCanShowWhenRun</key>
<true/>
```

OR

```
<key>AMCanShowWhenRun</key>
<false/>
```



Chapter 12:
**Configuring
an Action's
Property
List File**

Specialized Show Action When Run Options

You may recall from the first section of this book, that in some cases, if an action's interface contains multiple settings, it is sometimes possible to allow users to display only specified settings during processing. This behavior is either enabled or disabled by a Boolean value for the **AMCanShowSelectedItemsWhenRun** key in the action's *info.plist* file. The default value for this key is **true**, indicating that a user may configure the action to display only certain settings during processing. However, this behavior may be disabled by setting the value of this key to **false**. The following are examples of a properly configured **AMCanShowSelectedItemsWhenRun** property.

```
<key>AMCanShowSelectedItemsWhenRun</key>  
<true/>
```

OR

```
<key>AMCanShowSelectedItemsWhenRun</key>  
<false/>
```

Specifying Required Resources

Sometimes, an action may require certain resources to be installed in order for an action to be functional. To help prevent a user from running an action in an environment where these required resources are not available, you may configure the **AMRequiredResources** key in the action's *info.plist* file. This key may be configured to require paths or applications necessary for the action to be functional. If this key has been configured with required resources, then Automator will automatically verify that those resources are present prior to loading the action. If a required resource cannot be found, then Automator will not allow the user to configure or run the action until the resource is located.

The **AMRequiredResources** key represents an array of XML dictionaries, each of which may be used to specify a required resource. Each XML dictionary consists of the following keys:

- ▶ **Display Name** – This key represents a string value that indicates the name of the required resource. This key should remain empty if the **Type** key (see below) is set to a value of file.
- ▶ **Resource** – This key represents a string value that indicates the resource that is required. The value entered for this key will vary, depending on the value specified for the **Type** key. The following are acceptable types of resource key values:
 - ▶ Application Identifier, i.e. **com.adobe.indesign**



Chapter 12:

**Configuring
an Action's
Property
List File**

- Application Creator Code, i.e. **InDn**
- Resource Path, i.e. **/Library/ScriptingAdditions/XMail.osax**
- **Type** – This key represents a string value indicating the type of resource that should be located. Acceptable values include the following:
 - **application**
 - **creator code**
 - **file**
- **Version** – This key represents a string value indicating the minimum version of the specified resource that is required in order for the action to process successfully. A version number, if provided, should be entered using the following format:
1.0.1

The following is an example of a properly configured **AMRequiredResources** key value. This example checks for the presence of Adobe InDesign CS2, as well as for a required scripting addition.

```
<key>AMRequiredResources</key>
<array>
  <dict>
    <key>Display Name</key>
    <string> InDesign CS</string>
    <key>Resource</key>
    <string>com.adobe.indesign</string>
    <key>Type</key>
    <string>application</string>
    <key>Version</key>
    <string>3.0</string>
  </dict>
  <dict>
    <key>Display Name</key>
    <string></string>
    <key>Resource</key>
    <string>/Library/ScriptingAdditions/
XMail.osax</string>
    <key>Type</key>
    <string>file</string>
    <key>Version</key>
    <string>1.0</string>
  </dict>
</array>
```



Chapter 12:
**Configuring
an Action's
Property
List File**

Configuring a Warning

In some cases, an action may perform a task that will permanently modify the data being processed by the user. As we have seen, you may provide a warning in the description area of an action. However, you may wish to take this one step further to *really* ensure that the user knows that data will be modified. With the use of the **AMWarning** key, an action may be configured to display a custom warning message to the user whenever the action is placed into a workflow.

This warning message may also be configured to suggest that the user add a specific action prior to inserting the current one. For example, you might suggest that the user insert an action that will back up the data in some manner prior to triggering the current action.

The **AMWarning** key represents an XML dictionary that contains the following keys:

- ▶ **Action** – This key represents a string value indicating the bundle identifier for another action that will be automatically inserted into the workflow by Automator prior to the current action, if the user chooses to do so. The bundle identifier for an action can be determined by opening the *info.plist* file stored in the action's bundle, and determining the value of the **CFBundleIdentifier** property. If no actions should be added via this warning dialog, then the value for this key should be set to an empty string.
- ▶ **ApplyButton** – This key represents a string value indicating the name of the button that will serve

as the *Apply* button in the warning dialog. If the warning requires a specific action to be added, then clicking this button will cause the required action to be added to the workflow prior to placing the current action. If the warning does not require a specific action to be added, then this button will simply dismiss the warning message. The default value for this key is *Continue*.

- ▶ **IgnoreButton** – This key represents a string value indicating the name of the button that will serve as the *Don't Add* button when attempting to add a required action. If this button is clicked, then the required action will not be added. The default value for this key is *Cancel*. This value will be ignored if no required action is specified.
- ▶ **Level** – This key represents an integer value indicating the level of warning that should be displayed. The following are acceptable values:
 - ▶ **0** – This value is the default value for this key, and indicates that no warning message should be displayed.
 - ▶ **1** – This value indicates that the action will modify the users data in a manner that can be undone, once complete. If this value is specified, then the warning message will be displayed with a generic Automator icon.
 - ▶ **2** – This value indicates that the modifications that will be done to the user's data will be permanent, and may not be undone. If this value is specified, then the warning message will be displayed with a warning icon.



Chapter 12:
**Configuring
an Action's
Property
List File**

- **Message** – This key represents a string value indicating the contents of the warning message that should be displayed to the user.

The following is an example of a properly configured **AMWarning** key value. This example requires that the *Copy Finder Items* action be added to the workflow prior to the current action.

```
<key>AMWarning</key>
<dict>
  <key>Action</key>
  <string>com.apple.Automator.CopyFiles
</string>
  <key>ApplyButton</key>
  <string>Add</string>
  <key>IgnoreButton</key>
  <string>Don't Add</string>
  <key>Level</key>
  <integer>2</integer>
  <key>Message</key>
  <string>This action will modify your data.
  Would you like to add the Copy Finder Items
  action first?</string>
</dict>
```

The following is another example of a properly configured **AMWarning** key value. This example does not require an action.

```
<key>AMWarning</key>
<dict>
  <key>Action</key>
  <string></string>
  <key>ApplyButton</key>
  <string>Add</string>
  <key>IgnoreButton</key>
  <string></string>
  <key>Level</key>
  <integer>2</integer>
  <key>Message</key>
  <string>This action will modify your data.
  Proceed?</string>
</dict>
```




Chapter 12:
**Configuring
an Action's
Property
List File**

Localized Property List Strings

In addition to an *info.plist* file, an Automator action project also contains an *InfoPlist.strings* file. The purpose of this *InfoPlist.strings* file is to allow you to localize string values in different languages for keys contained within the *info.plist* file. By default, an Automator action project contains an English localization of this file. Additional localizations may be added for other languages as needed.

Like an *info.plist* file, a *.strings* file is a text-based file that contains keys and values associated with those keys. However, unlike an *info.plist* file, a *.strings* file is not XML based. The formatting for a key value pairing within a *.strings* file is as follows:

```
«key» = "«string value»"
```

By default, the English localized strings within the *InfoPlist.strings* file contain a number of pre-configured key values, including those for the action's name and description keys, as well as for its warning message keys. If you are using English as your preferred system language, then these key values will override the corresponding key values specified within your action's *info.plist* file. Therefore, in order for these strings to be properly displayed in Automator, you will need to either carry the values from the included localized strings over from your action's *info.plist* file into the *InfoPlist.strings* file, or you will need to remove the default keys and values from the *InfoPlist.strings* file all together.

You may open the *InfoPlist.strings* file within Xcode in the same manner as an *info.plist* file, by double clicking on it in the *Groups & Files* list in your project. Doing so will display the contents of the file for editing. See figure 12.15.

Once opened, you may make any necessary adjustments to the keys and values specified. If a key is not being used within the *info.plist* file, then the corresponding key and value may be removed from the *InfoPlist.strings* file.



Chapter 12:
**Configuring
an Action's
Property
List File**

```
1  /* Localized versions of Info.plist keys */
2
3  CFBundleName = "Log Activity";
4  NSHumanReadableCopyright = "Copyright 2005 Automated Workflows, LLC.";
5
6  AMName = "Log Activity";
7
8  /* AMApplication localized strings */
9  /* e.g. "TextEdit" = "TextEdit"; */
10
11 /* AMCategory localized strings */
12 /* e.g. "Text" = "Text"; */
13
14 /* AMDefaultParameters localized values */
15 /* e.g. myDefaultIntegerParameter = 0; */
16 /* e.g. myDefaultStringParameter = "Default String Value"; */
17
18 /* AMDescription localized strings */
19 AMOptions = "Workflow name, log name, location, and whether to include the time in each activity log entry.";
20 AMDSummary = "This action will generate an entry in an activity log during processing of a workflow.";
21
22 /* AMKeyword localized strings */
23 /* e.g. "Filter" = "Filter"; */
24
25 /* AMWarning localized strings */
26 ApplyButton = "";
27 IgnoreButton = "";
28 Message = "";
```

Figure 12.15 A Configured InfoPlist.strings File



Chapter 12:
**Configuring
an Action's
Property
List File**

What's Next

Now that we have discussed the preparation of an action's *info.plist* file, we can begin development on the action's settings interface. In the next chapter, we will leave the Xcode application environment briefly, while we walk through the steps involved in designing an interface in Interface Builder.

Also, please note that the *info.plist* file for an action does contain some keys that were not mentioned in this chapter. We will revisit the *info.plist* file again, once we have completed our discussion on interface creation, as it will be necessary to configure the keys and values for interface parameters that will be utilized within the action's processing code.

Example info.plist File

The following is an example of a configured *info.plist* file for an Automator action project:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD
PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>AMAccepts</key>
  <dict>
    <key>Container</key>
    <string>List</string>
    <key>Optional</key>
    <false/>
    <key>Types</key>
    <array>
      <string>com.apple.applescript.
object</string>
    </array>
  </dict>
  <key>AMApplication</key>
  <array>
    <string>TextEdit</string>
    <string>Automator</string>
  </array>
  <key>AMCanShowSelectedItemWhenRun</key>
  <false/>
  <key>AMCanShowWhenRun</key>
  <true/>
  <key>AMCategory</key>
  <string>Logs</string>
  <key>AMDefaultParameters</key>
  <dict>
    <key>workflowName</key>
    <string></string>
    <key>logName</key>
    <string>Activity Log</string>
```



Chapter 12: Configuring an Action's Property List File

```
<key>outputLocation</key>
<string>~/Documents</string>
<key>includeTime</key>
<true/>
</dict>
<key>AMDescription</key>
<dict>
  <key>AMDOptions</key>
  <string>Workflow name, log name, location,
  and whether to include the time in each
  activity log entry.</string>
  <key>AMDSummary</key>
  <string>This action will generate an entry
  in an activity log during processing of a
  workflow.</string>
</dict>
<key>AMIconName</key>
<string>hotScript.icns</string>
<key>AMKeywords</key>
  <array>
    <string>Generate</string>
    <string>Activity</string>
    <string>Logging</string>
  </array>
<key>AMName</key>
<string>Log Activity</string>
<key>AMProvides</key>
<dict>
  <key>Container</key>
  <string>List</string>
  <key>Types</key>
  <array>
    <string>com.apple.applescript.
    object</string>
  </array>
</dict>
<key>AMRequiredResources</key>
</array>
<key>AMWarning</key>
<dict>
```

```
<key>Action</key>
<string></string>
<key>ApplyButton</key>
<string></string>
<key>IgnoreButton</key>
<string></string>
<key>Level</key>
<integer>0</integer>
<key>Message</key>
<string></string>
</dict>
<key>CFBundleDevelopmentRegion</key>
<string>English</string>
<key>CFBundleExecutable</key>
<string>Log Activity</string>
<key>CFBundleGetInfoString</key>
<string>Log Activity version 1.0, Copyright (c)
2005, SpiderWorks, LLC.</string>
<key>CFBundleIconFile</key>
<string></string>
<key>CFBundleIdentifier</key>
<string>com.spiderworks.Automator.Log
Activity</string>
<key>CFBundleInfoDictionaryVersion</key>
<string>6.0</string>
<key>CFBundleName</key>
<string>Log Activity</string>
<key>CFBundlePackageType</key>
<string>BNDL</string>
<key>CFBundleShortVersionString</key>
<string>1.0</string>
<key>CFBundleSignature</key>
<string>????</string>
<key>CFBundleVersion</key>
<string>1.0</string>
<key>NSPrincipalClass</key>
<string>AMAppleScriptAction</string>
</dict>
</plist>
```



Chapter 13 Constructing an Action's Interface

Once you have created an Automator action project, and made some initial modifications to the action's *info.plist* file, the next thing that you will need to do is to design the action's settings interface. The interface that you design will be accessed by users when configuring a workflow. Because this aspect of the action will interact directly with the user, you should take care in the design of the interface. Doing so will ensure a pleasant experience for the user, without confusion. Suggested guidelines for the design of an action's interface will be discussed later in this chapter.

Preparing for Automator Action Interface Development

Prior to beginning development on an action's interface, you may wish to review your Interface Builder settings, and ensure that they are configured to your liking. In addition, you may want to make some adjustments to the interface elements available to you within Interface Builder's palette window.

If this is your first time working with Interface Builder, now might be an excellent time to take a detour and curl up with a good Cocoa or introductory Objective-C book. Once you have built a few basic Cocoa applications, then you will find the rest of this chapter much easier to understand.

Automator Interface Elements

By default, the palette window in Interface Builder contains most of the interface elements that you will ever want to use in your action interfaces, including text fields, buttons, popup menus, etc. However, since the space within an action's interface is limited, additional interface elements are available, that were



Chapter 13:
**Constructing
an Action's
Interface**

designed specifically for use within Automator actions. These interface elements offer more space-conscious ways to allow the user to specify certain information. To load these Automator interface elements, select *Preferences...* under the *Interface Builder* menu within Interface Builder. See figure 13.1.

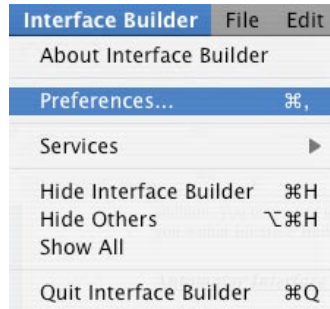


Figure 13.1 Automator's Preferences Menu Item

Once the *Interface Builder Preferences* window is displayed, click the *Palettes* tab. Next, click the *Add* button. When prompted, navigate to the *Developer > Extras > Palettes* folder on your computer, select the file *AMPalette.palette*, click the *Open* button, and close the *Interface Builder Preferences* window. Figure 13.2 shows the *Palettes* tab once the *AMPalette.palette* file has been added.

Once you have loaded the *AMPalette.palette* file, a category of *Automator* interface elements will appear in the palette window when working on a project. These interface elements include an application chooser popup, a directory chooser popup, and a file chooser popup. See figure 13.3.

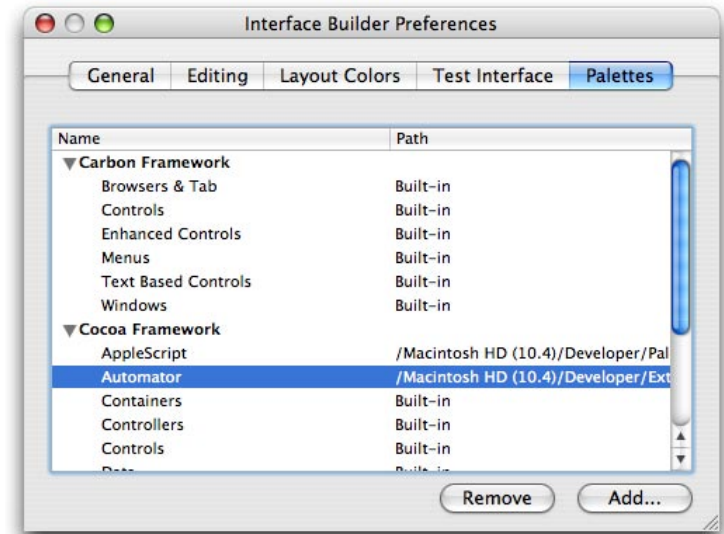


Figure 13.2 Preferences Window Palettes Tab



Figure 13.3 Automator Interface Elements



Chapter 13:
**Constructing
an Action's
Interface**

Building an Action's Interface

Once you have made any necessary adjustments to Interface Builder's settings or configuration, you are ready to begin development on your action's settings interface. Even if you do not wish your action to contain a settings interface, you should still review the topics covered in this chapter, in order to become familiar with the steps involved in constructing an action's interface.

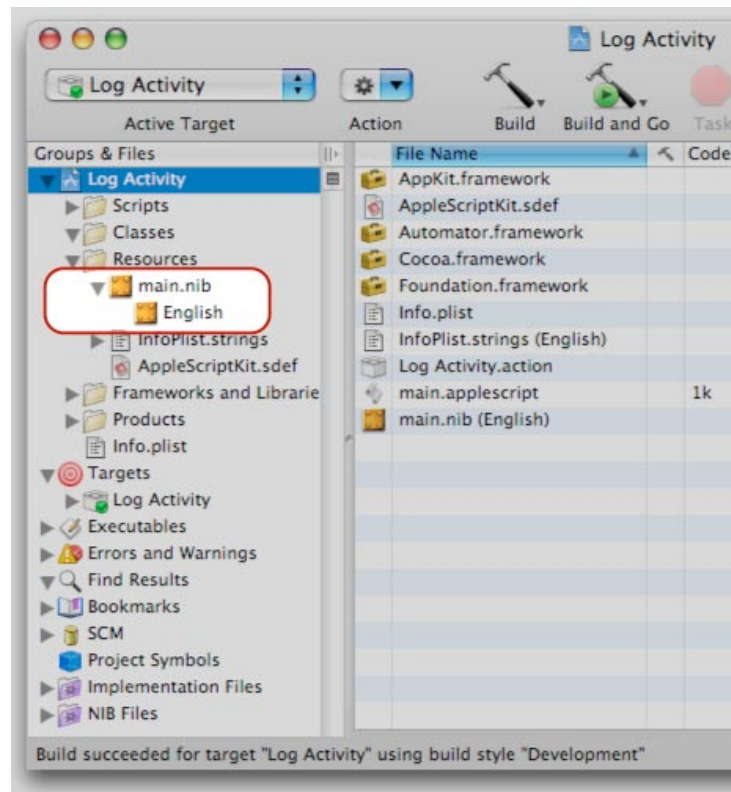


Figure 13.4 The *main.nib* File within a New Project

Opening the Action's Interface

To open the action's interface file in Interface Builder, locate the *main.nib* file under the *Resources* group in the action project's *Groups & Files* list in Xcode, and double click on it. See figure 13.4. Interface Builder will launch, if not already running, and the main *nib* will be displayed. See figure 13.5. By default, the *main.nib* file in a new Automator action will already be partially configured for you.



Figure 13.5 Automator Action Main Nib in Interface Builder

File's Owner Instance

The *File's Owner* instance in the *nib* is already set to the proper custom class. For AppleScript actions, it is set to a class of *AMAppleScriptAction*. See figure 13.6. For Cocoa actions, it is set to a class of *AMBundleAction*.



Chapter 13: Constructing an Action's Interface

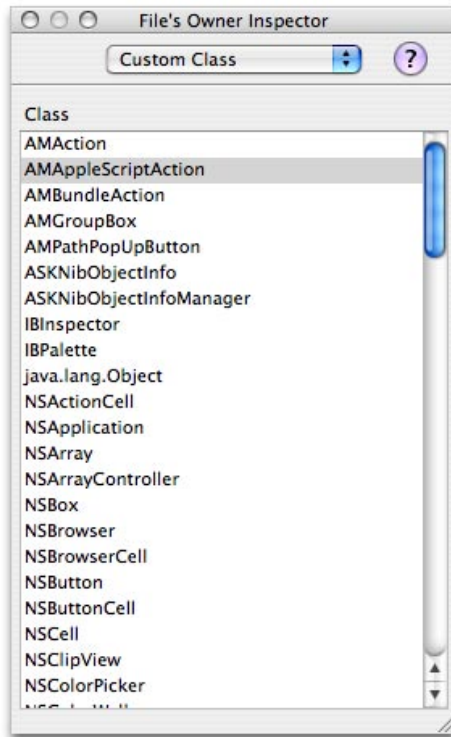


Figure 13.6 File's Owner Custom Class Configuration for an AppleScript Action

In addition, the *view* outlet for the *File's Owner* instance has also been pre-configured, and has been connected to the *view* (*NSView*) instance in the *nib*. See figure 13.7.

Because these settings have been pre-configured for the action template you are using to construct your action, you should not need to modify them.

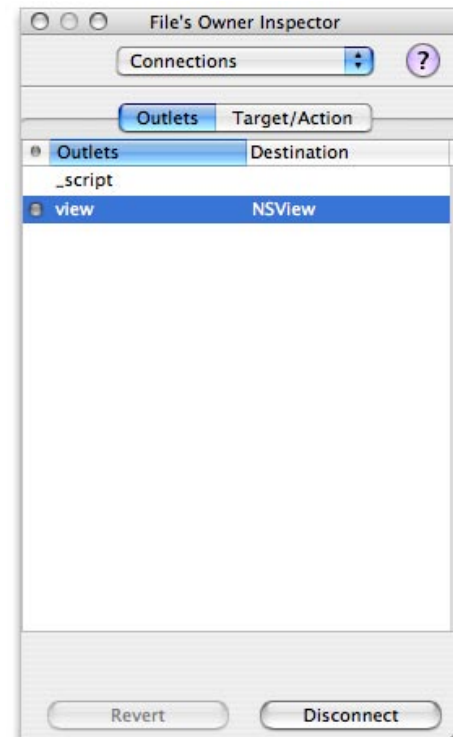


Figure 13.7 File's Owner Connection Configuration for an Action

View Instance

A view (*NSView*) instance in the main *nib* is also provided by default. This view instance will be the basis for the action's settings interface, which will be displayed in the Automator application during a workflow configuration, or when the action is run, if allowed. The presence of this view assumes that your action will require a settings interface. If your action will not require any settings to be displayed to the user,



Chapter 13:
**Constructing
an Action's
Interface**

then you can delete the view instance from the *nib* file.

