Extracted from:

TextMate

Power Editing for the Mac

Chapter 9

Automation Tips and Tricks

You now know how to build three kinds of automations for TextMate, but you can still learn a lot of tricks for using them effectively. You will pick these up over time as you create automations, but I'll jump-start your learning process by showing you some favorite techniques used by the TextMate gurus. This material can take you above the novice automator ranks and give you the ability to build powerful custom tools to handle your unique workflow.

You may not ever need some of these tips, and you certainly won't need them all at once. I recommend skimming this chapter to get an idea of what's covered and then referencing the material later when you are trying to accomplish a related task. I've tried to lay out the information to support this pattern of usage.

9.1 Building Bundle Menus

No one wants to scan through a menu hunting for the automation that does what they need. As a bundle grows, selecting automations from one long list becomes tedious. For that reason, TextMate gives you the ability to organize a bundle's menu with submenus and dividers.

You access the menu-building interface by choosing Bundles → Bundle Editor → Show Bundle Editor (^ ⌥ ⌘ B) and clicking the name of the bundle you want to organize. The Menu Structure box contains all the items in the bundle in their menu order and layout. Excluded Items serves two purposes, which will become clear as you move forward through this chapter.

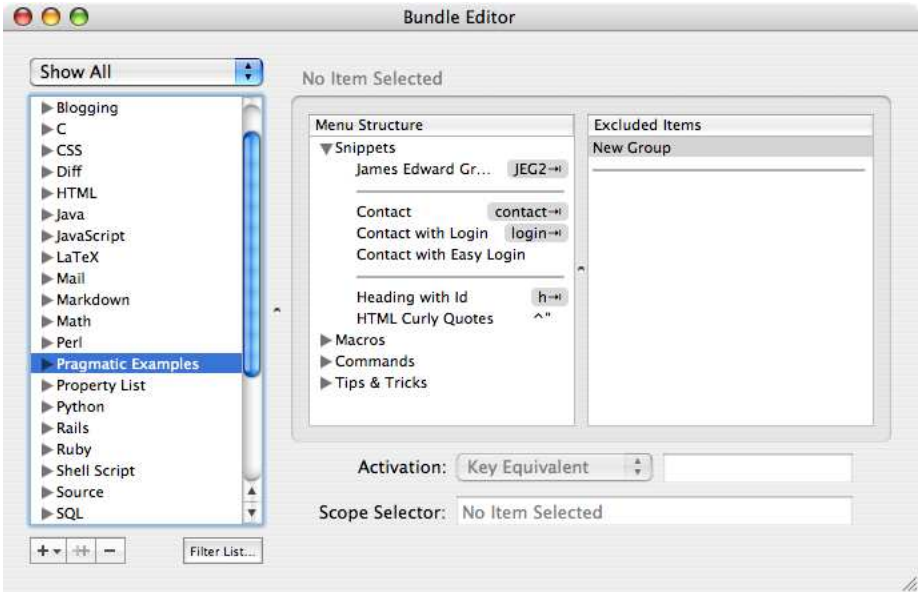

Figure 9.1: Bundle menu structure

Here are the changes you can make to the menu structure:

- Reorder automations as they will appear in the Bundles menu by dragging the automation name and dropping it where you want it to be listed.
- Add dividers to a menu by dragging the divider line from Excluded Items and dropping it into position.
- Create a new submenu by dragging the New Group label out of Excluded Items and dropping it in the menu for which you want to create a submenu.
- Rename a menu item, including newly created submenus, by double-clicking the item name to expose the editor and typing a new name.
- Unfold a submenu so you can arrange its contents by clicking the triangle just to the left of the submenu's name until it points straight down.

Try organizing your Pragmatic Examples bundle to get the hang of these features. You should pick it up in no time. You can see an example of my menu structure in Figure 9.1.

You can also drag automations from Menu Structure into Excluded Items. This will hide them so they do not appear in the Bundles menu. You can still activate automations in Excluded Items using the item's key equivalent or tab trigger. You may want to stick items in here that make sense to activate only via the keyboard (as opposed to using the Bundles menu). Just make sure the user has some way of knowing that the item is there at all. You can also use Excluded Items to depreciate automations you plan to remove from the bundle down the line.

Remember that the point of a good menu structure is to guide the user right to what they want to find. Users are lazy and impatient, so make sure the menus divide the available automations into a logical grouping of an easily digested size.

9.2 TextMate's Environment Variables

Both snippets and commands have access to a collection of environment variables when they run. TextMate sets up most of these variables for you, but you are free to set your own variables and use them in your automations.

To set a variable that will be used in all automations, add the variable's name and value to the Shell Variables list under TextMate → Preferences (⌘ ,). You can reach the list by clicking the Advanced icon at the top of the preferences window and then selecting the Shell Variables tab, as shown in Figure 9.2, on the following page.