Adding Interface Elements to the Action's View

Assuming that your action will have user-modifiable settings, you will need to add them to the view. To display the view's window, double click on the view instance in the *nib*'s main window. See figure 13.8.

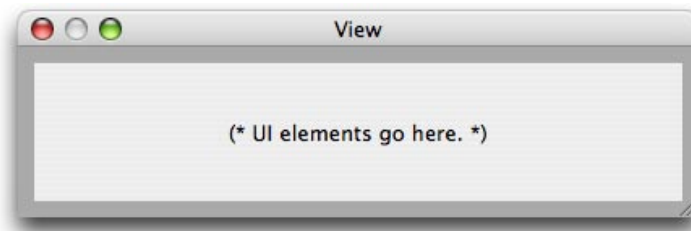


Figure 13.8 *Default Action View*

The default view contains a single text field indicating where interface elements should be placed within the view. This initial text field can safely be deleted. Now, you may proceed to design the view's interface by dragging and dropping any desired user interface elements into the view's window from the palette window, and arranging them as desired.

Interface Design Guidelines

Apple's *Aqua Human Interface Guidelines*, which can be found in the *Apple Developer Connection's* reference library, provide developers with detailed information regarding the proper design of a user interface. These guidelines help developers to design a custom interface that is consistent with the look and feel of other interfaces throughout the Mac OS X environment.

Developing an interface for an action is no different. An action's interfaces should also adhere to Apple's interface design guidelines, helping actions to achieve a consistent look and feel between one another, as well as with interfaces throughout the entire operating system, to some degree. Of course, since an action's interface space is rather limited, some additional guidelines are necessary for designing an action's interface.

The following guidelines should be followed when developing interfaces for Automator actions:

- ▶ **General View Guidelines** – These guidelines pertain to general aspects of the action's view instance.
 - ▶ Ensure a 10-pixel margin of empty space around the action's interface.
 - ▶ If your interface elements do not take up the entire default window, then condense the window to eliminate empty space.
- ▶ **General Interface Element Guidelines** – These settings pertain to interface elements that are placed within an action's view.
 - ▶ When designing an interface, try not to use



Chapter 13:
**Constructing
an Action's
Interface**

unnecessary interface elements in the action's interface, such as boxes and tab views. These types of interface elements take up too much space within the limited space available for an action. In addition, an action's interface should be as simple as possible for a user to navigate. These types of elements add complexity.

- ▶ Ensure that the size of an action's interface element is set to *Small* in the *Inspector* palette. See figure 13.9.

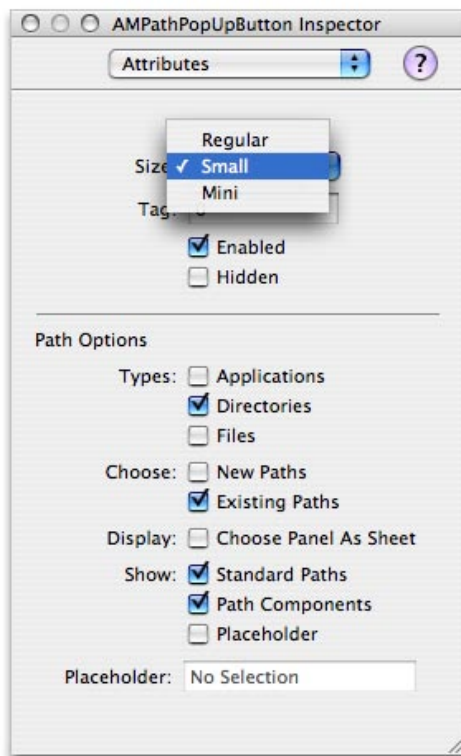


Figure 13.9 Setting the Size of an Interface Element to Small

- ▶ Keep any interface element labels as short as possible, and try not to duplicate information that is already contained within the title or description of the action.
- ▶ Consider using different types of interface elements to achieve the same result. For example, consider using a popup button when providing the user with a choice of settings, rather than using a radio button.
- ▶ General Interface Design Guidelines – These guidelines pertain to the general look and feel of the complete interface of the action.
 - ▶ Configure the action's interface to behave in a manner that might be expected by the user. For example, if you have multiple text fields within the interface, then establish connections between them, allowing the user to tab from field to field using the keyboard. See figure 13.10.



Chapter 13:
**Constructing
an Action's
Interface**

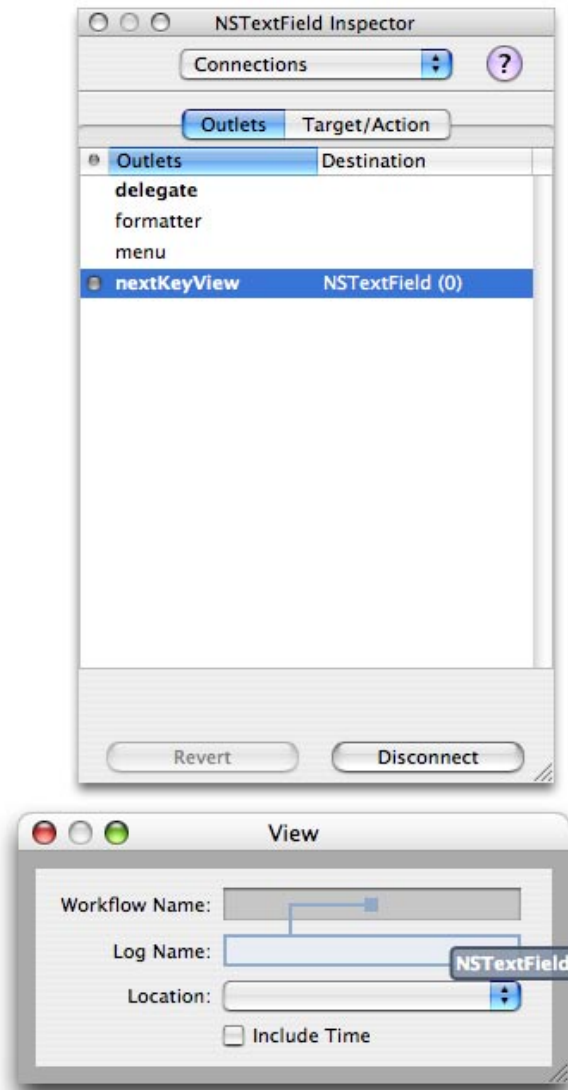


Figure 13.10 Example of Linked Text Fields

- Provide examples of an action's behavior wherever possible, without cluttering your interface. For example, if your action will allow the user to select a prefix for an output file name, then display an example of what a file name might look like once the prefix has been applied. See figure 13.11.

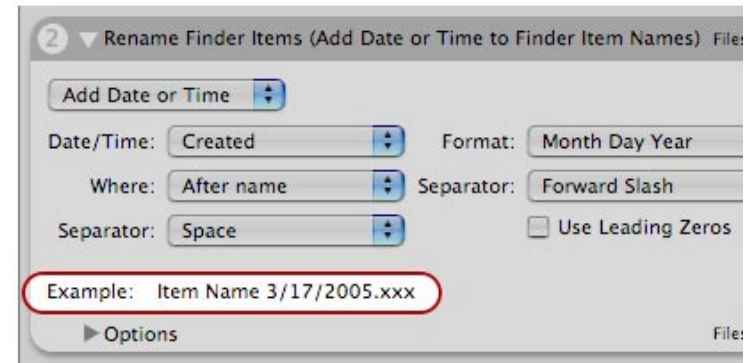


Figure 13.11 Displaying an Example of an Action's Behavior

- Provide feedback to the user when interface content is being loaded. For example, if your action contains a popup button of Mail email accounts, it might take a few seconds to retrieve the list of accounts from Mail and populate the popup button. Rather than leave the user wondering what is happening while this task is occurring, display a progress spinner indicating that some type of processing is occurring. See figure 13.12.



Chapter 13: Constructing an Action's Interface

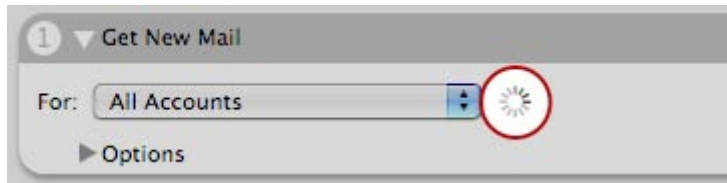


Figure 13.12 *Providing Feedback to the User for Lengthy Tasks*

- Overall, don't forget to ensure that the overall design and behavior of the interface adheres to the *Aqua Human Interface Guidelines* provided by Apple.

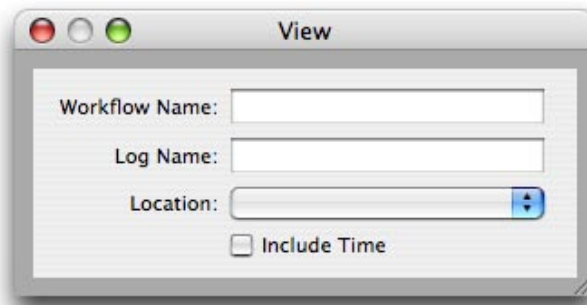


Figure 13.13 *An Example of a Custom Designed Action View*

Making sure that your Automator actions adhere to the guidelines we just discussed will help to ensure that your actions are visually appealing, while remaining as user friendly as possible.

Grouping Interface Elements

You may recall that in Automator's interface, certain actions will allow a user to configure that action to display its settings interface during processing, allowing the user to make adjustments at that time. In addition, some of these actions may even be configured to display only specific interface elements during processing. See figure 13.14.

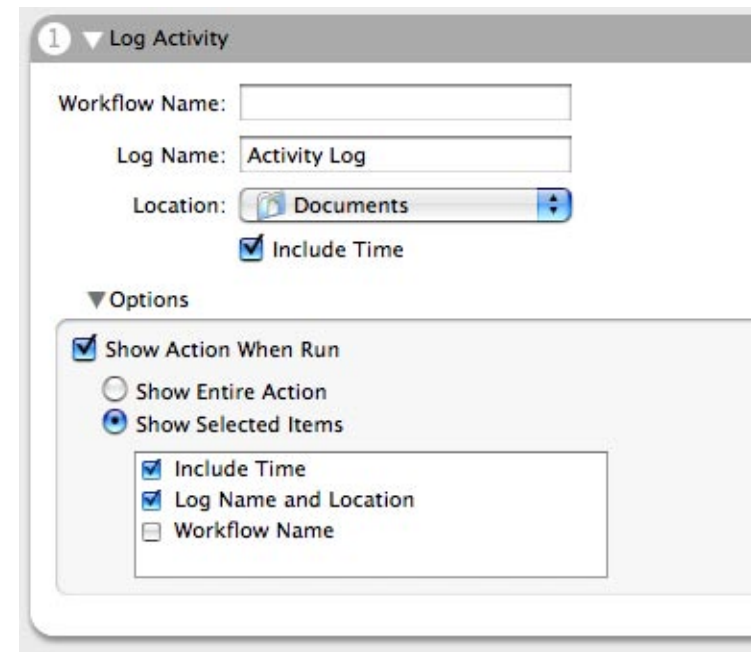


Figure 13.14 *Custom Settings Display Configuration*

As we discussed in the last chapter, the ability to enable and disable this functionality is handled by modifying key values within the action's *info.plist* file. However, when designing the action's interface, we can group



Chapter 13: Constructing an Action's Interface

certain interface elements together, allowing them to be listed as a single item in the action configuration options.

To do this, select the interface elements that you want to group together in the action's view. Next, select *Make subviews of > Automator Box* from the *Layout* menu in the menu bar. See figure 13.15.

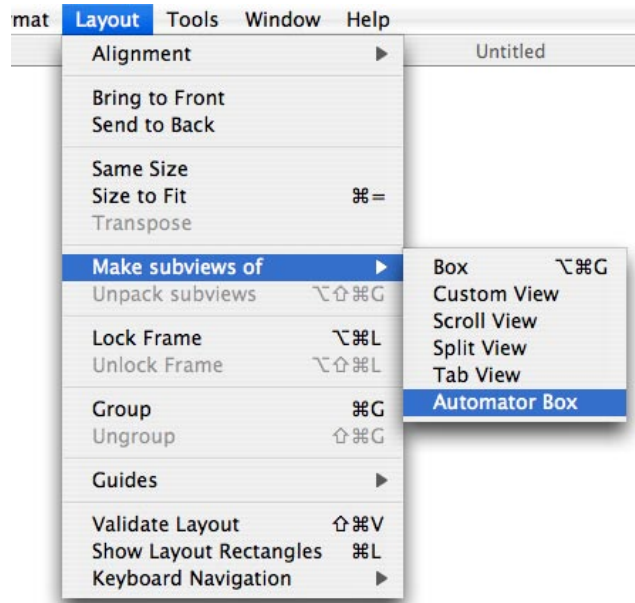


Figure 13.15 Grouping Interface Elements

The interface elements will now be grouped together, and you may assign a title for the grouped elements under *Attributes* in the *Inspector* palette. See figure 13.16. Please note that the title specified for the group will be displayed in the action's options within Automator, if the action is configured to allow the user

to show specified interface elements during workflow processing. See figure 13.14 again.

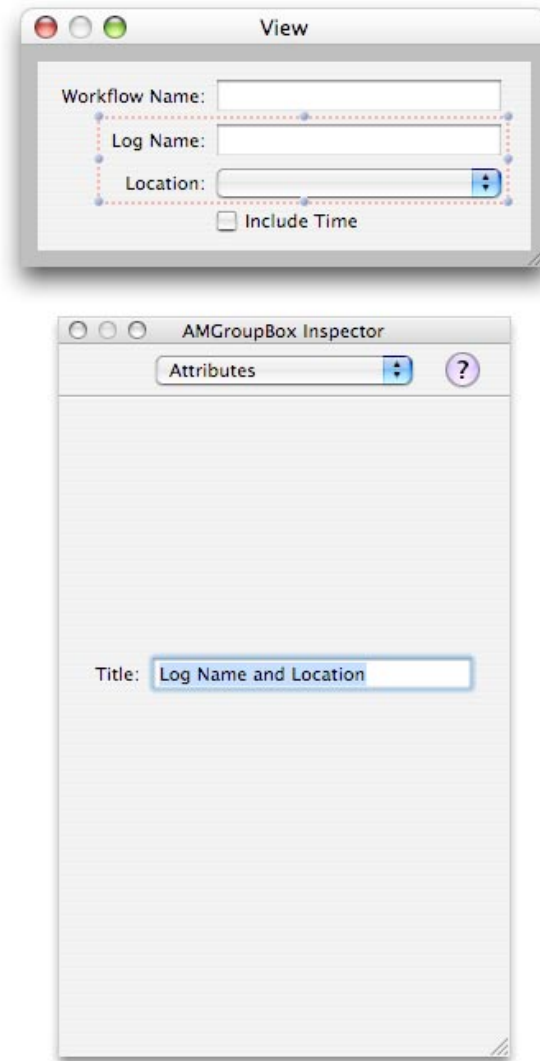


Figure 13.16 Naming an Interface Element Group



Chapter 13:
**Constructing
an Action's
Interface**

If you decide for some reason that you do not want interface elements to be grouped, you may ungroup them by selecting the group in the action's view, and then choosing *Unpack subviews* from the *Layout* menu.

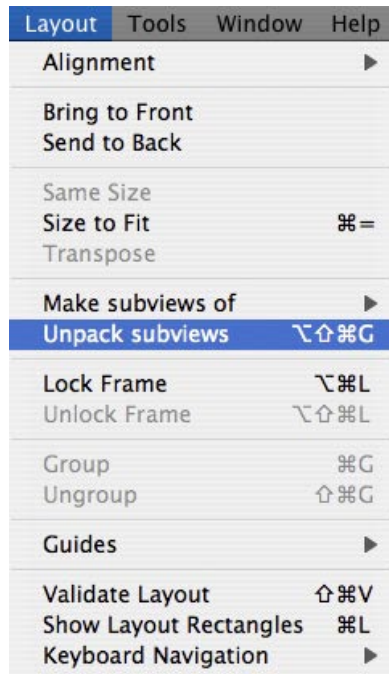


Figure 13.17 Ungrouping an Interface Element Group

What's Next

Now that we have discussed the construction of your action's interface, we still need to discuss how to link the values specified by the user back to your action's code. This will be covered in the next chapter.



Chapter 14 Retrieving an Action's Settings

nce you have constructed the settings interface for your Automator action,

you will need some way to retrieve the values entered for those settings by the user. Fortunately, Cocoa offers a mechanism, called Cocoa bindings, which makes this retrieval process quite simple.

A Cocoa binding is, essentially, a link between an interface element and a chunk of your project's code. By establishing a binding, you eliminated the need to write code to retrieve the value of that interface element. Instead, when the value of the interface element changes, Cocoa automatically calls the bound code for you.

For example, you might create a Cocoa binding between a popup menu and the code within your project that responds to that popup menu. When the user makes a selection from the popup menu, Cocoa automatically calls the chunk of code bound to that element, passing in the new value.

Establishing Interface Element Bindings

There are several steps involved in establishing Cocoa bindings between interface elements and an action's code. First, you will need to establish the bindings within the action's *nib* in Interface Builder. Then, you will need to reference the bindings in the project's *info.plist* file. We will discuss the Interface Builder aspects of assigning Cocoa bindings first.

Assigning Interface Parameter Keys

The first step necessary in order to establish bindings is to create **parameter keys** for the settings that you wish to access from your code. These parameter keys will later be linked to attributes of interface elements, and will then be referred to in your action's *info.plist* file and its code. To create interface parameter keys, perform the following steps:



Chapter 14: Retrieving an Action's Settings

- Click on the *Parameters* instance in the main *nib* window. See figure 14.1.

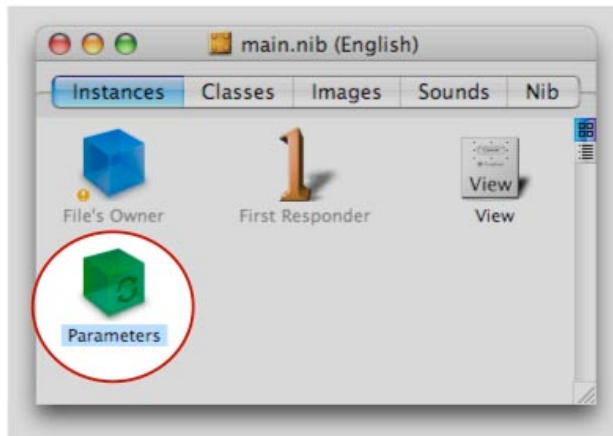


Figure 14.1 The *Parameters* Instance in the Main Nib Window

- Select *Attributes* from the popup button at the top of the *Inspector* palette. See figure 14.2.
- In the *Inspector* palette, click the *Add* button for each parameter key that you wish to add. You should add a parameter key for every interface attribute that you wish to be passed to your action's code.

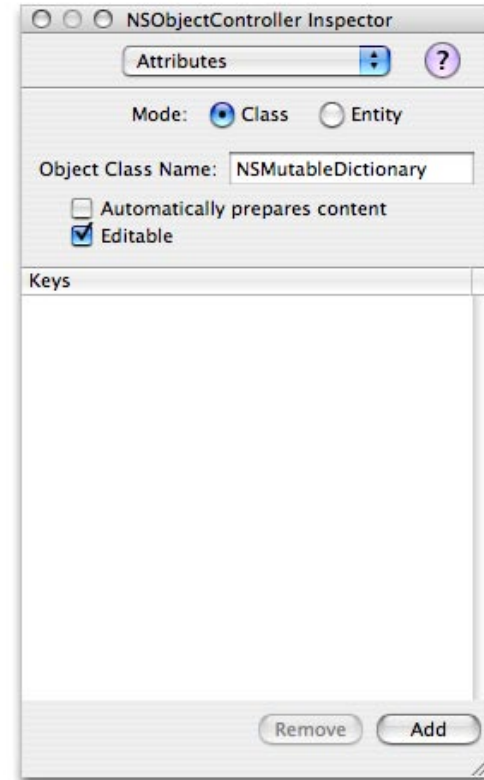


Figure 14.2 The *Attributes Inspector* for *Parameters*

- Double click on each added parameter key in the *Inspector* palette, and modify its name as desired. The names that you specify for the parameters here will be used to access the bound interface element attributes from within your code. So, you should be sure to specify parameter names that are descriptive relative to the value they are bound to. See figure 14.3.



Chapter 14: Retrieving an Action's Settings

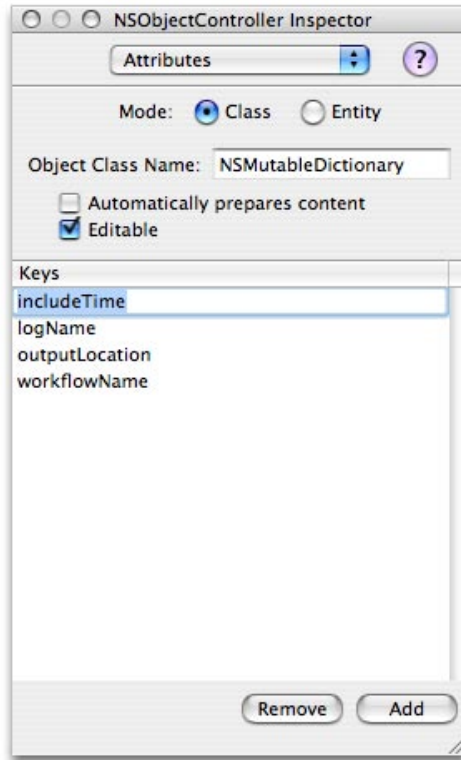


Figure 14.3 *Modifying Parameter Keys*

Linking Interface Elements to Parameters

The next step in configuring Cocoa bindings for the settings in your interface is to bind the parameter keys that you configured to the desired attributes of your interface's settings elements. To do this, perform the following tasks:

- ▶ Select an interface element in the interface's main view.

- ▶ Select *Bindings* from the popup button at the top of the *Inspector* palette. Please note that the bindings options displayed will vary, depending on the type of interface element that you have selected. Figure 14.4 shows an example of the initial bindings options for a text field.

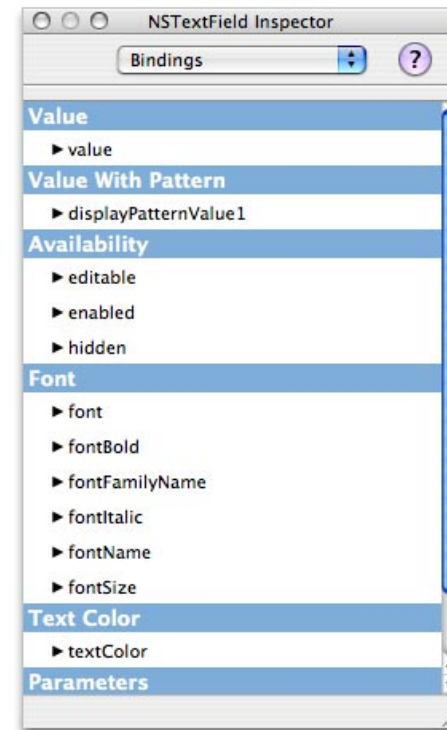


Figure 14.4 *An Text Field's Bindings*



Chapter 14: Retrieving an Action's Settings

- Locate and expand the desired interface element attribute in the *Inspector* palette. Figure 14.5 shows the expanded binding options for the *value* attribute of a text field. Again, different interface elements will possess different attributes, so you should select an attribute relative to the type of information you want to retrieve from the interface element. Because the interface element that I have selected is a text field, the value of the text field will represent the text entered by the user.

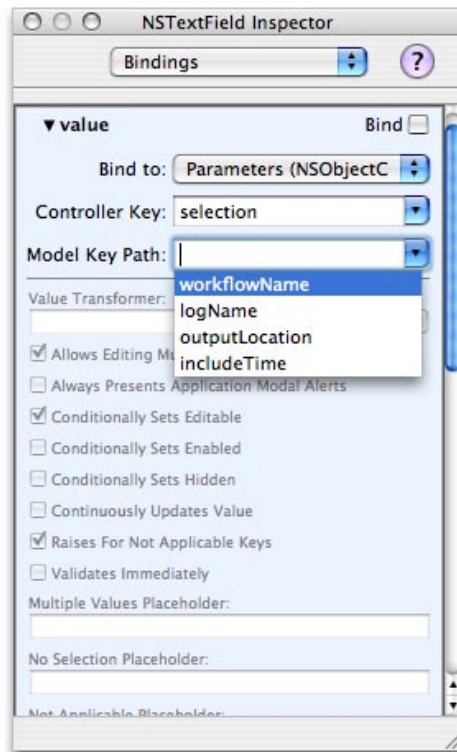


Figure 14.5 Configuring A Text Field's Value Binding

- From the *Model Key Path* field in the interface element's attribute binding options, select the desired parameter that will be used to contain the value of the specified attribute. Figure 14.6 displays an example binding for a text field. In this example, the value of the text field is bound to the parameter *workflowName*, which will be accessible within the code of the action.

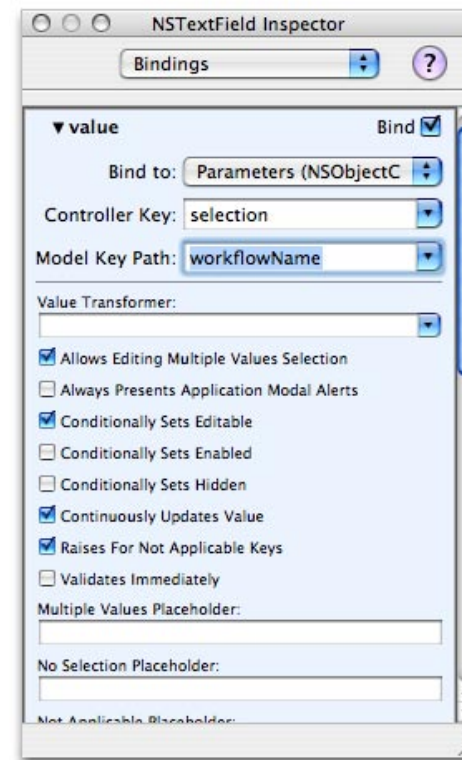


Figure 14.6 A Configured Text Field's Value Binding



Chapter 14: Retrieving an Action's Settings

- ▶ Make any additional adjustments to the bindings options in the *Inspector* palette. For example, for a text field's value binding, you may wish to select the *Continually Updates Value* checkbox, to ensure that the text field's value is continually updated as the user enters new values in the field.

Again, since different types of interface elements will have different attributes, the options for assigning bindings to an element will vary slightly between interface elements. You should be sure to assign bindings to attributes of all of the interface elements in your action's settings interface you want to access from your code.

For additional information about Cocoa bindings, please refer to the *Introduction to Cocoa Bindings* reference documentation that can be found in the *Apple Developer Connection Reference Library*.

For *path chooser* interface elements available through the *Automator* palette in Interface Builder, the *path* attribute should be used as the binding attribute. For example, see figure 14.7. This will ensure that the path to the item selected by the user is passed to your action's code.

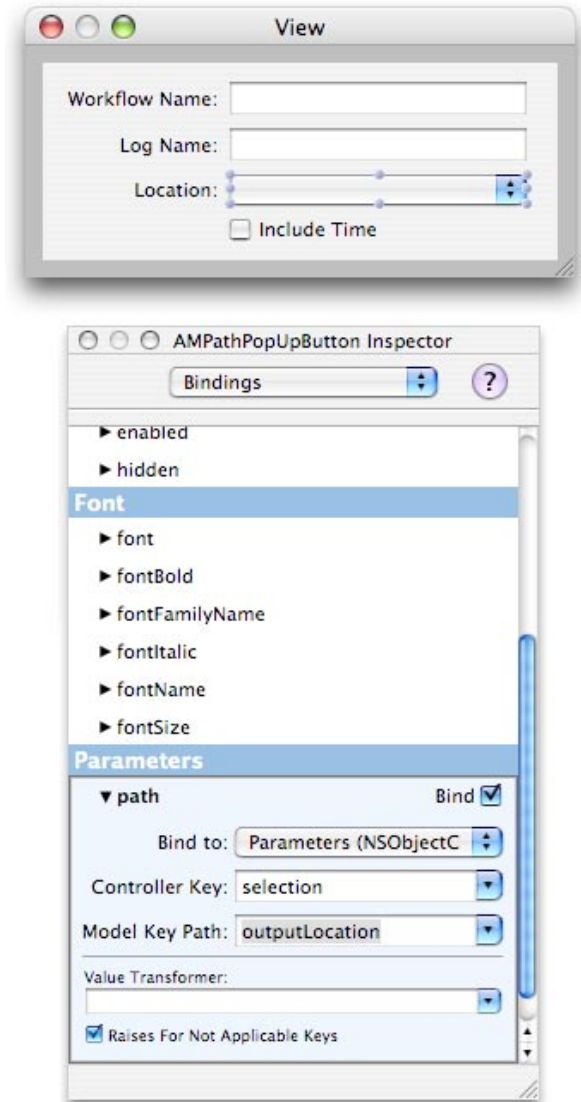


Figure 14.7 A Bound Automator Path Chooser Interface Element



Chapter 14: Retrieving an Action's Settings

Linking Parameters to the Action's Code

Once you have assigned bindings to the desired attributes of the elements within your settings interface, you will need to make some adjustments to the *info.plist* file in order for those values to be synchronized with your project code. To do this, go back into Xcode, and open the *info.plist* file.

In the *info.plist* file, you will need to modify the value of the **AMDefaultParameters** key. This property should represent an XML dictionary of keys and values for each parameter key that you specified in Interface Builder. Within this dictionary, a key and value should exist for each parameter key you assigned. In the **AMDefaultParameters** key, each value will represent the default value you would like to be applied to the bound attribute of the corresponding interface element. For example, a value for a key linked to a checkbox would contain a true or false value indicating whether the checkbox should be enabled by default when the action's interface is displayed.

The following is an example of a properly configured **AMDefaultParameters** key in an *info.plist* file.

```
<key>AMDefaultParameters</key>
<dict>
    <key>workflowName</key>
    <string></string>
    <key>logName</key>
    <string>Activity Log</string>
    <key>outputLocation</key>
    <string>~/Documents</string>
    <key>includeTime</key>
    <true/>
</dict>
```

Figure 14.8 shows an example of how the interface of an action uses the key values specified in the **AMDefaultParameters** key in the action's *info.plist* file.

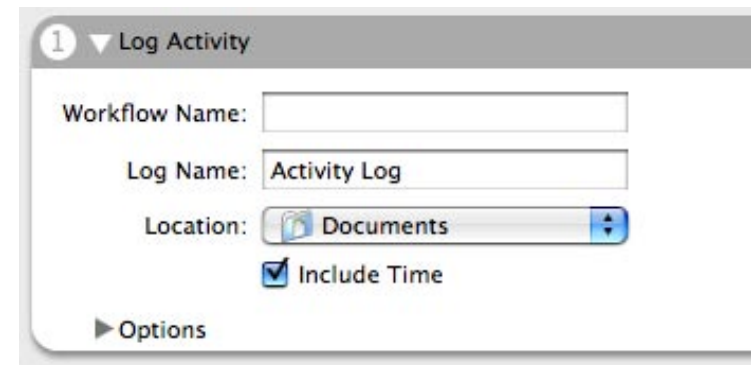


Figure 14.8 Default Parameter Values Displayed in an Action's Interface



Chapter 14:
**Retrieving
an Action's
Settings**

What's Next

The steps that we discussed in this chapter will ensure that the desired interface element attribute values will be properly bound from the action's interface to the code in your action. We still need to discuss how the values of these parameters can be used during the processing of the action's code. This will be covered in the next chapter. It is also possible to access the attribute values of interface elements that have not had Cocoa bindings assigned. This will also be discussed in the next chapter.



Chapter 15 Adding Code to an Action

We have now discussed a number of tasks involved in the creation of a custom Automator action. We have covered building the action project, updating the action's *info.plist* file, constructing the interface, and creating interface element attribute bindings. However, you may have noticed that we have yet to discuss one important aspect of action development, adding processing code to your action. It is finally time to begin this stage of the action development process. Throughout this chapter, we will discuss the steps involved in adding processing code to both an AppleScript-based action and a Cocoa, Objective-C-based action.

As mentioned earlier in this book, AppleScript and Objective-C will not be taught in this chapter. Rather, this chapter is meant to serve as a set of guidelines for adding processing code into a custom action. This chapter assumes that you are already familiar with either AppleScript or Objective-C. If you are not familiar with these languages, then please refer to chapter 18 for a list of recommended resources for continued learning.

Action Processing Code Overview

Before you start writing code, it is important to understand the primary functionality and requirements for the code that you will write.

The processing code within your action will be triggered when the action is run from within a workflow. Regardless of whether your action is an AppleScript-based action or an Objective-C-based action, this processing code will be responsible for a few primary tasks.

First, the processing code in every action that you create will need to be written to accept data as input. Typically, this input will normally be received in the form of a list (or array). Because of this, it is probably safe to assume that your action's code should always be written to accept data in this format. This will help to ensure consistency between your action and other actions.

If your action will perform some type of processing on its input data, then your code should also be written to loop through the list of input data and process each passed value individually. As we have discussed, some



Chapter 15:
**Adding Code
to an Action**

actions may be configured by users to ignore their input value. If your action supports this functionality, then you should also add processing code to handle this type of situation, should it be encountered.

Your action's processing code should also be written to return a value once processing is complete. This value will be passed on to the next action in a workflow sequence. Like the input value, an action's output value should be returned in the form of a list. An output value might contain the modified data that has been processed by the action. If you do not want your action to return a specific value, then you should configure the action to return its input value as its output value. This will allow a user to continue processing the data originally passed to your action, if desired.

Adding Code to an AppleScript Action

An AppleScript-based Automator action requires the presence of a *main.applescript* file. This file is automatically included in a new AppleScript-based action, and will contain the code that will be triggered when the action is run within a workflow. By default, a new action will already contain the basic code structure necessary for the action to run. This code may be modified as needed.

On Run Handler

In order for an AppleScript-based action to run successfully, it must contain a properly configured **on run** handler within its *main.applescript* file. The default implementation of this handler, included in a new AppleScript-based action project, appears as follows:

```
on run {input, parameters}
    return input
end run
```

To implement processing code into your action, add the desired AppleScript code into the run handler. If desired, you may implement additional handlers, and call them from the code in the run handler.

As you can see from the default code in a new action, the run handler must contain two positional parameters. By default, these parameters are labeled **input** and **parameters**. However, a handler's parameters are essentially variables. Therefore, the



Chapter 15: Adding Code to an Action

labels for these parameters may be changed, if desired, so long as you are sure to reference those parameters by the appropriate labels throughout the handler. For example, the following run handler would perform in the exact same manner as the default run handler previously specified.

```
on run {actionInput, userSettings}
    return actionInput
end run
```

Input Parameter

When your action is triggered in a workflow, the run handler will be invoked. At that time, the first parameter in the handler, **input**, will contain the input value for your action. Typically, this value will be received in list format, although you may want to check the class of the parameter in order to verify this during processing. For example, the following code could be used to verify that the input parameter is a list, coercing it to a list if it is not.

```
if (class of input) is not equal to list then
    set input to {input}
```

Parameters Parameter

When the run handler in your action is invoked, the second parameter, **parameters**, will contain an AppleScript record. This AppleScript record will be formatted in the following manner:

```
{|ignoresInput|: false, |parameterName1|: value,
|parameterName2|:value, ...}
```

The **parameters** record will contain a property indicating whether the action's input should be ignored. It will also contain properties and values for any interface parameters specified for the **AMDefaultParameters** key in your action's *info.plist* file. Each property name in the record will be surrounded by pipe characters, and must be referred to in this manner. For example, the following code demonstrates how you would retrieve four different interface parameter values:

```
set workflowName to |workflowName| of parameters
set logName to |logName| of parameters
set outputLocation to |outputLocation| of
parameters
set includeTime to |includeTime| of parameters
```

When referring to a property in the **parameters** record, be aware that the pipe characters are required, and that the property name is also case sensitive. So, for example, if the property name is **|workflowName|**, you may not refer to it as **|workflowname|**.

To determine whether a user has configured the action to ignore its input, you may access the **|ignoresInput|** property in the **parameters** record. For example:

```
if |ignoresInput| of parameters = true then
    -- Do something
end if
```




Chapter 15: Adding Code to an Action

An action will only be able to ignore its input if the **Optional** key in the **AMAccepts** key's dictionary in your action's *info.plist* file is set to a value of **true**. If the value of this key is set to **false**, then you do not need to determine the value of the `| ignoresInput |` parameter, as the input will never be ignored.

The `| ignoresInput |` property in the **parameters** record is really for informational purposes, though, as if the user configures the action to ignore its input, Automator will automatically strip out the input values, and pass an empty list to the action as input.

Processing the Input

Since an action will typically be written to process a list of input data, your action should, in most cases, be written to loop through this data. For example, the following code would loop through a list of input, setting the variable **currentData** to the current input value during each loop.

```
repeat with a from 1 to length of input
    set currentData to item a of input
    -- Process "currentData"
end repeat
```

If your action will process only a specific type of files, then you may want to configure your action's processing code to loop through the list of input paths when the run handler is first triggered. This way, you can verify that the input paths match a required file type or extension, prior to beginning any actual processing.

Return Value

As we have discussed, every action should return a value, to be processed by the next action in the workflow. In an AppleScript-based action, this would be done in the following manner:

```
return outputValue
```

If your action does not require any specific output value to be returned, then you should simply return the input value of the action. The default run handler implementation in a new action will do this for you already. However, the following method demonstrates a better way to do this. In the following example code, the value of the **input** parameter is copied to a new variable, named **output**, as soon as the run handler is triggered. This **output** variable is then returned once the handler is done processing. By copying the value of the **input** parameter to a new variable, it will ensure that the original value is retained and returned once processing is complete, even if your action's code proceeds to modify the value in the **input** parameter during processing.

```
on run {input, parameters}
    copy input to outputValue
    -- Process the input data
    return outputValue
end run
```



Chapter 15:
**Adding Code
to an Action**

Error Reporting

If an error occurs during processing of your action, you may generate an error message to indicate to Automator that an error has occurred. This will cause Automator to stop the workflow from processing further. To generate an error, you should use the following syntax:

```
error errorDescription number errorNumber
```

For example:

```
error "Could not copy the file to the output  
folder because there is already a file there  
with the same name." number -15267
```

When returning an error, Automator will display an alert to the user, indicating that an AppleScript error occurred during processing, as well as the text of your error message. To generate an error without causing Automator to display this alert, then you may use error number -128. This error number signifies to Automator that a user has cancelled the process. For example:

```
error number -128
```

Triggering AppleScript Code From the Interface

Like any AppleScript Studio project, it is possible to trigger AppleScript code from within the interface of an Automator action. For example, you could configure an action to retrieve settings from an application, prior

to displaying the interface. Or, you could configure a button on the action's settings interface so that, when clicked, it would trigger AppleScript code to perform some type of processing. See figure 15.1.

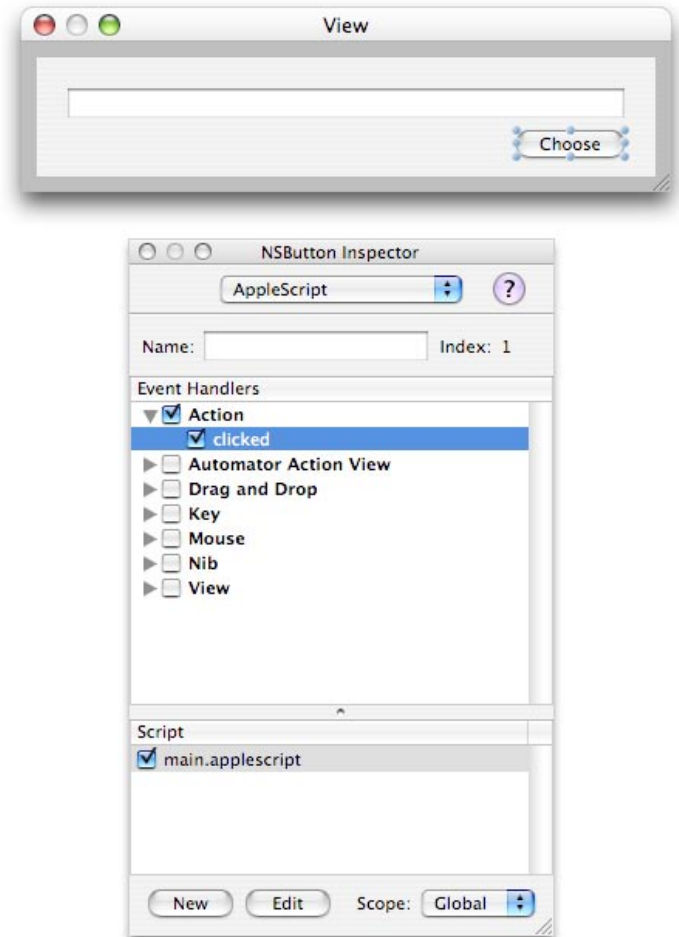


Figure 15.1 Assigning an AppleScript Studio Handler to an Interface Element



Chapter 15:
**Adding Code
to an Action**

The basic structure for the AppleScript Studio handler that would be created, based on the connected *clicked* event shown in figure 15.1, would appear as follows:

```
on clicked theObject
    -- Add your choose file or folder code here
end clicked
```

An AppleScript Studio handler may be connected to the action's *main.applescript* file, or it may be connected to another AppleScript file that has been added to the action project. Just like any other AppleScript Studio project, additional script files may be added to an action project, although the *main.applescript* file must contain the run handler that will be triggered when the action is run.

Automator Event Handlers

You may have noticed, when working in Interface Builder, that the event handlers available in the *Inspector* palette for an action's view include event handlers that are specific to Automator. These event handlers, displayed beneath the *Automator Action View* category in the *Event Handlers* table, are not available in other types of AppleScript Studio projects. See figure 15.2.

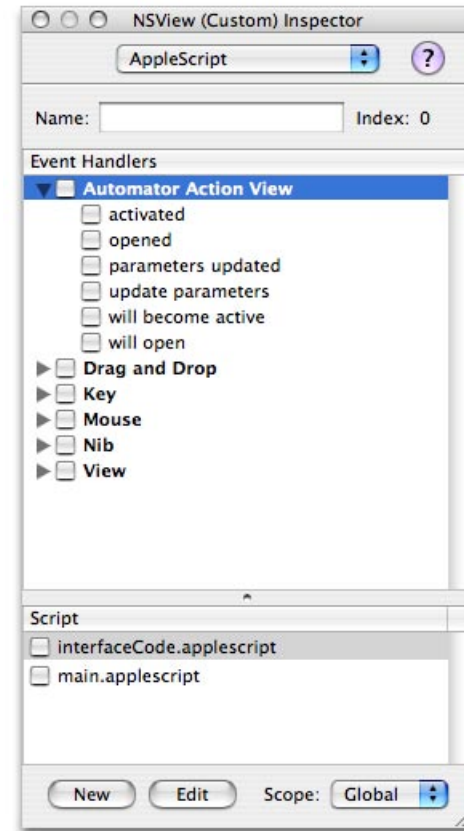


Figure 15.2 Automator Action Event Handlers

The following Automator event handlers may be used, if desired, to trigger AppleScript code when an action's interface is opened or activated in a workflow.

- ▶ *activated*
- ▶ *opened*
- ▶ *will become active*
- ▶ *will open*



Chapter 15:
**Adding Code
to an Action**

Parameter-Related Event Handlers

In an AppleScript-based action, it is also possible to configure the action to use AppleScript to interact with the parameters of the action, rather than using Cocoa bindings. To do this, you may utilize the following AppleScript Studio event handlers in your project:

- ▶ **parameters updated** – When enabled, this handler will be called whenever the parameters of the action are updated, such as when the action is loaded and displayed in a workflow, or when the action's interface is displayed during processing of a workflow. This handler may be used to update the interface of the action, based on the current values of the parameters when the handler is triggered.
- ▶ **update parameters** – When enabled, this handler will be called immediately before the processing code of the action is triggered. When called, the handler may be used to retrieve interface element attribute values, and insert those into the parameters record. This handler must return the parameters record, once it is finished executing.