You can also set project-level variables used only for automations run on the files of that project. To reach the semi-hidden interface for this, select View → Show Project Drawer (^ ⌥ ⌘ D) unless it is already visible, make sure nothing is selected in the drawer (click the whitespace if you need to deselect items), and click the i button in the lower-right corner of the project drawer.

The SQL bundle is a great example of how these variables might be useful to you. I have TM_DB_SERVER set to mysql in my TextMate preferences, so the bundle knows which database I favor. Then, inside each database-oriented project I work with, I set MYSQL_USER to my login name for MySQL (defaults to your Mac OS X login name), MYSQL_PWD to my password, and MYSQL_DB to the database I am working with.[1]

1. If you are a Postgres fan, consult Bundles → SQL → Help for details on how to set up that database server.

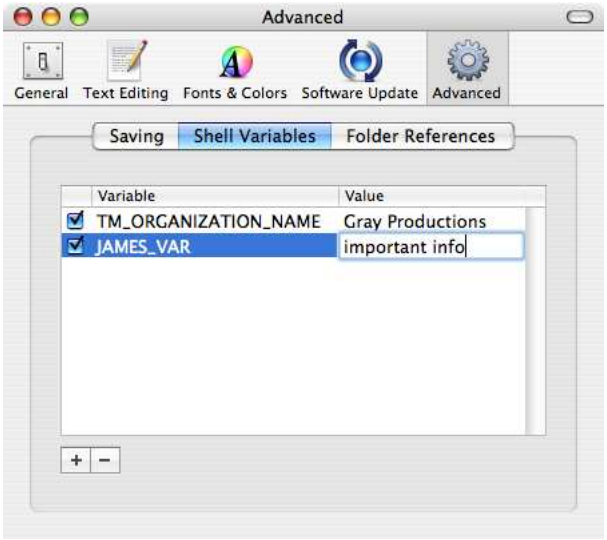

Figure 9.2: Setting an environment variable

With that set up, I can select any SQL query in my project and choose Bundles → SQL → Execute Selection as Query (^ ⇧ Q) to have TextMate display the results. That's a convenient way to reality check database contents while you work without needing to leave TextMate.

Here's a list of the variables TextMate maintains for you and suggestions about how you might use them:

TM_BUNDLE_SUPPORT

When you write a small command, it's fine to crack open the Bundle Editor, jot down some code, and try it. More complicated commands require better organization, though, and you might want to share some code or other resources among a group of related commands. TextMate supports this through this variable.

If the bundle a command is called from contains a top-level Support folder, this variable will point to that folder. You can use this to locate the needed external resources.

For example, to use an external library in a bundle you are building, create a Support/lib directory in the bundle, add the library you

need to this directory, and require the library in your command with code similar to this Ruby example:

```
require "#{ENV['TM_BUNDLE_SUPPORT']}/lib/external_library"
```

Another perk of the Support folder is that Support/bin is added to the path while a command executes. This ensures that you don't even need this variable to reach external programs, as long as you place them in this folder.

TM_SUPPORT_PATH
: The TextMate application also contains a Support folder including several programs, code libraries, and other resources useful in developing automations. This variable always points to the root of that directory, so you can load TextMate's GUI dialog box support library with the following line of Ruby:

```
require "#{ENV["TM_SUPPORT_PATH"]}/lib/dialog"
```

Again, the bin directory of this folder is added to the path while a command executes. This allows you to shell out to bundled commands, including CocoaDialog, Markdown.pl, Textile.rb, and more, from any automation.

TM_CURRENT_LINE
TM_CURRENT_WORD
TM_SELECTED_TEXT
: These variables function just like the fallback menu equivalents for command input described in Section 8.2, *Command Input and Output*, on page 111. Environment variables do have a size limit, which can cause the data in these variables to be truncated in extreme cases. Therefore, it's better to have these sent to your command as input.

TM_SCOPE
: This is the scope the caret is currently inside. The Show Scope command of the TextMate bundle prints the contents of this variable as a tooltip.