Please note that regardless of whether AppleScript event handlers or Cocoa bindings are used to update the parameters record, you must still add parameter keys to the *parameters* instance in your action's *nib*, and you must also add the parameters to the **AMDefaultParameters** key in the action's *info.plist* file.

The following code is an example of the correct usage of the **parameters updated** and **update parameters** event handlers in an AppleScript-based action. This example is based on the *Log Activity* example action that will be created in chapter 17.

```
global interfaceReference

on awake from nib theObject
    set interfaceReference to theObject
end awake from nib

on parameters updated theObject parameters
    theParameters
        set content of text field "workflowName" of
            interfaceReference to |workflowName| of
                theParameters
        set content of text field "logName" of
            interfaceReference to |logName| of
                theParameters
        set title of popup button "outputLocation"
            of interfaceReference to |outputLocation| of
                theParameters
        set integer value of button "includeTime" of
            interfaceReference to |includeTime| of
                theParameters
    end parameters updated

on update parameters theObject parameters
    theParameters
        set |workflowName| of theParameters to
            content of text field "workflowName" of
                interfaceReference
        set |logName| of theParameters to content of
            text field "logName" of interfaceReference
        set |outputLocation| of theParameters to
            title of popup button "outputLocation" of
                interfaceReference
```



Chapter 15: Adding Code to an Action

```
set |includeTime| of theParameters to  
integer value of button "includeTime" of  
interfaceReference  
return theParameters  
end update parameters
```

Figure 15.3 shows an example of the AppleScript Studio event handler connections within Interface Builder for the previous code.

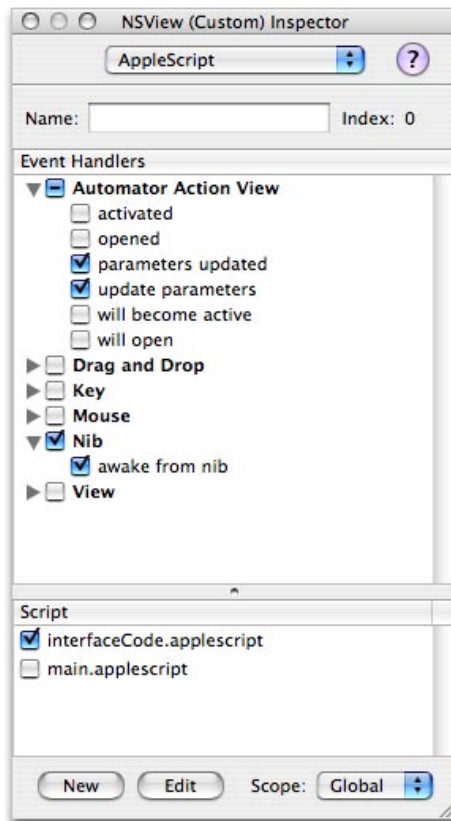


Figure 15.3 Parameter Event Handler Connections

Adding Code to a Cocoa (Objective-C) Action

A Cocoa Automator action, by default, includes both a *projectName.h* file and *projectName.m* file. Like the *main.applescript* file in an AppleScript-based action, these files will contain the primary processing code of your action, although additional components may be added to the project, if desired.

We will now discuss adding processing code to a Cocoa action. In addition, chapter 17 will include sample code for creating a Cocoa Automator action.

Frameworks

By default, a Cocoa Automator action will include links to all of the frameworks required in order for the action to function. The following linked frameworks will be included in a new Cocoa action project:

- ▶ *Cocoa.framework*
- ▶ *Automator.framework*

Links to additional frameworks may be manually established in your action project, if deemed necessary for your action to include the desired functionality. For example, if your action will interact with the Core Image framework, then you will need to add a link to the *QuartzCore.framework* framework to your action project.

Automator Classes

The *Automator.framework*, which is included in every Cocoa action, implements the following public classes,



Chapter 15:
**Adding Code
to an Action**

which may be accessed by your Objective-C code:

- ▶ **AMAction** – This class, an abstract subclass of **NSObject**, is responsible for defining the core interface and characteristics of an Automator action. **AMAction** declares the following methods:

- ▶ **activated** – This method is triggered whenever a workflow window containing an action is brought to the front.
 - (void)activated
- ▶ **definition** – This method is triggered to retrieve a mutable dictionary containing the parameters of an action.
 - (NSMutableDictionary *)definition
- ▶ **initWithDefinition:fromArchive:**
 - This method is triggered whenever Automator loads or unarchives an action's bundle.
 - initWithDefinition:(NSDictionary *)dict fromArchive:(BOOL)archived
- ▶ **opened** – This method is triggered whenever an action is first added to a workflow, and should be used to perform any initialization tasks, rather than performing them in the **awakeFromNib** method.
 - (void)opened

- ▶ **parametersUpdated** – This method is triggered whenever the action needs to update its settings interface parameters.
 - (void)parametersUpdated
- ▶ **reset** – This method is triggered to reset an action to its initial state, and release its output.
 - (void)reset
- ▶ **runWithInput:fromAction:error:**
 - This method is triggered to perform the main processing of the action. Additional information about this method will be described shortly.
 - (id)runWithInput:(id)input fromAction:(AMAction *)anAction error:(NSMutableDictionary **)errorInfo
- ▶ **stop** – This method is triggered to stop an action from running in a workflow.
 - (void)stop
- ▶ **updateParameters** – This method is triggered when an action's interface is dismissed, and its settings need to be stored as parameters. This occurs whenever the action is copied, run, or saved.
 - (void)updateParameters
- ▶ **writeToDictionary** – This method is triggered when an action writes information to the parameters dictionary.
 - (void)writeToDictionary:(NSMutableDictionary *)dictionary



Chapter 15:
**Adding Code
to an Action**

► **AMBundleAction** – This class, a concrete subclass of **AMAction**, is responsible for implementing a Cocoa action as a loaded bundle. An **AMBundleAction** declares the following methods:

► **awakeFromBundle** – This method is triggered when an action is first unarchived from its bundle.

- (void)awakeFromBundle

► **bundle** – This method will return the bundle object for the action.

- (NSBundle *)bundle

► **hasView** – This method will return a boolean value indicating whether the action possesses a view to be displayed.

- (BOOL)hasView

► **initWithDefinition:fromArchive** – This method is invoked to initialize and return the **AMBundleAction** object.

- (id)initWithDefinition:(NSDictionary *)dict fromArchive:(BOOL)archived

► **parameters** – This method will return the parameters of the action, as specified by the user in the action's interface.

- (NSMutableDictionary *)parameters

► **setParameters** – This method may be used to manually set the parameters of an action. This is not necessary in actions that employ Cocoa bindings.

- (void)setParameters:(NSMutableDictionary *)newParameters

► **view** – This method will return an action's view object.

- (NSView *)view

► **AMAppleScriptAction** – This class, a concrete subclass of **AMAction**, is responsible for implementing an AppleScript-based action as a loadable bundle. An invisible instance of an **AMAppleScriptAction** is included in an AppleScript-based action project. An **AMAppleScriptAction** declares the following methods:

► **script** – This method will return the **OSAScript** object representing the action's AppleScript file containing a run handler.

- (OSAScript *)script

► **setScript:** – This method will set an action's main script to a specified script file. The specified script file must contain a run handler.

- (void)setScript:(OSAScript *)newScript



Chapter 15: Adding Code to an Action

Detailed information about the classes we just discussed, including a complete overview of their methods, can be found in the *Automator Objective-C Reference* document, included in the *Apple Developer Connection Resource Library*.

projectName.h File

In a new Cocoa action project, the *projectName.h* header file has already been preconfigured with a basic implementation necessary for your action to function. This implementation includes imports for any frameworks referenced by default, which in the case of a new action, include the *Cocoa.framework* and the *Automator.framework*. Additional framework imports may be added to the header file as needed, if you are referencing constants or non-class data structures in your header file. In addition, the header file also contains an interface declaration for the action, which is inherited from the **AMBundleAction** class.

The following represents the basic implementation code in a new Cocoa action's Objective-C header file.

```
#import <Cocoa/Cocoa.h>
#import <Automator/AMBundleAction.h>

@interface projectName : AMBundleAction
{
}

- (id)runWithInput:(id)input fromAction:(AMAction *)anAction error:(NSDictionary **)errorInfo;

@end
```

projectName.m File

In a new Cocoa action project, the *projectName.m* Objective-C implementation file has also been preconfigured for you. By default, this file will contain an override implementation of the **runWithInput:fromAction:error:** method. This method is inherited from the **AMAction** class in the Automator framework, and will contain the processing code that will be run when your action is triggered. An implementation of this method is required in all Cocoa-based Automator actions.

The following represents the basic Objective-C implementation code in a new Cocoa action.

```
#import "projectName.h"

@implementation projectName

- (id)runWithInput:(id)input
  fromAction:(AMAction *)anAction
  error:(NSDictionary **)errorInfo
{
    // Add your code here, returning the data
    // to be passed to the next action.

    return input;
}

@end
```

Input and Output Values

Like AppleScript-based actions, Cocoa actions must also specify their input and output value types in the **AMAccepts** and **AMProvides** keys within the



Chapter 15: Adding Code to an Action

action's *info.plist* file. However, unlike AppleScript-based actions, the type identifiers that may be specified in these keys are somewhat limited. Cocoa actions may accept any of the following type identifiers as input and output values:

- ▶ `com.apple.cocoa.string`
- ▶ `com.apple.cocoa.path`
- ▶ `com.apple.cocoa.url`

Other type identifiers for use in Cocoa actions are not supported by Automator at this time, although Apple does plan to expand this list in the future.

When a Cocoa action is triggered in a workflow, the input value for the action is contained within the **`runWithInput:fromAction:error:`** method's **`input`** object. An action's input value is typically formatted as an array.

As we have discussed, once processing is complete, an action will need to return an output value to be passed to the next action in the workflow sequence. This output value should also be formatted as an array, and may refer to data that has been modified by the action. If your action does not need to output a specific value, then be sure to return the input value as the action's output. For example:

```
{
    id output = input;
    // Processing code
    return output;
}
```

Processing the Input

Most likely, your action will be performing some type of task on the data provided as input. Since the input for your action will typically be in the form of an array, your action's code should be written to loop through the input array, processing each input value. For example:

```
NSEnumerator *enumerator = [input
    objectEnumerator];
NSString *currentData;

while (currentData = [enumerator nextObject])
{
    // Process the Data
}
```

Depending on the value specified for the **`Optional`** key in the **`AMAccepts`** key's dictionary in your action's *info.plist* file, the user may be able to configure the action to ignore its input. If this is the case, and the user has configured the action to ignore its input, then Automator will automatically strip out the input value prior to triggering the action, passing the action an empty array as input. You will then need to test the input value to determine whether there is data to be processed.

Optionally, you may send an **`ignoresInput`** message to the previous action in the workflow, which is passed as the `anAction` parameter, in order to determine whether the action has been configured to ignore its output. This message will return a Boolean value to your action, indicating whether the input may be ignored.



Chapter 15:
**Adding Code
to an Action**

Error Reporting

If an error is encountered during the course of processing in the **runWithInput:fromAction:error:** method, your action will need to stop processing, and report back to Automator that an error occurred. You can do this by returning a dictionary containing the following keys and values:

- ▶ **NSAppleScriptErrorNumber** – This key should contain an **NSNumber** value specifying an error code as an integer.
- ▶ **NSAppleScriptErrorMessage** – This key should contain an **NSString** value providing a description of the error that occurred. For example:

```
NSArray *objArray = [NSArray arrayWithObjects:
    [NSNumber numberWithInt:errOSASystemError],
    [NSString stringWithFormat:@"ERROR: Could
    not successfully process document: %@\n",
    myFile], nil];
NSArray *keyArray = [NSArray
    arrayWithObjects:NSAppleScriptErrorNumber,
    NSAppleScriptErrorMessage, nil];
*errorInfo = [NSDictionary
    dictionaryWithObjects:objArray
    forKeys:keyArray];
return nil;
```

Alternatively, you may choose to substitute **OSAScriptErrorNumber** and **OSAScriptErrorMessage** for the above keys, although this will require that you add a link in your project to the *OSAKit.framework*. For example:

```
NSArray *objArray = [NSArray arrayWithObjects:
    [NSNumber numberWithInt:errOSASystemError],
    [NSString stringWithFormat:@"ERROR: Could
    not successfully process document: %@\n",
    myFile], nil];
NSArray *keyArray = [NSArray
    arrayWithObjects:OSAScriptErrorNumber,
    OSAScriptErrorMessage, nil];
*errorInfo = [NSDictionary
    dictionaryWithObjects:objArray
    forKeys:keyArray];
return nil;
```

Manually Updating Interface Parameters

Like an AppleScript-based action, a Cocoa action is not required to use Cocoa bindings to tie interface parameters to the code of the action. Instead, a Cocoa action may implement the **parametersUpdated** and **updateParameters** methods, which are triggered when an action is run or saved.



Chapter 15:
**Adding Code
to an Action**

Triggering Code From Other Languages

Regardless of the type of Automator action you create, it is possible to trigger code in other languages. This can allow you to make your action more robust, by extending its capabilities beyond what is possible in the core language.

Calling Cocoa from AppleScript

AppleScript is a great choice for interacting with scriptable applications or scriptable aspects of Mac OS X. However, it cannot directly interact with the system frameworks. To bypass this limitation, you can invoke the **call method** command in your AppleScript code. This command can be found in the *Application Suite* in AppleScript Studio's dictionary. The syntax for this command is as follows:

```
call method specifier : the object for the command
    [of any] : the object to send the method to
                  (exclusive from the "of class" parameter)
    [of class text] : the class to send the
method to (exclusive from the "of object"
parameter)
    [of object any] : deprecated in favor of the
"of" parameter
    [with parameter any] : a parameter to be
passed to the method (exclusive of "with
parameters")
    [with parameters list] : a list of
parameters to be passed to the method
(exclusive of "with parameter") Calling
AppleScript from Objective-C
```

command. This example will retrieve a reference to the action's bundle object, within a workflow:

```
call method "bundleWithIdentifier:" of class
"NSBundle" with parameter "«Action Bundle
Identifier»"
```

Calling UNIX from AppleScript

AppleScript actions may also be written to initiate UNIX commands using the **do shell script** command, found in the *Miscellaneous Commands* suite in the Standard Additions scripting addition. The syntax for this command is as follows:

```
do shell script string : the command or shell
script to execute. Examples are 'ls' or '/bin/
ps -auxwww'
    [as type class] : the desired type of
result; default is Unicode text (UTF-8)
    [administrator privileges boolean] : execute
the command as the administrator
    [user name string] : use this administrator
account to avoid a password dialog (If this
parameter is specified, the "password" parameter
must also be specified.)
    [password string] : use this administrator
password to avoid a password dialog
    [altering line endings boolean] : change all
line endings to Mac-style and trim a trailing
one (default true)
```

The following is an example of a **do shell script** command. This example will retrieve a list of connected volumes:



Chapter 15:
**Adding Code
to an Action**

```
do shell script "ls /Volumes"
```

Triggering AppleScript from Objective-C

Objective-C code can also be written to load and execute AppleScript code, if necessary. This would be useful if you are developing a Cocoa action, but want it to interact with an application that does not have a public API.

Classes for interacting with AppleScript code can be found in both the *OSAKit* and *Foundation* frameworks.

- ▶ *OSAKit* Framework Classes:
 - ▶ **OSALanguage**
 - ▶ **OSAScript**
 - ▶ **OSAScriptController**
 - ▶ **OSAScriptView**
- ▶ *Foundation* Framework AppleScript-related Classes:
 - ▶ **NSAppleEventDescriptor**
 - ▶ **NSAppleEventManager**
 - ▶ **NSAppleScript**

Conversion Actions

When triggering a workflow, as we have discussed, the output value of one action and the input value of the next action must match, in order for the workflow to successfully execute. However, in some cases, this simply isn't possible. To help alleviate the possibility of an error occurring in this type of situation, Automator may attempt to run a conversion action. A conversion action is a specially configured action, which, though loaded by Automator, does not appear in its interface. The purpose of a conversion action is to convert data from one type to another. During the processing of a workflow, if the output and input values of two actions do not match, Automator will try to find a conversion action for the necessary type of data conversion. If an action is found, then it will be invisibly inserted between the two mismatched actions.

When developing a custom action for Automator, you may encounter a scenario where your action requires a specific type of conversion action, which does not already exist. If this situation occurs, you may choose to create your own conversion action.

A conversion action is created in the same manner as a normal Automator action, with the following exceptions:

- ▶ You must change the extension of the action bundle generated when you build your action project from *.action* to *.caction*. This can be done by selecting the action's target in your project's *Files & Groups* list within Xcode. Next, type command + *i*, or select



Chapter 15:
**Adding Code
to an Action**

Get Info from the *Project* menu. Click the *Build* tab, and scroll down to the *Wrapper Extension* setting. Next, change the value for this setting to *caction*. See figure 15.4.

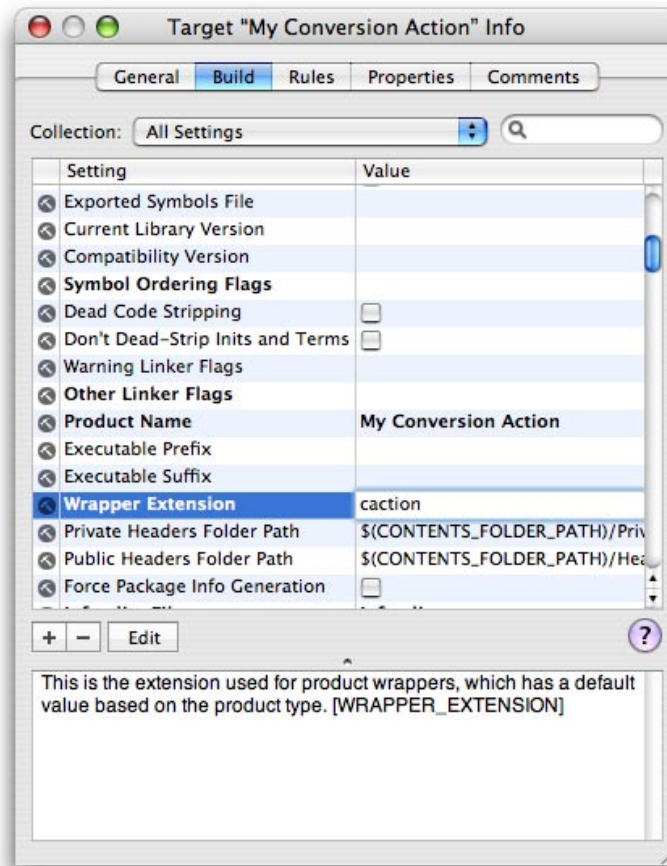


Figure 15.4 *Configuring a Conversion Action Extension*

- ▶ In the *info.plist* file for the action, set the value of the **AMAccepts** key to the type identifier for the type of value from which you want to convert.
- ▶ In the *info.plist* file for the action, set the value of the **AMProvides** key to the type identifier for the type of value to which you want to convert.
- ▶ In the *info.plist* file for the action, set the value of the **AMCategory** key to **Converter/Filter**.
- ▶ In the *info.plist* file for the action, set the value of the **AMApplication** key to **Automator**.

Once built, a conversion action should be installed into one of the *Library > Automator* folders in order to be loaded and recognized by the Automator application.

What's Next

As you can see, action development involves a number of steps, all brought together in the processing code of the action. In the next chapter, we will discuss the steps involved in conducting testing an action, a process that you should perform rigorously when developing any Automator action. Once we have discussed action testing, then we will walk through the process of creating a sample AppleScript and Cocoa action. At that time, all of the topics we have discussed will be brought together.



Chapter 16 Testing and Debugging an Action

Once you are finished writing your primary action's code, you will want to begin testing and debugging the action, making any necessary adjustments. In this chapter, we will discuss the typical steps involved in testing and debugging an action.

Build	Debug	Design	SCM	Window
Build Results				⇧ ⌘ B
Build				⌘ B
Build and Run				⌘ R
Build and Debug				⌘ Y
Clean				⇧ ⌘ K
Clean All Targets				
Next Build Warning or Error				⌘ =
Previous Build Warning or Error				⌘ +
Compile				⌘ K
Preprocess				
Show Assembly Code				

Figure 16.1 Xcode's Build Menu

Building and Running from within Xcode

When you have completed initial development on an action, you do not need to actually install the action in order to begin testing it. Instead, you can actually run the action from within Xcode. To do this, select *Build and Run* or *Build and Debug* from the *Build* menu (see figure 16.1), or click the appropriate button in the project's toolbar. See figure 16.2.

When one of these methods of triggering the action is invoked, a new instance of Automator will be launched, and the action will be loaded into Automator's list of actions. You may stop testing of the action by quitting the new instance of Automator, or by clicking the *Terminate* button in Xcode's *Run Log* window.



Chapter 16:
**Testing and
Debugging
an Action**

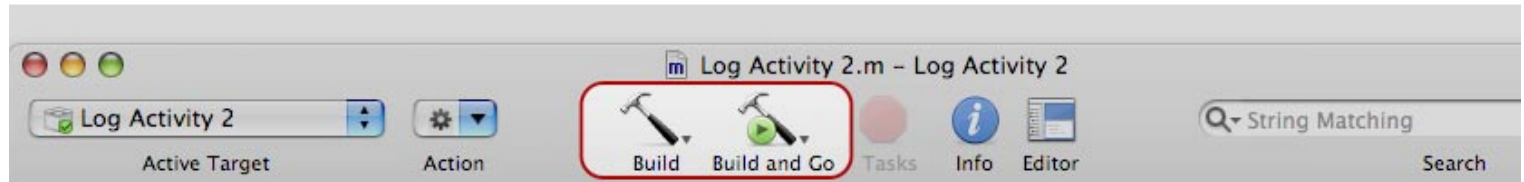


Figure 16.2 Xcode's Toolbar

Linking to the Automator Executable

For your action to be properly loaded within Automator, when run from within Xcode, you should first verify that the *Executable* of your project is properly configured. To do this, expand the disclosure triangle next to *Executables* in your project's *Groups & Files* list. See figure 16.3.

Next, select the *Automator* executable and either type command + i, or select *Edit Active Executable* 'Automator' from the *Project* menu in Xcode. This will display the *Executable* 'Automator' *Info* window. Once this window appears, click on the *Arguments* tab. Verify that the *action* argument, listed in the *Arguments to be passed upon launch* table is set to the name of your action. See figure 16.4.

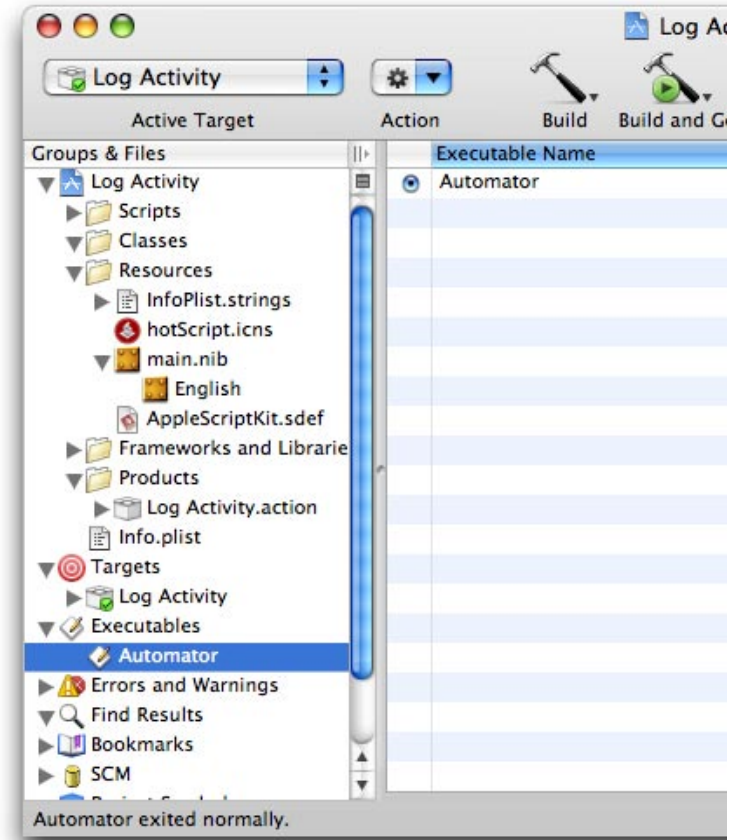


Figure 16.3 The Automator Executable



Chapter 16:
**Testing and
Debugging
an Action**

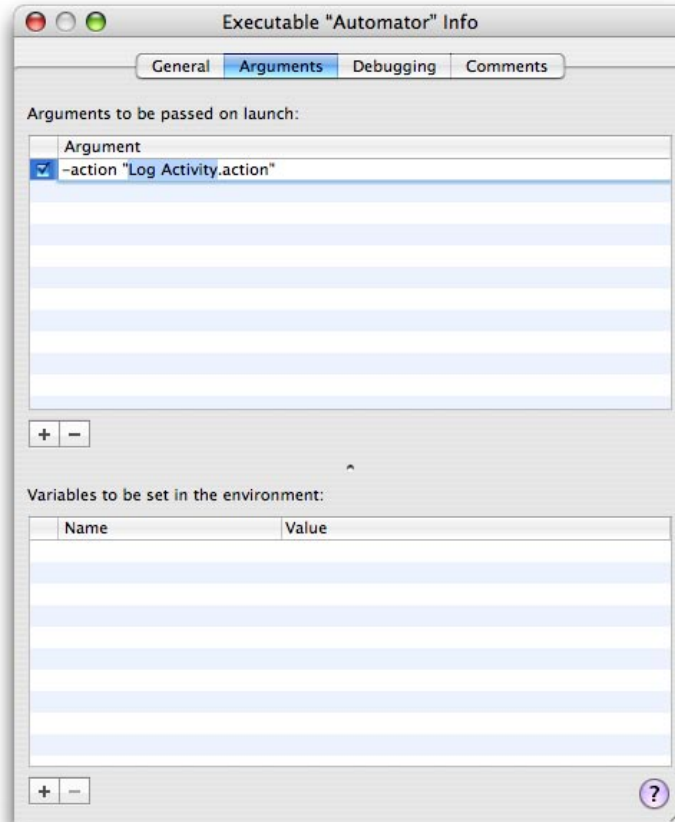


Figure 16.4 Linking the Automator Executable to the Action

Building, Installing, and Testing

At times, you may wish to install an action for testing, rather than running it directly from within Xcode. To do this, build your project by clicking the *Build* button in Xcode's toolbar, or select the *Build* menu item in the *Build* menu. Next, locate your built action bundle in the *build* folder within your project's folder structure in the Finder. See figure 16.5.

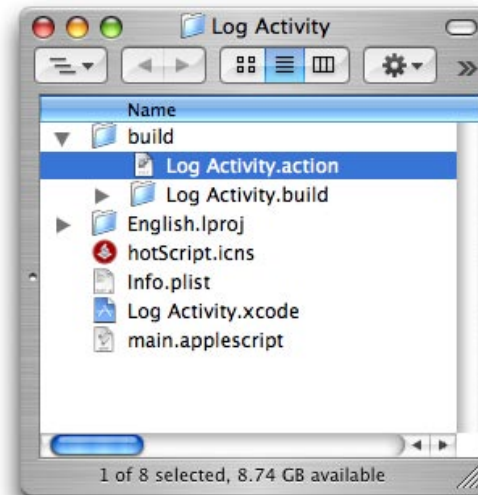


Figure 16.5 A build Automator Action in an Xcode Project Folder



Chapter 16:
**Testing and
Debugging
an Action**

Once you have located the built action, copy it into one of the following folders on your machine:

- ▶ Library > Automator
- ▶ Users > YourUserName > Library > Automator

If Automator is already running, then you will need to re-launch Automator in order for the newly installed action to appear in the *Action* list.

Building and Debugging

Unfortunately, the debugging mechanism for AppleScript Studio is not yet fully implemented within Xcode. Because of this, AppleScripts may not be debugged using Xcode's built-in debugger. However, third party applications, such as Script Debugger, from Late Night Software, Ltd (<http://www.latenightsw.com/>), offer integration with AppleScript Studio, and may be used for debugging. In addition, developers may insert the **log** command into their AppleScripts in order to send messages to the *Run Log* window, if the action is run from within Xcode. If the action is run externally from Xcode, then these messages are sent to the Console application. The following example code demonstrates the proper usage of the **log** command.

```
log "Performing some task..."
```

Xcode does support debugging of Objective-C code, and this method may be used to locate and fix potential problems within a custom action project. When using Xcode's debugger, you may set breakpoints and step through your project's code, line-by-line, checking for problem areas. Detailed information about debugging a project within Xcode can be found in the *Debugging* section of the *Xcode User Guide*, available via the *Apple Developer Connection Reference Library*.



Chapter 16:
**Testing and
Debugging
an Action**

Tips for Testing Actions

When conducting tests of your actions, as with the testing of any software product, you should be as thorough as possible. The following are some suggested steps when performing testing of a custom Automator action:

- ▶ Verify that the action's category appears in the *Library* list within Automator.
- ▶ Verify that the action's category is displayed with the proper icon, if the category represents an application.
- ▶ Verify that the action's name appears in the *Action* list, when you click on the appropriate category in the *Library* list.
- ▶ Verify the spelling of the action's name, and compare the action's name to the names of similar actions.
- ▶ Click on the action in the *Action* list, and verify that its name, icon, and description information appear correctly in the *Description* area of Automator's interface.
- ▶ Verify that the names of any required actions appear properly in the *Description* area of Automator's interface.
- ▶ Add the action to a workflow, and verify that the action's interface appears properly. If a warning dialog appears prior to adding the action, verify that the text and buttons of the warning appear properly. Also, verify that the behavior of the warning is correct. If it supposed to insert another action, then verify that this actually occurs.
- ▶ Adjust settings and monitor the action's behavior. Look for areas that might appear confusing, or that might not behave properly.
- ▶ Ensure that the interface elements displayed in the action's settings interface are properly laid out, and that there is not too much empty space.
- ▶ Try running the workflow in a scenario where the incorrect type of data is passed to your custom action. Observe the result.
- ▶ Try running the workflow in a scenario where the correct type of data is passed to your custom action. Observe the result.
- ▶ If your action has the ability to ignore its input value, then enable this functionality, and test your action's behavior when the workflow is run.
- ▶ If your action has the ability for its interface to be displayed during processing, then enable this functionality, and test the action's behavior when the workflow is run.
- ▶ Add the *View Results* action, found in the Automator category, after your action. Then, run the workflow, and observe the result of your action.
- ▶ If your custom action performs a task in a third party application, observe the results of the action within



Chapter 16:
**Testing and
Debugging
an Action**

that application.

- ▶ Open Automator's log drawer, and observe log entries as the workflow is run.

In the course of testing an action, if you encounter problems or notice improper behavior, go back into Xcode and Interface Builder, and make any necessary adjustments. Next, walk through the testing scenarios again, paying special attention to areas that were previously problematic.

Common Problems and Possible Solutions

The following are some common problems that may be encountered when conducting testing of a custom Automator action, along with possible solutions to those problems.

- ▶ **Problem** – I ran my action from within Xcode, but my action does not appear in Automator.
Possible Solution 1 – Verify that the Automator executable's *action* launch argument in your project is properly set to the name of your action.
Possible Solution 2 – Verify that all necessary keys and values are properly entered in your action's *info.plist* file. Also, be sure that all XML tags are properly written.
Possible Solution 3 – Verify that your action's **CFBundleIdentifier** key is set to a unique value in your action's *info.plist* file. Also, verify that a copy of the action you are running is not already installed as an action that Automator is loading.
- ▶ **Problem** – My action's icon appears generic in Automator.
Possible Solution – Verify that you have entered the proper name of the desired icon for the **AMIconName** key in the action's *info.plist* file. If you are using a custom icon, or any other icon not included in Automator's bundle or the *System > Library > CoreServices > CoreTypes* bundle, then make sure that you have added the icon file, in either TIFF or *icns* format, to your action's project.



Chapter 16:
**Testing and
Debugging
an Action**

- ▶ **Problem** – I entered description information for my action, but when I view the action’s description in Automator, I still see generic default text.
Possible Solution – Verify that you have either removed or updated the corresponding description key values in your action’s localized *InfoPlist.strings* file.

- ▶ **Problem** – When I run my AppleScript action, I get an error saying that Automator can’t get one of my parameter values.
Possible Solution 1 – Verify that the parameter key, and its default value, has been entered into the **AMDefaultParameters** dictionary in the action’s *info.plist* file.
Possible Solution 2 – Verify that your code that is retrieving the parameter’s value contains pipe characters around the parameter name. For example: `|parameterName|`.
Possible Solution 3 – Verify that the parameter’s name that you are referencing in your AppleScript code exactly matches the parameter key specified in the action’s *nib* and in the **AMDefaultParameters** dictionary in the *info.plist* file. Even the case must match.

What’s Next

We have now discussed all of the steps involved in creating custom Automator actions. Now, it is time to begin putting all of the things that we have talked about together. In the next chapter, we will create some sample Automator actions, putting the techniques that we have learned to the test.



Chapter 17 Pulling it Together

Now that we have discussed the various aspects of creating an Automator action, it is time to begin putting all of the topics that we have discussed together. Throughout this chapter, we will create two sample Automator actions, one in AppleScript, and one in Objective-C. The code for these example actions is included in the book's companion files for you to use for reference and for testing. Note that the AppleScript-based action will involve Cocoa bindings. Code for an action that updates its parameters using AppleScript Studio, rather than Cocoa bindings, is included in the companion files.

Please note that the steps included in this chapter are not described in complete detail. Rather, they are meant to serve as overviews of the topics that we have discussed throughout the book. If you find a step or task to be unclear, go back and consult the appropriate chapter in order to gain a better understanding.

Log Activity AppleScript-Based Action

Our first example action will be AppleScript-based, and will log activity during the processing of a workflow. The following steps will walk you through the creation of this action.

Step 1 – Planning the Action

As we discussed, the first task when creating an action is to create an outline of the action's functionality. The following is an example of an action outline.

- ▶ Action Overview: The purpose of this action will be to generate an activity log during the process of executing a workflow.
- ▶ Action Input: Anything
- ▶ Action Output: Anything (The input value will be returned as output)
- ▶ Configurable Settings: Workflow Name, Log Name, Log Output Location, Include Time in Log Entry
- ▶ Action Workflow:



Chapter 17: Pulling it Together

- ▶ Retrieve all parameters from settings
- ▶ Verify that a workflow name and log name have been specified
- ▶ Build the path to the activity log file
- ▶ Create the log text. If possible, it should contain a list of input values.
- ▶ Write the log text to the activity log file
- ▶ Return the input of the action as output

Step 2 – Build the Project

The next step in creating our AppleScript-based action is to begin creating the project within Xcode. Do this by creating a new Xcode project, using the *AppleScript Automator Action* template, provided by Apple. Name the action project *Log Activity*. As we have seen, this new action project will contain everything that we need to get started with building our action. See figure 17.1.

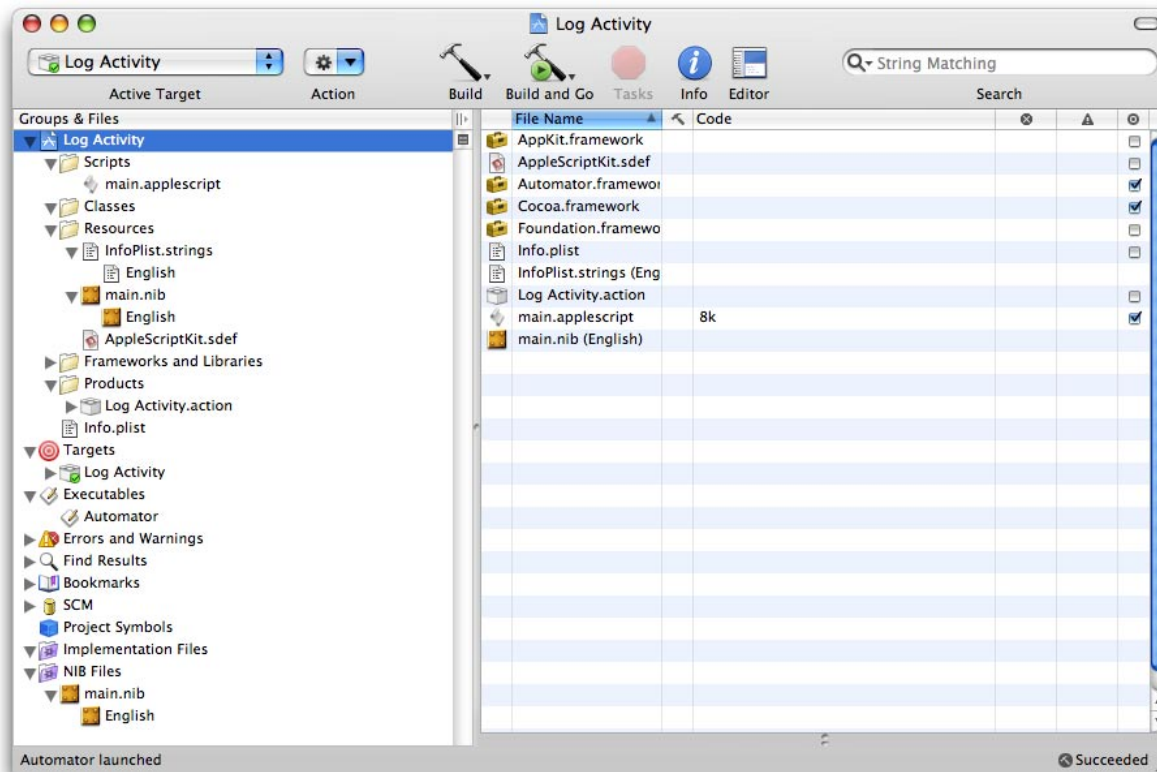


Figure 17.1 The Log Activity AppleScript Action Project



Chapter 17: Pulling it Together

Step 3 – Configuring the Information Property List File

Once our action's project has been created, the next step is to update the key values included in the action's *info.plist* file. This will ensure that Automator recognizes our action, and that it is displayed and behaves correctly.

The following is the complete XML for our configured *info.plist* file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD
  PLIST 1.0//EN" "http://www.apple.com/DTDs/
  PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>AMAccepts</key>
  <dict>
    <key>Container</key>
    <string>List</string>
    <key>Optional</key>
    <false/>
    <key>Types</key>
    <array>
      <string>*</string>
    </array>
  </dict>
  <key>AMApplication</key>
  <array>
    <string>TextEdit</string>
    <string>Automator</string>
  </array>
  <key>AMCanShowSelectedItemsWhenRun</key>
  <true/>
  <key>AMCanShowWhenRun</key>
  <true/>
  <key>AMCategory</key>
```

```
<string>Logs</string>
<key>AMDefaultParameters</key>
<dict>
  <key>workflowName</key>
  <string></string>
  <key>logName</key>
  <string>Activity Log</string>
  <key>outputLocation</key>
  <string>~/Documents</string>
  <key>includeTime</key>
  <true/>
</dict>
<key>AMDescription</key>
<dict>
  <key>AMDOptions</key>
  <string>Workflow name, log name, location,
  and whether to include the time in each
  activity log entry.</string>
  <key>AMDSummary</key>
  <string>This action will generate an entry
  in an activity log during processing of a
  workflow.</string>
</dict>
<key>AMIconName</key>
<string>TextEdit</string>
<key>AMKeywords</key>
  <array>
    <string>Generate</string>
    <string>Activity</string>
    <string>Logging</string>
  </array>
<key>AMName</key>
<string>Log Activity</string>
<key>AMProvides</key>
<dict>
  <key>Container</key>
  <string>List</string>
  <key>Types</key>
  <array>
    <string>*</string>
```



Chapter 17: Pulling it Together

```
</array>
</dict>
<key>AMRequiredResources</key>
  <array/>
<key>AMWarning</key>
<dict>
  <key>Action</key>
  <string></string>
  <key>ApplyButton</key>
  <string></string>
  <key>IgnoreButton</key>
  <string></string>
  <key>Level</key>
  <integer>0</integer>
  <key>Message</key>
  <string></string>
</dict>
<key>CFBundleDevelopmentRegion</key>
<string>English</string>
<key>CFBundleExecutable</key>
<string>Log Activity</string>
<key>CFBundleGetInfoString</key>
<string>Log Activity version 1.0, Copyright (c)
2005, SpiderWorks, LLC.</string>
<key>CFBundleIconFile</key>
<string></string>
<key>CFBundleIdentifier</key>
<string>com.spiderworks.Automator.Log
Activity</string>
<key>CFBundleInfoDictionaryVersion</key>
<string>6.0</string>
<key>CFBundleName</key>
<string>Log Activity</string>
<key>CFBundlePackageType</key>
<string>BNDL</string>
<key>CFBundleShortVersionString</key>
<string>1.0</string>
<key>CFBundleSignature</key>
<string>????</string>
<key>CFBundleVersion</key>
```

```
<string>1.0</string>
<key>NSPrincipalClass</key>
<string>AMAppleScriptAction</string>
</dict>
</plist>
```

Once you have configured the *info.plist* file for the action, you will need to remove or update any key values included in the English *InfoPlist.strings* localized version of the *info.plist* file. The following is an example of the contents of an updated *InfoPlist.strings* file.

```
/* Localized versions of Info.plist keys */

CFBundleName = "Log Activity";
NSHumanReadableCopyright = "Copyright 2005
SpiderWorks, LLC.";

AMName = "Log Activity";

/* AMApplication localized strings */
/* e.g. "TextEdit" = "TextEdit"; */

/* AMCategory localized strings */
/* e.g. "Text" = "Text"; */

/* AMDefaultParameters localized values */
/* e.g. myDefaultIntegerParameter = 0; */
/* e.g. myDefaultStringParameter = "Default
String Value"; */

/* AMDescription localized strings */
AMDOptions = "Workflow name, log name, location,
and whether to include the time in each
activity log entry.";
AMDSummary = "This action will generate an entry
in an activity log during processing of a
workflow.";
```




Chapter 17: Pulling it Together

```
/* AMKeyword localized strings */  
/* e.g. "Filter" = "Filter"; */  
  
/* AMWarning localized strings */  
ApplyButton = "";  
IgnoreButton = "";  
Message = "";
```

Step 4 – Building the Interface

Next, we will need to begin building the interface for our action. To do this, open the action project's *nib* file in Interface Builder, and begin designing the interface. Figure 17.2 shows an example of how the interface for this example action should appear.

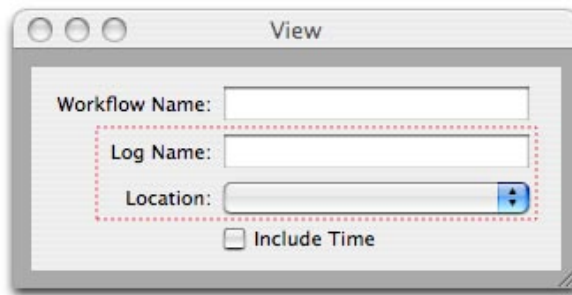


Figure 17.2 Log Activity Action Interface

Note that the *Log Name* and *Location* text fields in the interface have been grouped together to form an *Automator Box* subview. These text fields have also been linked together to allow tabbing between the fields. The *Location* popup is a *Directory Chooser* (*NSPathPopUpButton*), which can be found in the *Cocoa-Automator* palette in the *Palettes* window.

Step 5 – Assigning Bindings

Once the interface has been designed, it is time to begin applying Cocoa bindings to the elements within our interface. As mentioned before, using AppleScript Studio event handlers to update the parameters for the action is possible, but not explained in this section. An example of this, however, is included in the companion files for this chapter.

To apply Cocoa bindings, first add the following parameter keys for the action. See figure 17.3 for an example of properly added parameter keys.

- ▶ includeTime
- ▶ logName
- ▶ outputLocation
- ▶ workflowName



Chapter 17: Pulling it Together

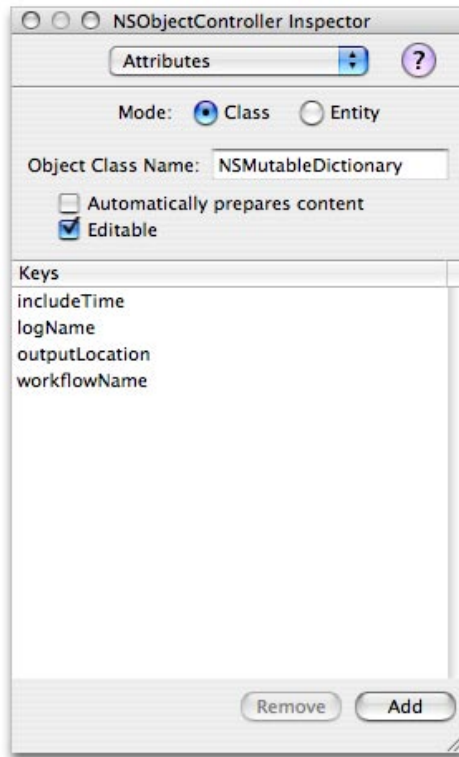


Figure 17.3 Parameter Keys for the Log Activity Action

Next, bind the interface elements to the parameter keys you just assigned. Figure 17.4 shows an example of a parameter key that has been properly bound to the location path popup.

The next step in assigning Cocoa bindings in our action would normally be to add the specified parameter keys to our action's *info.plist* file. We actually already did this in step 4. This is evident in the following text from our *info.plist* file.

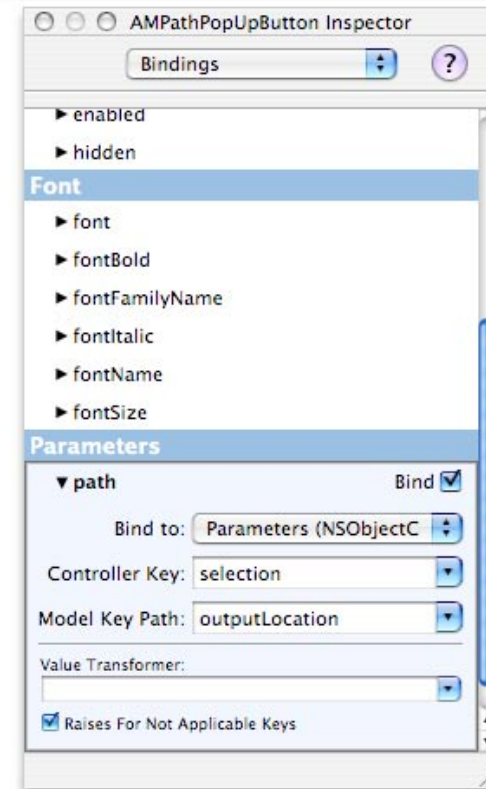
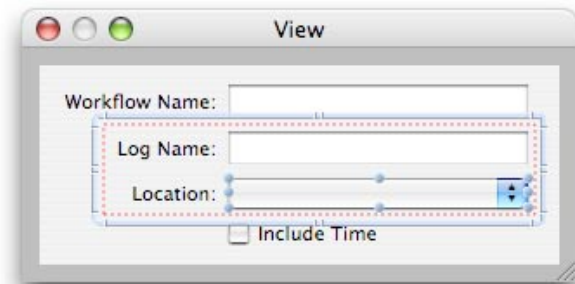


Figure 17.4 A Properly Bound Interface Element



Chapter 17: Pulling it Together

```
<key>AMDefaultParameters</key>
<dict>
  <key>workflowName</key>
  <string></string>
  <key>logName</key>
  <string>Activity Log</string>
  <key>outputLocation</key>
  <string>~/Documents</string>
  <key>includeTime</key>
  <true/>
</dict>
```

Step 6 – Writing the Code

The final step in the actual development of our action, before we can begin testing, is to write the action's processing code. For the purposes of this example, the following code may be entered into the action's *main.applescript* file.

```
-- main.applescript
-- Log Activity

-- Created by Benjamin Waldie on 3/16/05.
-- Copyright 2005 SpiderWorks, LLC.
-- All rights reserved.

on run {input, parameters}

  -- Copy the input to an output variable
  copy input to output

  -- Retrieve all parameters
  set workflowName to |workflowName| of parameters
  set logName to |logName| of parameters
  set outputLocation to |outputLocation| of
parameters
  set includeTime to |includeTime| of parameters
```

```
-- If required information is not present,
-- then return the result
if workflowName = "" or logName = "" then
return output

-- Determine the path to where the log file
-- should be generated
set outputLocation to ((POSIX file (do shell
script "echo " & outputLocation)) as alias)
as string
set shortDate to do shell script "date
+'%m.%d.%y'"
set logName to shortDate & " - " & logName
if logName does not end with ".txt" then set
logName to logName & ".txt"
set logPath to outputLocation & logName

-- Generate the log text
set logText to "Workflow '" & workflowName & "'
executing..." & return
if includeTime = true then set logText to
(time string of (current date)) & " - " & logText

-- Loop through the input, and build a list
-- of input values to log, if able to do so
if class of input is not equal to list then
set input to {input}
set inputLogText to ""
repeat with a from 1 to length of input
  try
    set currentInput to (item a of input)
  as string
    if inputLogText is not equal to ""
then set inputLogText to inputLogText & ", "
    set inputLogText to inputLogText &
currentInput
  on error
    set inputLogText to "Input could not
be converted to a string for logging."
  exit repeat
```



Chapter 17: Pulling it Together

```
        end try
    end repeat
    set logText to logText & "Input: " &
inputLogText & return & return

    -- Write to the activity log
    try
        close access file logPath
    end try
    set logFile to open for access file logPath with
write permission
    write logText to logFile starting at eof
    close access logFile

    -- Return the result
    return output
end run
```

Step 7 – Testing the Action

Now that we have completed initial development, we can begin testing our new, custom action. First, make sure that the *Automator* executable in your action's project contains an *action* launch argument set to “*Log Activity.action*”. See figure 17.5.

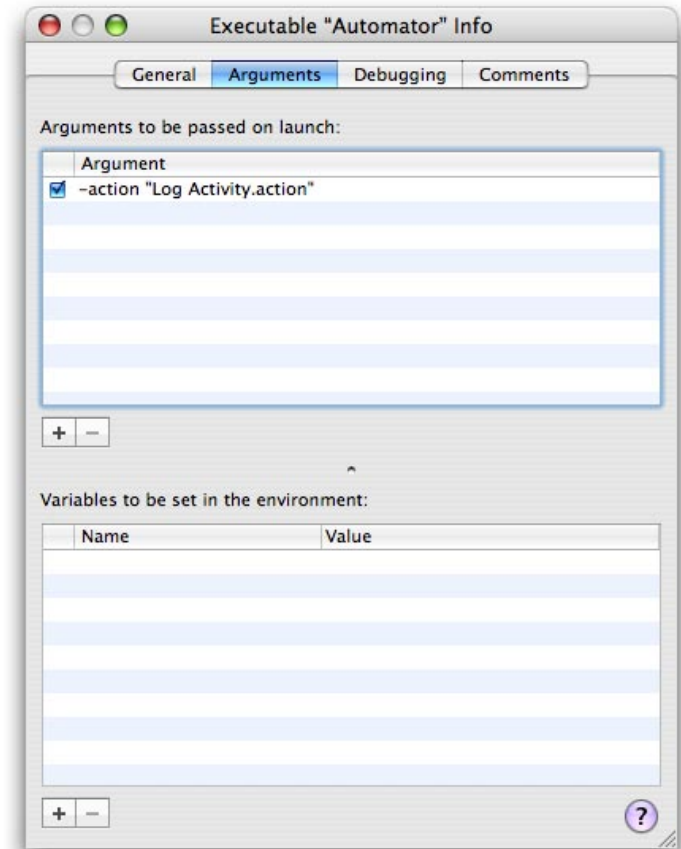


Figure 17.5 Automator Executable Launch Argument

Next, build and run the action project from within Xcode. Once a new instance of Automator launches, click on the Automator icon in the *Library* list, and verify that the *Log Activity* action appears in the *Action* list. See figure 17.6.



Chapter 17: Pulling it Together

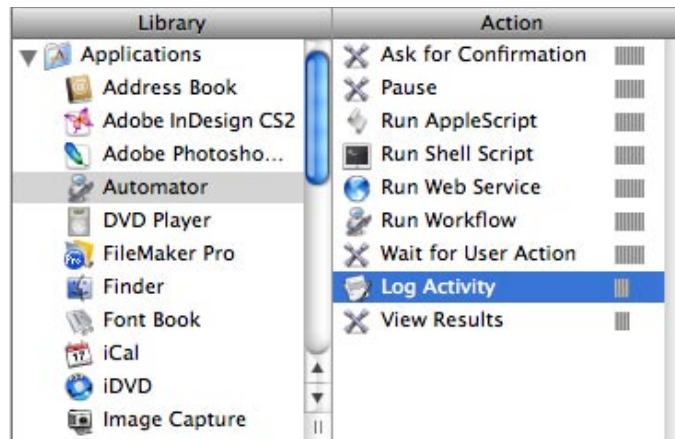


Figure 17.6 Log Activity Action in Automator's Interface

Click on the *Log Activity* action, and verify that the action's description and icon appear properly. Finally, construct a workflow that includes the *Log Activity* action, and begin testing the action. Make any necessary adjustments, based on your testing, and always be sure to re-test once any changes have been made. See figure 17.7.

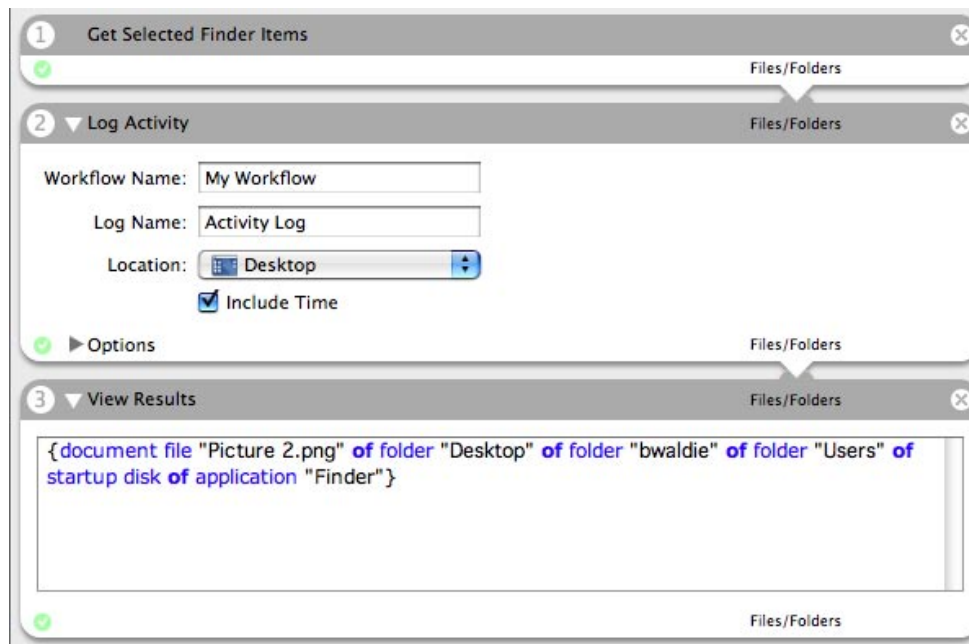


Figure 17.7 Log Activity Action in a Workflow



Chapter 17:
**Pulling it
Together**

Adjust Image Color Cocoa Objective-C-Based Action

Our next example action is based on Cocoa and Objective-C. This action will use the Core Image framework to allow a user to adjust the color in image files. The design for this action was based on the *Adjust Gamma* example Automator action project that is included with the Mac OS X developer tools. The following steps will walk you through the creation of this action.

Step 1 – Planning the Action

As we did with our AppleScript-based action example, the first step in creating our Cocoa action will be to create a project plan. The following is an example of an action outline that will serve as our project plan.

- ▶ Action Overview: The purpose of this action will be to adjust various color aspects of images, using values specified by the user.
- ▶ Action Input: Image paths
- ▶ Action Output: Image paths
- ▶ Configurable Settings: Saturation, Brightness, and Contrast Levels
- ▶ Action Workflow:
 - ▶ Loop through the passed input
 - ▶ Trigger the **CIColorControls** Core Image color adjustment filter on each image.

- ▶ Save the modified image
- ▶ Return the input of the action as output

Step 3 – Build the Project

The next step in creating our Cocoa action is to begin creating the project within Xcode. Start by creating a new Xcode project, using the *Cocoa Automator Action* template, provided by Apple. Name the new action project *Adjust Color of Images*. See figure 17.8.

Once you have created the project, you will need to add the *QuartzCore.framework* to the project. This framework is necessary in order for our action to access the necessary *Core Image* filter during processing. You can add the *QuartzCore.framework* to the project by clicking on the *Linked Frameworks* group in the *Groups & Files* list in your project, and selecting *Add to Project* from the *Project* menu. Next, navigate to the *System > Frameworks* folder on your machine, and select the framework to be added to the project.



Chapter 17: Pulling it Together

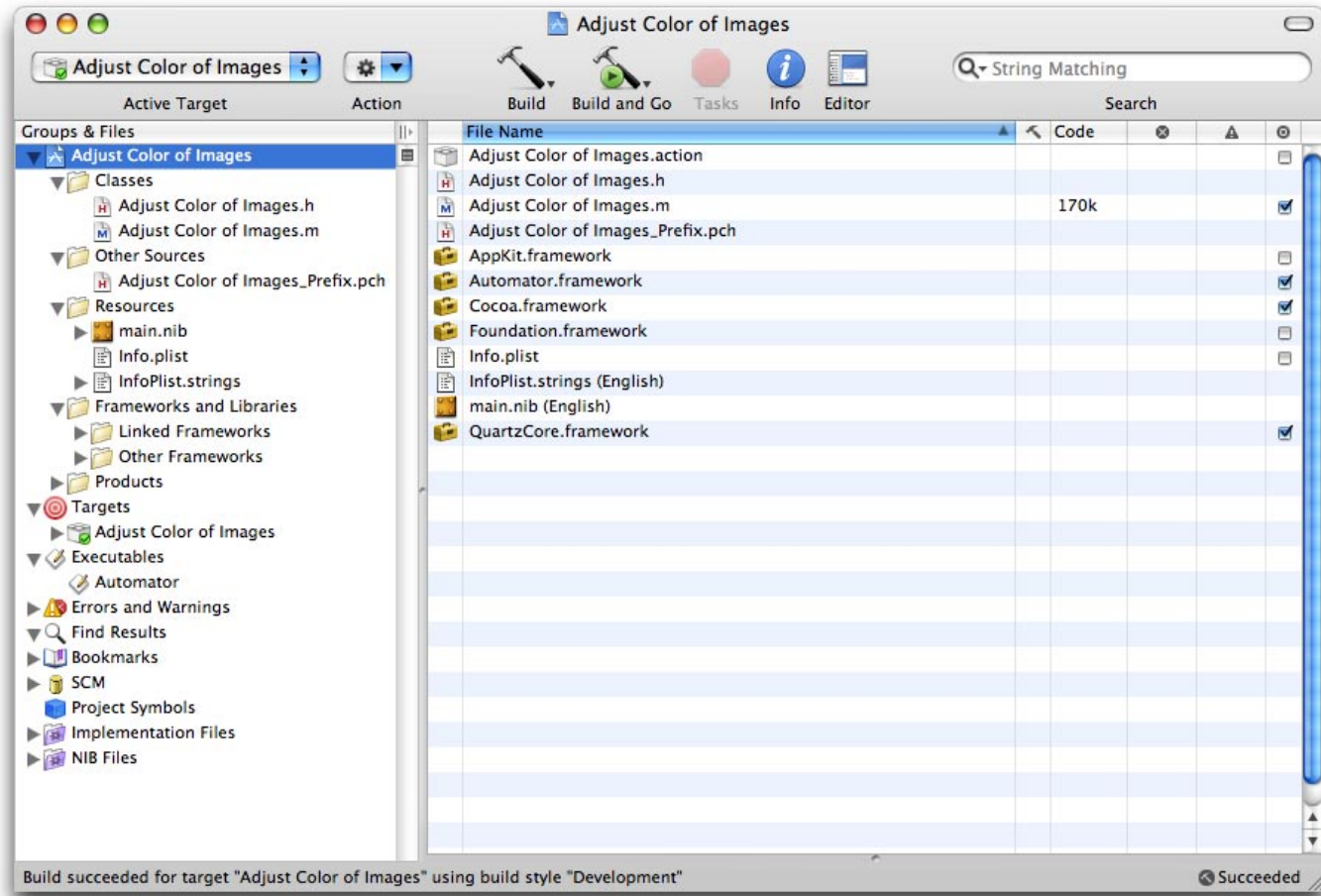


Figure 17.8 Adjust Color of Images Action Project

Step 3 – Configuring the Information Property List File

Once the action's project has been created, the next step will be to update the key values included in the action's *info.plist* file. The following is the complete XML for

our configured *info.plist* file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD
PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
```



Chapter 17: Pulling it Together

```
<plist version="1.0">
<dict>
  <key>AMAccepts</key>
  <dict>
    <key>Container</key>
    <string>List</string>
    <key>Optional</key>
    <false/>
    <key>Types</key>
    <array>
      <string>com.apple.cocoa.path</string>
    </array>
  </dict>
  <key>AMApplication</key>
  <string>Preview</string>
  <key>AMCanShowSelectedItemWhenRun</key>
  <false/>
  <key>AMCanShowWhenRun</key>
  <true/>
  <key>AMCategory</key>
  <string>Images</string>
  <key>AMDefaultParameters</key>
  <dict>
    <key>saturation</key>
    <string>1.0</string>
    <key>brightness</key>
    <string>0.0</string>
    <key>contrast</key>
    <string>1.0</string>
  </dict>
  <key>AMDescription</key>
  <dict>
    <key>AMDOptions</key>
    <string>Saturation, brightness, and contrast
    levels.</string>
    <key>AMDSummary</key>
    <string>This action will adjust the
    saturation, brightness, and contrast of
    images.</string>
  </dict>
```

```
<key>AMIconName</key>
<string>Preview</string>
<key>AMKeywords</key>
<array>
  <string>core</string>
  <string>saturation</string>
  <string>brightness</string>
  <string>contrast</string>
</array>
<key>AMName</key>
<string>Adjust Color of Images</string>
<key>AMProvides</key>
<dict>
  <key>Container</key>
  <string>List</string>
  <key>Types</key>
  <array>
    <string>com.apple.cocoa.path</string>
  </array>
</dict>
<key>AMRequiredResources</key>
<array/>
<key>AMWarning</key>
<dict>
  <key>Action</key>
  <string>com.apple.Automator.CopyFiles</
string>
  <key>ApplyButton</key>
  <string>Add</string>
  <key>IgnoreButton</key>
  <string>Don't Add</string>
  <key>Level</key>
  <integer>1</integer>
  <key>Message</key>
  <string>This action will change the image
  files passed into it. Would you like to add a
  Copy Finder Items action so that the copies are
  changed and your originals are preserved?</
string>
</dict>
```



Chapter 17: Pulling it Together

```
<key>CFBundleDevelopmentRegion</key>
<string>English</string>
<key>CFBundleExecutable</key>
<string>Adjust Color of Images</string>
<key>CFBundleGetInfoString</key>
<string>Adjust Color of Images version 1.0,
Copyright (c) 2005, SpiderWorks, LLC.</string>
<key>CFBundleIconFile</key>
<string></string>
<key>CFBundleIdentifier</key>
<string>com.spiderworks.Automator.
AdjustColorOfImages</string>
<key>CFBundleInfoDictionaryVersion</key>
<string>6.0</string>
<key>CFBundleName</key>
<string>Adjust Color of Images</string>
<key>CFBundlePackageType</key>
<string>BNDL</string>
<key>CFBundleShortVersionString</key>
<string>1.0</string>
<key>CFBundleSignature</key>
<string>????</string>
<key>CFBundleVersion</key>
<string>1.0</string>
<key>NSPrincipalClass</key>
<string>AdjustColorOfImages</string>
</dict>
</plist>
```

Once you have configured the *info.plist* file for the action, you will need to remove or update any key values included in the English *InfoPlist.strings* localized version of the *info.plist* file. The following is an example of the contents of an updated *InfoPlist.strings* file.

```
/* Localized versions of Info.plist keys */

CFBundleName = "Adjust Color of Images";
NSHumanReadableCopyright = "Copyright 2005
SpiderWorks, LLC.";

AMName = "Adjust Color of Images";

/* AMApplication localized strings */
"Preview" = "Preview";

/* AMCategory localized strings */
"Images" = "Images";

/* AMDefaultParameters localized strings */

/* AMDescription localized strings */
AMDAAlert = "This action will modify your
original images.";
AMDOptions = "Saturation, brightness, and
contrast levels.";
AMDSummary = "This action will adjust the
saturation, brightness, and contrast of
images.";

/* AMWarning localized strings */
ApplyButton = "Add";
IgnoreButton = "Don't Add";
Message = "This action will change the image
files passed into it. Would you like to add a
Copy Finder Items action so that the copies are
changed and your originals are preserved?";
```

Step 4 – Building the Interface

Next, we will need to build the interface for our action. To do this, open the action project's *nib* file in Interface Builder, and begin designing the interface for the action. Figure 17.9 shows an example of how the interface for



Chapter 17: Pulling it Together

this example action should appear.



Figure 17.9 *Adjust Color of Images Action Interface*

Note that the text fields and sliders have been connected together so that they are synchronized whenever one of their values is changed. In addition, the slider values are configured as follows:

- ▶ Saturation Slider – Minimum Value: 0.0, Maximum Value: 3.0, Current Value: 1.0
- ▶ Brightness Slider - Minimum Value: -1.0, Maximum Value: 1.0, Current Value: 0.0
- ▶ Contrast Slider - Minimum Value: 0.3, Maximum Value: 4.0, Current Value: 1.0

Step 5 – Assigning Bindings

Once the interface has been designed, it is time to apply Cocoa bindings to the elements within our interface. To do this, first add the following parameter keys for the action. See figure 17.10 for an example of properly added parameter keys.

- ▶ brightness
- ▶ contrast
- ▶ saturation

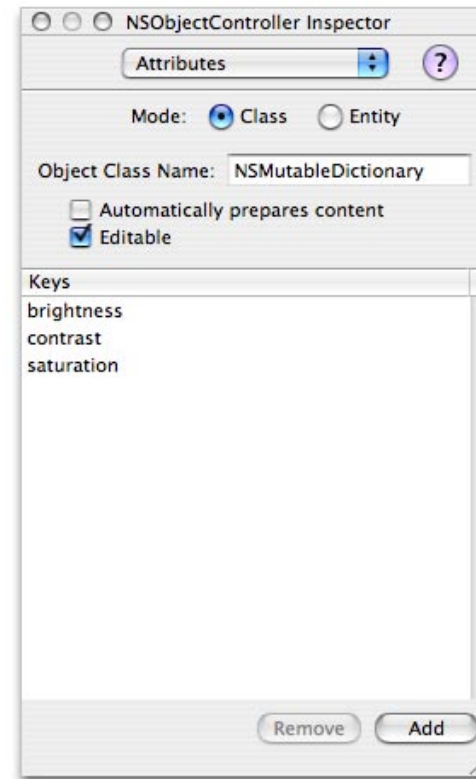


Figure 17.10 *Parameter Keys for the Adjust Color of Images Action*

Next, bind the slider in the interface to the parameter keys you just assigned. Figure 17.11 shows an example of a parameter key that has been properly bound to the saturation slider.



Chapter 17: Pulling it Together

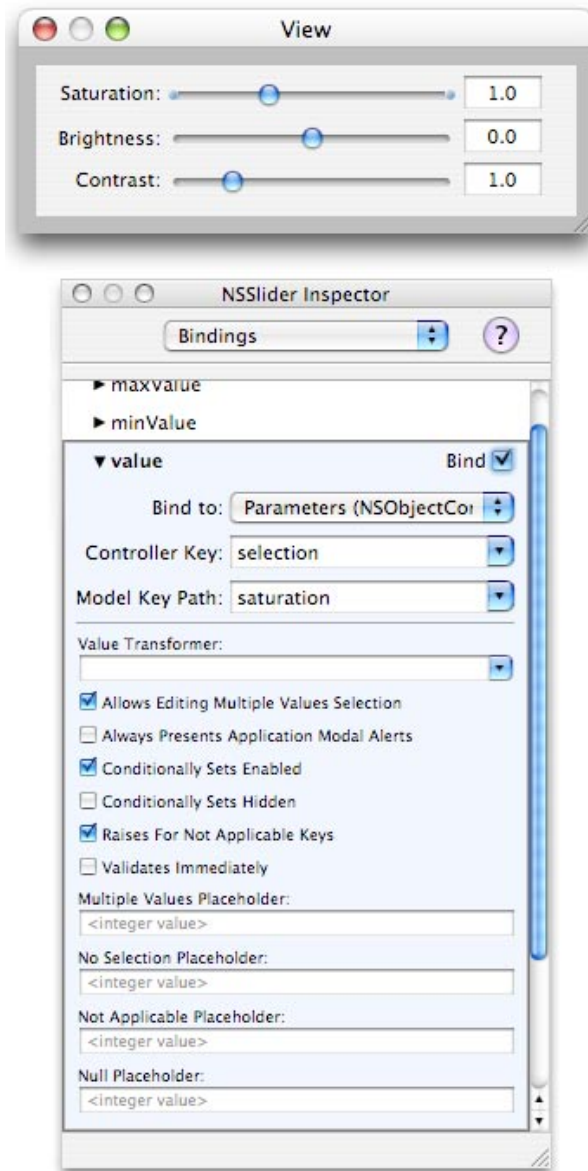


Figure 17.11 A Properly Bound Slider

The next step in assigning Cocoa bindings in our action would normally be to add the specified parameter keys to our action's *info.plist* file. We actually already did this in step 4. This is evident in the following text from our *info.plist* file.

```
<key>AMDefaultParameters</key>
<dict>
  <key>saturation</key>
  <string>1.0</string>
  <key>brightness</key>
  <string>0.0</string>
  <key>contrast</key>
  <string>1.0</string>
</dict>
```

Step 6 – Writing the Code

The final step in the actual development of our action, before we can begin conducting testing, is to write the action's processing code. The following code may be entered into the action's *Adjust Color of Images.h* file.

```
/*
  Adjust Color of Images.h
  Adjust Color of Images
*/

#import <Cocoa/Cocoa.h>
#import <QuartzCore/QuartzCore.h>
#import <Automator/AMBundleAction.h>

@interface AdjustColorOfImages : AMBundleAction
{
    CIContext *context;
}
```



Chapter 17: Pulling it Together

```
- (id)runWithInput:(id)input fromAction:(AMAction
    *)anAction error:(NSDictionary **)errorInfo;

@end
```

We will also need to enter code into the *Adjust Color of Images.m* file. The following code should be entered into this file.

```
/*
    Adjust Color of Images.m
    Adjust Color of Images
*/

#import "Adjust Color of Images.h"

@implementation AdjustColorOfImages

- (NSData *)tiffDataForImage:(CIImage *)image
    fromRect:(CGRect)extent
{
    CFMutableDataRef data = CFDataCreateMutable(
        kCFAllocatorDefault, 0);
    CGImageDestinationRef ref =
        CGImageDestinationCreateWithData(data,
        (CFStringRef)@"public.tiff", 1, NULL);
    if (ref == NULL)
    {
        CFRelease(data);
        CFRelease(ref);
        return nil;
    }

    // make a CGImageRef
    CGImageRef iref = [context createCGImage:
        image fromRect:extent];
```

```
    // add image to the ImageIO destination
    // (specify the image we want to save)
    CGImageDestinationAddImage(ref, iref, NULL);

    // save the image to the TIFF format as data
    if (!CGImageDestinationFinalize(ref))
    {
        CFRelease(data);
        CFRelease(ref);
        return nil;
    }

    CFRelease(ref);
    return [(NSData *)data autorelease];
}

- (id)runWithInput:(id)input
    fromAction:(AMAction *)anAction
    error:(NSDictionary **)errorInfo
{
    id output = input;

    if (![input isKindOfClass:[NSArray class]])
    {
        input = [NSArray arrayWithObject:input];
    }

    // get the graphics context
    if (context == nil)
    {
        context = [CIColorControls contextWithCGContext:
            [[NSGraphicsContext currentContext]
            graphicsPort] options: nil];
        [context retain];
    }

    // create and setup the filter
    CIFilter *colorControls =
        [CIFilter filterWithName:@"CIColorControls"];
    [colorControls setDefaults];
```




Chapter 17: Pulling it Together

```
[colorControls setValue: [NSNumber numberWithFloat:[[[self parameters] objectForKey:@"saturation"]
floatValue]] forKey:@"inputSaturation"];
[colorControls setValue: [NSNumber numberWithFloat:[[[self parameters] objectForKey:@"brightness"]
floatValue]] forKey:@"inputBrightness"];
[colorControls setValue: [NSNumber numberWithFloat:[[[self parameters] objectForKey:@"contrast"]
floatValue]] forKey:@"inputContrast"];

NSEnumerator *enumerator = [input objectEnumerator];
NSString *path;

while (path = [enumerator nextObject])
{
    // create a CIImage from the file
    CIImage *image = [CIImage imageWithContentsOfURL:[NSURL fileURLWithPath:path]];
    if (image)
    {
        [colorControls setValue:image forKey:@"inputImage"];
        CIImage *result = [colorControls valueForKey:@"outputImage"];
        if (result)
        {
            // save the changes to the path
            NSData *data = [self tiffDataForImage:result fromRect:[image extent]];
            if (data)
            {
                if (![data writeToFile:path atomically:YES])
                {
                    NSString *errorString = @"Adjust Color of Images could not create
output file.";
                    *errorInfo = [NSDictionary dictionaryWithObjectsAndKeys: [errorString
autorelease], NSAppleScriptErrorMessage, nil];
                }
            }
        }
    }
    [context release];
    context = nil;
    return output;
}

@end
```



Chapter 17: Pulling it Together

Step 7 – Testing the Action

Now that we have completed initial development on our new custom action, we can begin testing the action. First, make sure that the *Automator* executable in your action's project has an *action* launch argument set to “*Adjust Color of Images.action*”. See figure 17.12.

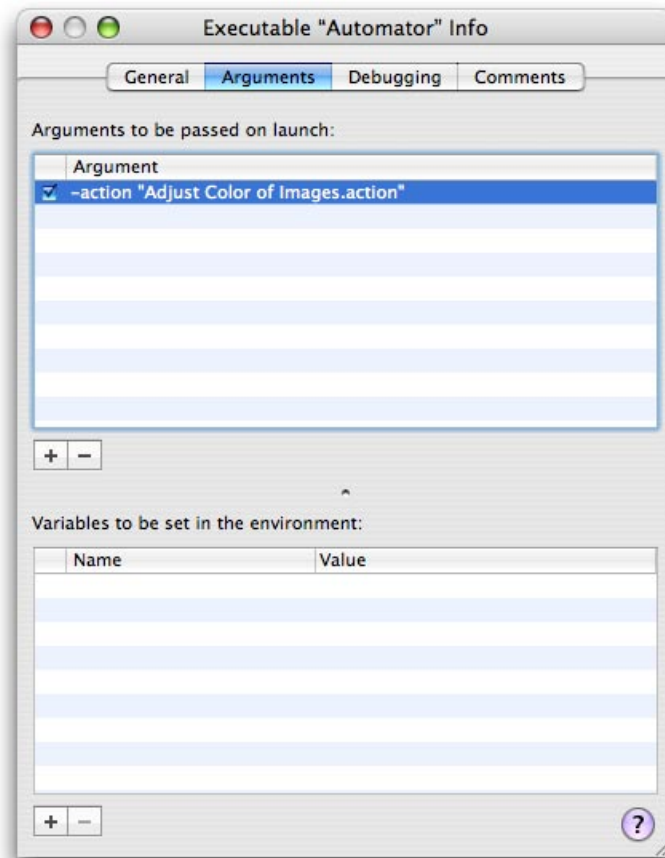


Figure 17.12 Automator Executable Launch Argument

Next, build and run the action project from within Xcode. Once Automator launches, click on the Preview icon in the *Library* list, and verify that the *Adjust Color of Images* action appears in the *Action* list. See figure 17.13.

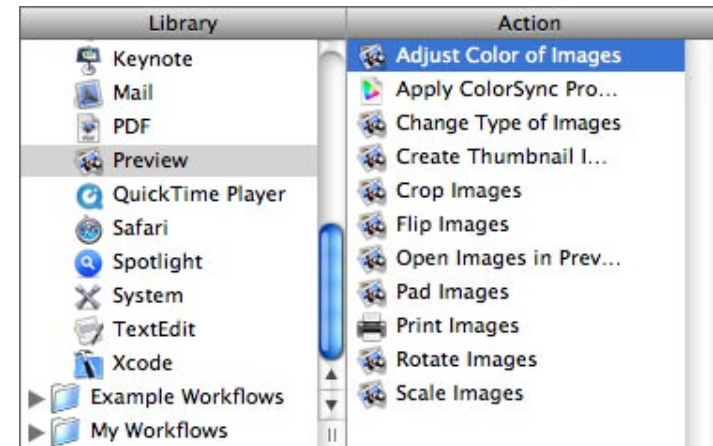


Figure 17.13 Adjust Color of Images Action in Automator's Interface

Click on the *Adjust Color of Images* action, and verify the action's description and icon appear properly. Finally, construct a workflow that includes the *Adjust Color of Images* action, and begin conducting testing of the action. Make any necessary adjustments, based on your testing, and re-test any changes you make. See figure 17.14.



Chapter 17:
**Pulling it
Together**

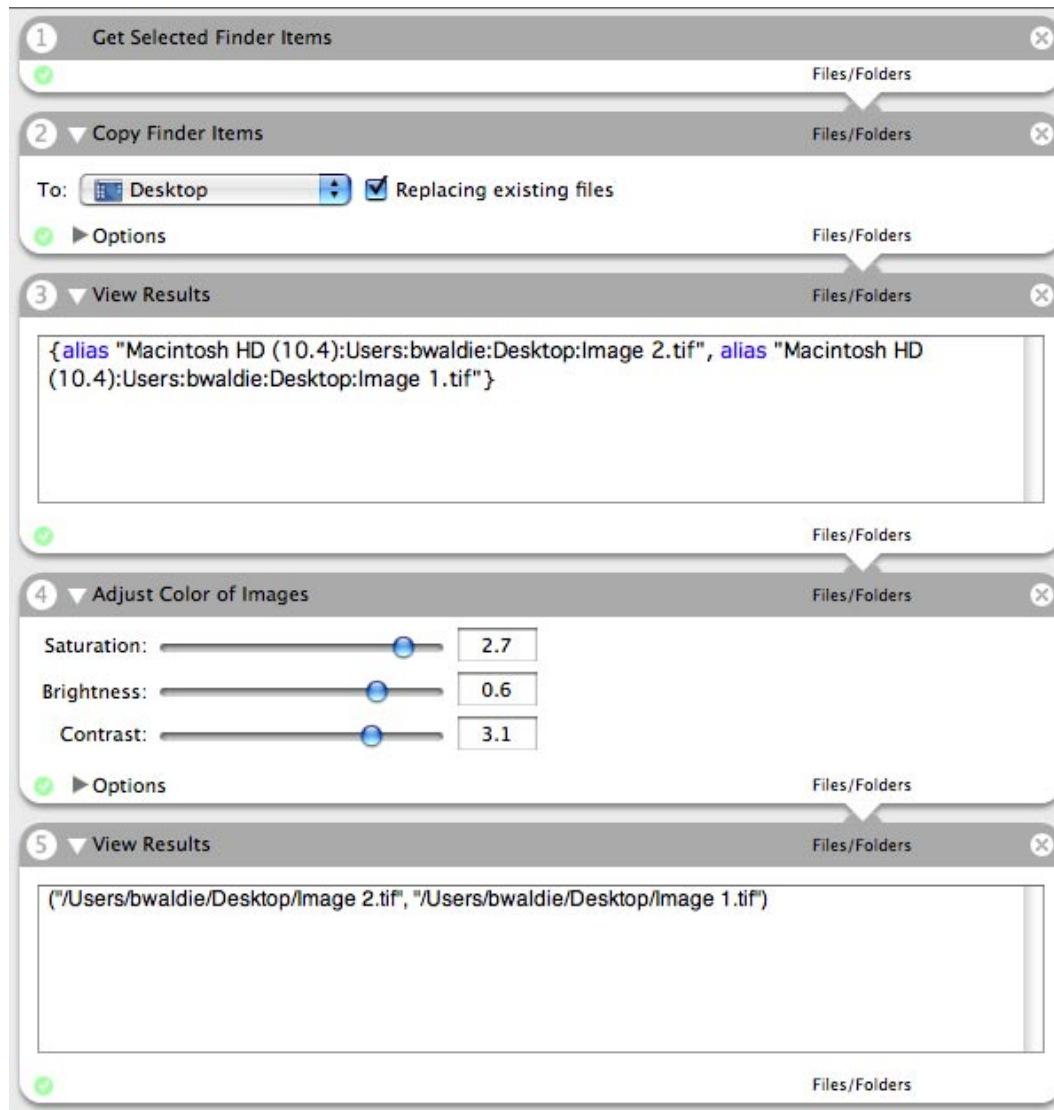


Figure 17.14 Apply Color to Images in a Workflow



Chapter 17:

Pulling it Together

What's Next

This concludes the development section of this book. It is now time for you to begin putting what you have learned to work for you by creating your own custom actions. The next, and final, chapter of this book will provide suggested resources for continued learning about developing for Automator, as well as other Mac OS X-related development technologies.



Chapter 18 In Conclusion

Throughout this book, we have explored a number of topics that should help both users and developers to better understand Automator and begin using it to their advantage. In this final chapter, we will discuss some recommended resources that are available to both users and developers, which can provide additional information about Automator, as well as other technologies that were mentioned in this book. Like the book itself, the following resources are broken into a set of general resources and a set of developer-related resources.

General Automator Resources

The following resources pertain to general use of Automator. These resources may be of interest to users interested in expanding their knowledge and use of Automator.

Third-Party Actions

While Apple provides users with several hundred actions for use with Apple applications and technologies, additional actions are available that provide expanded functionality, or interaction with third-party applications. While some of these actions are available commercially, others may be downloaded for free.

The following is a list of some web sites that serve as resources for expanding your Automator action library:

► Automated Workflows, LLC

(<http://www.automatedworkflows.com>) – Sure, this is a shameless plug for my own company, but I have created a number of Automator actions, which are available through my company's web



Chapter 18:
In Conclusion

site, for applications including Adobe InDesign and Photoshop.

► **Apple's Third Party Download Site**

(<http://www.apple.com/downloads/macosx/>) – This web site, provided by Apple, serves as an excellent online resource for locating third-party Automator actions, in addition to other Mac OS X software.

► **Automator Actions**

(<http://www.automatoractions.com/>) – This web site is a sub-section of MacScripter.net, a popular AppleScript web site. It features a comprehensive list of third-party actions available commercially, or for free download.

Web Sites

In addition to the web sites mentioned above for downloading third-party actions, additional web sites focus on Automator in general.

► **Apple's Automator Web site**

(<http://www.apple.com/automator/>) – This comprehensive website provides a detailed overview of Automator, along with example workflows and other resources.

► **MacScripter.net**

(<http://www.macscripter.net/>) – In addition to hosting third-party Automator actions, MacScripter also offers up to the minute AppleScript and Automator news, as well as an online Automator forum.

► **Automator World**

(<http://www.automatorworld.com/>) – This web site serves as another repository for Automator news, and also offers an online forum for discussing Automator.



Chapter 18:
In Conclusion

Developer Resources

The following resources pertain to developers interested in furthering their Automator action development skills, or skills in Mac programming in general.

Documentation

There are a number of documentation resources provided by Apple for developers interested in creating Automator actions. These resources can be found in the *Apple Developer Reference Library*, which is available online, as well as with the Xcode developer tools. Documentation of particular usefulness that exists as of the writing of this book includes:

- ▶ **Automator Objective-C Reference** – This document discusses the Cocoa API that may be used by Objective-C programmers for interacting with Automator actions.
- ▶ **Automator Programming Guide** – This document provides a fairly comprehensive walk-through of the development process used to construct Automator actions.
- ▶ **AppleScript Studio Terminology Reference** – This document discusses, in detail, the terminology that is used to create AppleScript Studio applications.
- ▶ **AppleScript Studio Programming Guide** – This document provides detailed instructions for creating AppleScript Studio projects in Xcode. It includes a number of tutorials that walk through the complete process of project creation.

- ▶ **Objective-C Programming Guide** – This document provides a complete introduction to the Objective-C programming language. Also covered in detail, is an introduction to object-oriented programming.
- ▶ **Introduction to Developing Cocoa Objective-C Applications** – This document serves as a tutorial to creating basic Objective-C-based Cocoa applications, using Xcode and Interface Builder.
- ▶ **Framework References** – The Apple Developer Reference Library contains a number of documents providing overviews of the methods and classes used for various Mac OS X frameworks, including the Application Kit framework, Core Data framework, Foundation framework, Preference Pane framework, Web Kit framework, and more.

Books

- ▶ **Danny Goodman's AppleScript Handbook** (<http://spiderworks.com/books/ashandbook.php>) – This eBook is perhaps the most popular AppleScript book of all time. It was originally written shortly after AppleScript was first introduced, though it has since been updated for Mac OS X. This book serves as an excellent study guide to anyone looking to learn AppleScript.
- ▶ **AppleScripting the Finder** (<http://spiderworks.com/books/asfinder.php>) – Another shameless plug. This eBook was written by yours truly. It takes an in-depth look at



Chapter 18: In Conclusion

AppleScripting the Finder in Mac OS X, and explains why Finder scripting is necessary in order to achieve certain automation goals. You may find this book useful if you are interested in writing Automator actions that interact with the Finder in Mac OS X.

- ▶ **Learn C on the Macintosh, Mac OS X Edition** (<http://spiderworks.com/books/learncmac.php>) – This book is geared toward users with no prior programming experience. It focuses on the basics of programming, while at the same time, teaches C, one of the most widely used programming languages.
- ▶ **Learn Objective-C, Mac OS X Edition** (<http://www.spiderworks.com/>) – This book, available from SpiderWorks, LLC, serves as the sequel to *Learn C on the Macintosh*, and offers an in-depth look at programming with Objective-C.

Sample Code

The action templates included with Xcode provide an excellent place to start when constructing your own actions. However, Apple also provides some fully editable functional example action projects. These example projects can be found in the *Developer > Examples > Automator* folder. These example projects provide demonstrations of Cocoa-based actions, as well as AppleScript-based actions.

Web Sites

The following websites will provide you with information about the Apple technologies discussed throughout this book.

- ▶ **Apple Developer Connection** (<http://developer.apple.com/>) – The Apple Developer Connection is the best resource around for information about development in Mac OS X. Here, you will find complete documentation and examples for developing virtually anything for Mac OS X. For information specific to Xcode tools, visit (<http://developer.apple.com/tools/>).
- ▶ **Apple's AppleScript Web Page** (<http://www.apple.com/applescript/>) – This web site contains AppleScript documentation, links, success stories and more for anyone looking to learn AppleScript. In addition, it also contains sample downloadable, and fully editable AppleScripts for scripting a variety of Apple applications, including the Finder.
- ▶ **MacScripter.net** (<http://www.macscripter.net/>) – MacScripter.net is probably the most comprehensive AppleScript developer's resource on the net. It contains up to date AppleScript news, links, forums, articles, tips and more. It also includes thousands, and I mean thousands, of downloadable scripts, some of which are commercial, and some of which are shareware or freeware. Many of these sample scripts are fully editable.
- ▶ **Objective-C Beginner's Guide** (<http://www.otierney.net/objective-c.html>) – This website offers a nice tutorial of the Objective-C programming language.



Chapter 18:
In Conclusion

Mailing Lists and Forums

Mailing lists and forums offer an unprecedented learning resource for developers. These mailing lists and forums provide an excellent place to post questions and talk directly with other developers.

► **Apple's Mailing Lists**

(<http://lists.apple.com/>) – Apple hosts a variety of mailing lists for developers, including the following:

Accessibility API List (<http://lists.apple.com/mailman/listinfo/accessibility-dev>)

AppleScript Implementer's List (<http://lists.apple.com/mailman/listinfo/applescript-implementors>)

AppleScript Studio List (<http://lists.apple.com/mailman/listinfo/applescript-studio>)

AppleScript Users List (<http://lists.apple.com/mailman/listinfo/applescript-users>)

Carbon Developer's List
(<http://lists.apple.com/mailman/listinfo/carbon-dev>)

Cocoa Developer's List
(<http://lists.apple.com/mailman/listinfo/cocoa-dev>)

Objective-C Language List (<http://lists.apple.com/mailman/listinfo/objc-language>)

Xcode User's List (<http://lists.apple.com/mailman/listinfo/xcode-users>)

► **Apple's Discussion Boards**

(<http://discussions.info.apple.com/>) – The Apple discussion forums provide another excellent resource for information about a variety of Apple technologies. While most of these are application specific, and aren't necessarily geared toward developers, there is an AppleScript discussion available.

► **MacScript Mailing List**

(<http://listserv.dartmouth.edu/archives/macscript.html>) – This email list, which is hosted by Dartmouth, is host to thousands of AppleScript developers who can provide answers to virtually any AppleScript-related question you may have.

► **MacScripter.net's BBS**

(<http://bbs.applescript.net/>) – This online forum provides yet another excellent resource for asking questions about AppleScript and Finder scripting. Much like the email lists, this forum is frequented by thousands of developers who are always willing to share their scripting knowledge.

► **Objective-C News Group**

(news.comp.lang.objective-c) – This news group focuses discussions of the Objective-C programming language.



Chapter 18:
In Conclusion

In Closing

In closing, it has been a pleasure to walk you through the intricacies of Automator.

For users, we have discussed the Automator application, and explored how to begin building workflows in order to make your everyday routines more efficient. Now, it is time for you to begin bringing the things that we have discussed into reality. I encourage you to jump right in and begin constructing workflows today... the water's fine.

For developers, you are encouraged to begin building custom actions for your own use, as well as for others. As you have seen in this book, creating custom actions is a feat that can be accomplished with a little time and practice. Once you get the hang of it, creating Automator actions will become second nature. So, take hold of this opportunity now. What you develop today may dramatically impact the lives of many Macintosh users tomorrow.



Appendix A Automator Action Development Step-By-Step

1. Determine the action that you want to build.
2. Create an outline of the action's functionality, settings, and input and output values.
3. Determine the type of action that you want to create, AppleScript, Cocoa, or a combination of the two.
4. Create a new Xcode project, using the action template appropriate for the desired type of action.
5. Update the key values for the action's general information, description, requirements, input and output values, etc., in the action's *info.plist* file.
6. Update or remove keys and values in the action's localized *InfoPlist.strings* file.
7. Open the project's main *nib* file in Interface Builder.
8. If no interface is required, then delete the view instance in the project's main *nib*.
9. If an interface is required, then design the action's interface, taking into account Apple's guidelines for interface design.
10. Add parameter keys in the action's *nib* file for each interface element attribute that will be passed to the action's processing code as a parameter.
11. Assign bindings corresponding to the appropriate parameter keys to the desired interface element attributes.
12. Update the *info.plist* file to contain keys and default values for any parameters that were assigned in the action's *nib* file.
13. Add processing code to your action. For AppleScript-based actions, add the code to the **on run** handler contained within the *main.applescript* file. For Objective-C-based actions, add the code



Appendix A:

**Automator
Action
Development
Step-By-Step**

to the `runWithInput:fromAction:error:` method within the *ProjectName.m* file in the project.

14. Begin testing and debugging your action by running it in Automator from within Xcode.
15. Make any necessary adjustments to your action, based on your testing results.



Appendix B Automator Input and Output Uniform Type Identifiers (UTIs)

AppleScript Type Identifiers

Type Identifier	Conforms To	Data Type
com.apple.applescript.object	com.apple.applescript.object	Generic AppleScript Object
com.apple.applescript.alias-object	com.apple.applescript.alias-object	AppleScript Alias
com.apple.applescript.alias-object.image	com.apple.applescript.alias-object	AppleScript Image Alias
com.apple.applescript.alias-object.movie	com.apple.applescript.alias-object	AppleScript Movie Alias
com.apple.applescript.alias-object.pdf	com.apple.applescript.alias-object	AppleScript PDF Alias
com.apple.applescript.data-object	com.apple.applescript.object	AppleScript Data Reference
com.apple.applescript.text-object	com.apple.applescript.object	AppleScript Text Reference
com.apple.applescript.url-object	com.apple.applescript.object	AppleScript URL Reference

Cocoa Type Identifiers

Type Identifier	Conforms To	Data Type
com.apple.cocoa.path	com.apple.cocoa.string	String Path
com.apple.cocoa.string	com.apple.cocoa.string	String
com.apple.cocoa.url	com.apple.cocoa.string	URL

Public Type Identifiers

Type Identifier	Conforms To	Data Type
public.item	com.apple.cocoa.path	File/Folder



Appendix B:
**Automator
 Input and
 Output
 Uniform Type
 Identifiers
 (UTIs)**

Application Specific AppleScript Type Identifiers

Type Identifier	Conforms To	Data Type
com.apple.addressbook.group-object	com.apple.applescript.object	Address Book Group Reference
com.apple.addressbook.item-object	com.apple.applescript.object	Address Book Item Reference
com.apple.addressbook.person-object	com.apple.applescript.object	Address Book Person Reference
com.apple.finder.file-or-folder-object	com.apple.applescript.object	Finder File or Folder Reference
com.apple.ical.calendar-object	com.apple.applescript.object	iCal Calendar Reference
com.apple.ical.event-object	com.apple.applescript.object	iCal Event Reference
com.apple.ical.item-object	com.apple.applescript.object	iCal Item Reference
com.apple.ical.todo-object	com.apple.applescript.object	iCal To Do Reference
com.apple.idvd.menu-object	com.apple.applescript.object	iDVD Menu Reference
com.apple.idvd.slideshow-object	com.apple.applescript.object	iDVD Slideshow Reference
com.apple.iphoto.album-object	com.apple.applescript.object	iPhoto Album Reference
com.apple.iphoto.item-object	com.apple.applescript.object	iPhoto Item Reference
com.apple.iphoto.photo-object	com.apple.applescript.object	iPhoto Photo Reference
com.apple.itunes.item-object	com.apple.applescript.object	iTunes Item Reference
com.apple.itunes.playlist-object	com.apple.applescript.object	iTunes Playlist Reference
com.apple.itunes.source-object	com.apple.applescript.object	iTunes Source File Reference
com.apple.itunes.track-object	com.apple.applescript.object	iTunes Track Reference
com.apple.mail.account-object	com.apple.applescript.object	Mail Account Reference
com.apple.mail.item-object	com.apple.applescript.object	Mail Item Reference
com.apple.mail.mailbox-object	com.apple.applescript.object	Mail Mailbox Reference
com.apple.mail.message-object	com.apple.applescript.object	Mail Message Reference
com.apple.safari.document-object	com.apple.applescript.object	Safari Document Reference
com.apple.spotlight.item	com.apple.applescript.object	Spotlight Item Reference
com.apple.textedit.document-object	com.apple.applescript.object	TextEdit Document Reference

Microsoft Specific AppleScript Type Identifiers

Type Identifier	Conforms To	Data Type
com.microsoft.powerpoint.presentation-object	com.apple.applescript.object	PowerPoint Presentation Reference
com.microsoft.powerpoint.slide-object	com.apple.applescript.object	PowerPoint Slide Reference



License Agreement

This is a legal agreement between you and SpiderWorks, LLC, a Virginia Limited Liability Corporation, covering your use of this electronic book and related materials (the “Book”). Be sure to read the following agreement before using the Book. BY USING THE BOOK, YOU ARE AGREEING TO BE BOUND BY THE TERMS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO THE TERMS OF THIS AGREEMENT, DO NOT USE THE BOOK AND DESTROY ALL COPIES IN YOUR POSSESSION.

Unauthorized distribution, duplication, or resale of all or any portion of this Book is strictly prohibited. No part of this Book may be reproduced, stored in a retrieval system, shared or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

By using the Book, you acknowledge that the Book and all related products constitute valuable property of SpiderWorks and that all title and ownership rights to the Book and related materials remain exclusively with SpiderWorks. SpiderWorks reserves all rights with respect to the Book and all related products under all applicable laws for the protection of proprietary information, including, but not limited to, intellectual properties, trade secrets, copyrights, trademarks and patents.

The Book is owned by SpiderWorks and is protected by United States copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. Therefore, you must treat the Book like any other copyrighted material. The Book is licensed, not sold. Paying the license fee allows you the right to use the Book on your own personal computer. You may not store the Book on a network or on any server that makes the Book available to anyone other than yourself. You may not rent, lease or lend the Book, nor may you modify, adapt, translate, copy, or scan

the Book. If you violate any part of this agreement, your right to use this Book terminates automatically and you must then destroy all copies of the Book in your possession.

The Book and any related materials are provided “AS IS” and without warranty of any kind and SpiderWorks expressly disclaims all other warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Under no circumstances shall SpiderWorks be liable for any incidental, special, or consequential damages that result from the use or inability to use the Book or related materials, even if SpiderWorks has been advised of the possibility of such damages. In no event shall SpiderWorks’s liability exceed the license fee paid, if any.

Copyright 2005 SpiderWorks, LLC. “SpiderWorks” is a trademark of SpiderWorks, LLC. Macintosh is a trademark of Apple Computer, Inc. Microsoft Windows is a trademark of Microsoft Corporation. All other third-party names, products and logos referenced within this Book are the trademarks of their respective owners. All rights reserved.



Index

A

action

- collapsing 39
- expanding 39

actions 10, 15

- adding an interface to 185, 193
- AppleScript-based actions 158, 159
- application interaction with 18
- assigning an application 119
- assigning an icon 121
- assigning a bundle identifier 121
- assigning a category 120
- assigning a description 124
- assigning keywords 120
- Cocoa-based actions 158, 165, 190
- configuring localized strings 184, 193
- configuring properties 182, 191
- constructing 182, 190
- deleting 39
- disabling 41
- displaying description of 31
- enabling 41
- error reporting 162, 170
- icons 121
- importing 62
- input values 16, 168
- interaction between 16
- localizing string values 137
- moving 42
- output values 16, 168
- overview of 15
- planning 181, 190

- processing code overview 158
- processing input 161, 169
- returning a value 161
- settings 17
- testing 188, 198
- third-party 18
- writing code for 187, 195

action categories

- icon size of 25

action confirmation dialog 32

action count indicator 27

actions handling within a workflow 101

actions handling within Automator 101

action list 25

- location of 25

actions naming 108

action settings

- configuring 32
- displaying during processing 33
- showing entire action during processing 34
- showing selected items during processing 34

actions templates 111

adding interface elements 145

Address Book 11, 24, 121

Address Book framework 96

Add Attachments to Front Message action 36, 76, 77, 106, 108

Add Date or Time to Finder Item Names action 70

adjusting actions behavior 132

AMAccepts 125, 128, 131, 161, 168, 169, 173

AMAction 166, 167, 168

AMActionCategory 120

AMAppleScriptAction 143, 167

AMApplication 119, 173



Index

- AMBundleAction 143, 167, 168
- AMCanShowSelectedItemsWhenRun 133
- AMCanShowWhenRun 132
- AMCategory 173
- AMDAler 126
- AMDefaultParameters 156, 160, 164, 180
- AMDescription 124
- AMDInput 125
- AMDNote 126
- AMDOptions 126
- AMDRelatedActions 127
- AMDRequires 126
- AMDResult 125
- AMDSummary 124
- AMIconName 121, 179
- AMKeywords 120
- AMName 119
- AMPalette.palette 142
- AMProvides 125, 128, 131, 168, 173
- AMRequiredResources 133
- AMWarning 135
- ANSI C 96
- API. See application program interface
- Apple's website 18, 85, 202
- Apple's Mailing Lists 205
- AppleScript 11, 19, 48, 83, 88, 90, 92, 93, 94, 95, 96, 97, 158, 181, 207
 - scriptable applications 18
 - scripting Automator 84
 - triggering Cocoa code from 171
 - triggering from within a workflow 83
 - triggering UNIX code from 171
- AppleScript-based actions 97, 101, 109
- AppleScriptKit.sdef file 112
- AppleScript Studio 95, 171, 177, 181
 - Automator event handlers 163
 - terminology 112
 - triggering code from an interface 162
- AppleScript Utility application 54
- Apple Developer Connection 204
- Applications folder 14
- Applications group 24, 25
- Application Kit framework 96, 112
- application program interface 18, 172
- Ask for Confirmation action 32
- Assigning Interface Parameter KeysCocoa bindings
 - assigning parameter keys 151
- attachable applications 95
- automation 9
 - benefits of 14
- Automator
 - AppleScript support 84
 - application icon 14
 - benefits of 14
 - classes 165
 - how it works 15
 - interface 30
 - introduction to 10
 - limitations of 18
 - navigating 20
 - providing feedback about 85
 - related technologies 88
- Automator.framework 165, 168
 - classes 165
- Automator eExecutable 175
- Automator executable 188, 198



Index

Automator menu 85

B

- bindings, Cocoa 90, 151
- building, installing, and testing actions
 - building 176
- building, installing, and testing Xcode
 - building a project 176
- building and running actions
 - running from within Xcode 174
- build and run or build and debug Xcode
 - building a project 174
 - building running a project 174
- build Xcode Action Project 85
- bundles 98, 99, 100, 101
 - loadable bundles 98

C

- C++ 96
- call method command 171
- categories. See action categories
- CFBundleIdentifier 121, 127, 135, 179
- checking the log drawer 61
- CIColorControls 190
- classes 95
- Cocoa 88, 95, 96, 97, 141, 151, 190, 207
- Cocoa.framework 165, 168
- Cocoa bindings 151, 170, 181, 207
 - assigning 185, 194
 - linking to project code 156
- Cocoa frameworks. See frameworks
- Cocoa Objective-C-based actions 97, 109

- common problems, actions 179
- configuring a warning 135
- configuring input values 131
- configuring output values 131
- contextual menu 46
- conversion actions 17, 172
- Copy Finder Items action 128, 136
- CoreTypes bundle 122
- Core Audio framework 96
- Core Image 190
- Core Image framework 96, 165, 190
- Core Video framework 96
- Create Archive action 69, 70
- Create Package action 85
- creating a workflow from Finder items 64

D

- Danny Goodman 11, 19, 93, 203
- Database Events 84
- Dave Mark 8, 11
- Dave Wooldridge 8
- debugging actions 177
- debugging Xcode project 177
- description area 27
- Desktop 17, 48, 67, 69, 71, 73, 74, 75, 78, 79
- Developer Resources, Automator 203
- AppleScript dictionaries 93
- distributed workflows 45
- Dock 45, 55
- Documentation, Automator 203
- Documentation, developer 203
- Documents folder 48



Index

Download URLs action 17
do shell script command 171
DVD Player 24

E

editing properties 114
Edit menu 40
example code 12
example info.plist file 139
example workflows 24, 26

F

feedback 85
File's Owner 143
FileMaker Pro 19
File menu 30, 44, 45, 46, 55, 56, 59, 62, 63, 67, 71, 80
Finder 44, 45, 46, 47, 48, 49, 50, 55, 63, 64, 65, 67, 68, 70, 71, 74, 75,
76, 77, 78, 79, 80, 95, 99, 103, 106, 116, 128, 131, 136
Finder's contextual menu 64, 99
Finder Plug-ins 47
Find Finder Items action 81
Folder Actions 48
Folder Actions Setup application 49
Folder Action plug-ins. See saving workflows
Folder Action plug-ins 48
Foundation framework 96, 172
frameworks 90, 96, 97, 112, 165

G

General Automator Resources 201
Get New Mail action 32, 34

Get Selected Finder Items 64
Get Selected Finder Items action 80, 106
Get Specified Finder Items action 36, 64, 68, 78
goals of the book 10
grouping interface elements 148
groups. See workflow groups

I

iCal 50, 51, 71, 72, 73
iCal Alarm plug-ins. See saving workflows
Automator application icon 55
ignore results from previous action. See input values, ignoring
Image Capture 52
Image Capture Plug-ins. See saving workflows
importing workflows 45
importing actions 62
indicator action status 57
info.plist 101, 112, 141, 160, 207
InfoPlist.strings 112, 137, 207
input actions
 input values 107, 128
input values
 ignoring 37, 68
 special handling 37
 using 37
 working with 35
Interface Builder 11, 88, 90, 91, 95, 101, 185, 207
Interface Design Guidelines 145
iPhoto 9, 11, 20, 121, 128
iPod 17
iTunes 11, 17, 18, 20, 26, 121, 128



Index

J

Java 90, 96
Jenifer Waldie 8

K

keywords, searching by 31

L

Late Night Software, Ltd 177
library list 24
linking interface elements to parameters 153
linking to the Automator executable 175
loadable bundles. *See* bundles
localized strings 137
log activity AppleScript-based actions 181
log command 177
log drawer 28

M

Mach-O 101
Mach object. *See* Mach-O
MacScripter.net 18, 202, 204
Mac OS X 9, 10, 11, 14, 18, 19, 43, 45, 48, 52, 53, 57, 73, 81, 83, 88,
90, 92, 94, 95, 96, 97, 98, 101, 103, 121
Mail 11, 16, 18, 20, 31, 32, 34, 36, 73, 75, 76, 77, 78, 106
main.applescript file 112, 159, 187, 207
main.nib 112, 143
Mark Dalrymple 8, 11, 204
methods 95
Microsoft PowerPoint 128
Move to Trash action 78

My Workflows group 25, 44, 63

N

New Folder action 67
New iPod Note action 34
New Mail Message action 75, 76, 106
New PDF Contact Sheet action 74
nib 101, 112, 143, 152, 185, 207
NSAppleScriptErrorMessage 170
NSAppleScriptErrorNumber 170
NSNumber 170
NSObject 166
NSPathPopUpButton 185
NSView 101, 112, 144

O

object-oriented 95
Objective-C 11, 88, 90, 96, 97, 141, 158, 177, 181
 triggering AppleScript code from 172
Objective-C-based actions
 Cocoa-based actions 101
 Objective-C-based actions 101
opening a workflow 55
Open Finder Items action 106
Open Images in Preview action 127
OSAKit.framework 170, 172
OSAScriptErrorMessage 170
OSAScriptErrorNumber 170
output actions values 107, 128
output values
 special handling 37
 working with 35



Index

P

- packages 98
- palette window, Interface Builder 141
- parameters, Updated method 170
- parameters updated event handler 164
- parameter keys 207
- PDF 52, 53, 57, 73, 74, 75, 76, 78, 79
- PDF action category 24
- PERL 54
- PlistEdit Pro 116, 117, 118
- Printing a Workflow 56
- Print Workflow Plug-ins 52
- projectName.h 112, 165, 168
- projectName.m 112, 165, 168, 208
- projectName_Prefix.pch file 112
- projects 89, 109
- Property List Editor 116, 117
- property list files
 - see info.plist 114

Q

- QuarkXPress 19
- QuartzCore.framework 165, 190

R

- AppleScript recordable applications 94
- recording scripts. See AppleScript
- Rotate Images action 36
- Run button 22, 57
- running a workflow 57

- runWithInput\
 - fromAction\(:error\): method 166, 168, 169, 170, 208
- Run AppleScript action 84
- run handler 112, 159, 207
 - input parameter 160
 - parameters 159
 - parameters parameter 160
 - returning a value 161
- run log window 174, 177
- Run Shell Script action 83

S

- Safari 66, 67, 68, 69, 70, 71, 76
- Sal Soghoian 8
- sample code. See example code
- saved workflows
 - distributing 44
 - opening 55
 - organizing 24
- saving workflows as workflow files 44
- saving workflows
 - as an iCal Alarm plug-ins 71
 - as applications 45, 71
 - as Folder Action plug-ins 80
 - as iCal Alarm plug-ins 50
 - as plug-ins 46, 71
 - as Script Menu plug-ins 53
- Scott Knaster 11, 204
- scriptable applications 93
- scripting language 92
- Script Debugger 177
- Script Editor 50, 92, 93, 94



Index

- Script Menu plug-in. *See* saving workflows
- searching for actions 22, 26, 31
- selecting an actions
 - templates 109
- shell scripts 54, 83
- Show Action When Run checkbox 34
- Show Package Contents 99
- specifying an Action name and directory 110
- specifying required resources 133
- Spotlight 81
- Standard Additions scripting addition 171
- status indicator in menu bar 45
- Stop button 22
- syntax 93

T

- TextEdit 119
- third-party actions 201
- threading architecture 102
- Tim Davis 8
- tips for testing actions 178
- toolbar 20, 22, 57, 70
 - contextual menu 23
 - customizing 22
 - saving changes to 24
- troubleshooting 29, 59
- type identifiers. *See* uniform type identifiers

U

- uniform type identifiers 128
 - AppleScript type identifiers 129, 209
 - application specific type identifiers 130, 210

- Cocoa type identifiers 129, 169, 209
 - Microsoft type identifiers 130, 210
 - public type identifiers 129, 209
- UNIX 83, 171
- Update iPod action 33
- update parameters event handler 164
- update parameters method 170
- URL 15, 17
- UTIs. *See* uniform type identifiers

V

- View menu 22, 25, 28, 61
- View Results action 61

W

- Web Kit framework 96
- Window menu 20
- workflows 10, 15
 - adding actions to 31
 - constructing 30, 67, 74
 - distributing workflow files 44
 - mismatched actions in 37
 - outlining 66, 73
 - planning 66, 73
 - saving. *See* saving workflows
 - testing 70, 79
- workflow groups 24
 - adding 25
- workflow groups
 - importing workflows into 45
- Workflow menu 57



Index

- workflow status indicator
 - in Automator 28, 70
- workflow view 27
 - displaying actions in 28
 - location of 27
- workflow window 20
 - saving layout changes 20

X

- Xcode 11, 85, 88, 89, 90, 95, 97, 104, 108, 109, 110, 111, 114, 115,
116, 117, 122, 137, 139, 143, 174, 182, 207, 208
 - running an action from within 174
- XML 114, 116, 119, 124, 131, 133, 135, 137, 191