TM_LINE_INDEX
TM_LINE_NUMBER
: These variables are indices into the document being edited. TM_LINE_INDEX is a zero-based count to the current caret location in the line. This variable is aware of the line encoding and thus will count multibyte characters correctly. TM_LINE_NUMBER is the line of

the document the caret is currently on, counting from one. In a Ruby command that is sent the document as input, you could use code like the following to examine the text around the caret:

```
doc     = ARGF.readlines
line_no = ENV['TM_LINE_NUMBER'].to_i
line    = doc[line_no - 1]
line_i  = ENV['TM_LINE_INDEX'].to_i

puts "Line before caret:  #{line[0...line_i]}"
puts "Line after caret:  #{line[line_i..-1]}"
```

TM_INPUT_START_LINE
TM_INPUT_START_LINE_INDEX

These variables provide offsets describing where the input sent to your command began in the document. Among other uses, you can use them to locate the caret's position inside the input you received using code like the following:

```
# set line and col with the indices for the caret in the input
line = ENV['TM_LINE_NUMBER'].to_i - ENV['TM_INPUT_START_LINE'].to_i
col  = ENV['TM_LINE_INDEX'].to_i
if ENV['TM_LINE_NUMBER'].to_i == ENV['TM_INPUT_START_LINE'].to_i
  col -= ENV['TM_INPUT_START_LINE_INDEX'].to_i
end
```

TM_COLUMN_NUMBER
TM_COLUMNS

You can use these variables to find the current column location of the caret (counting from one) and the number of columns available in the editing window, assuming Soft Wrap is active. You might prefer TM_COLUMN_NUMBER to the previously mentioned TM_LINE_INDEX in places where you want to know exactly where the caret is. For example, if you are trying to find the indent level where the command is triggered, TM_LINE_INDEX may tell you that you are two characters in, but if those characters happen to be tabs, TM_COLUMN_NUMBER holds exactly how far into the line you are, accounting for the current tab size.

TM_TAB_SIZE
TM_SOFT_TABS

If you need to mimic user settings for indention in some command output, these two variables are helpful. TM_TAB_SIZE will tell you the current size of a tab in the editing window, and TM_SOFT_TABS will tell you whether those tabs are being represented as actual tab characters (variable set to NO or unset) or as the equivalent

number of spaces (a YES setting). Since you can't count on what TM_SOFT_TABS will be when tabs are used, always test for the YES value.

TM_DIRECTORY
TM_PROJECT_DIRECTORY
TM_FILEPATH

You can use these variables to locate the directory containing the currently active file, the top-level project directory for the project containing the file, and the current file itself. It's important to note that these variables may not be set. The user may not have a project open, and the current document may not yet be saved to the disk.

If you need document content, it's better to set up the command input to send you what you need than to try to read it using these variables. The user may have unsaved changes that wouldn't be reflected in the disk file. Still, these variables can be useful for fetching information from the file system or manipulating files based on their location. See Section 9.7, *Hooking Into Save Operations*, on page 136 for details about a kind of command that might need these variables.

TM_SELECTED_FILE
TM_SELECTED_FILES

You can use these variables to find out what is currently selected in the project drawer, assuming the user is working with a project and there is currently a selection of files and folders in the drawer. The singular variable gives only the path to the first selected item, and the second gives a shell-escaped listing of all currently selected files. If you would like to get these files into an Array inside a Ruby command, use the following code:

```
require "shellwords"

selected = Shellwords.shellwords(ENV["TM_SELECTED_FILES"])
```

TM_DROPPED_FILE
TM_DROPPED_FILEPATH
TM_MODIFIER_FLAGS

This family of variables is populated only during the execution of a drag command. You use these variables as your primary means of interacting with the dropped file.

TM_DROPPED_FILE holds a relative path to the file from TM_DIRECTORY. I find it easier to work with an absolute path most of the time, and you can find that in TM_DROPPED_FILEPATH.

When a file is dragged onto a TextMate document, the user may choose to hold down one or more modifier keys on the keyboard. If your command needs to react to these keys, you can find out what was held with the variable TM_MODIFIER_FLAGS. The variable holds a string like "SHIFT|CONTROL|OPTION|COMMAND", assuming all options are pressed. To check for an option using Ruby, you can write this:

```
if ENV["TM_MODIFIER_FLAGS"].include? "OPTION"
  # handle option key pressed here
else
  # handle option key not pressed here
end
```

If you would like to spot-check the variables your command will be passed when invoked, run the Show TM_* Variables command in the TextMate submenu of the Bundles menu in place of the command you would have run. A tooltip will appear with the name and contents of the variables that would have been passed to your command.

9.3 Useful Shell Commands

Between the files stored on the hard drive and what the operating system itself knows, a lot of data is available to snippets and commands. Shell commands are the gateway to that data, and learning how to use them can really give a boost to your text-editing abilities.

For example, signing any generated content with the name of the current user is as easy as shelling out to the Directory Service utility to get the name, and you can throw in a call to sed to clean it up:

```
dscl . read /Users/$USER realname | sed -E 's/^realname: +//'
```

Mac OS X ships with hundreds of applications accessible from the shell. I couldn't begin to tell you what they all do, but here are a handful of commands that are handy to know when editing text, manipulating files, working with the operating system, or even just for learning about other commands:

cat

This utility outputs files given as arguments or the data it receives on STDIN. You could use this to insert content directly into TextMate documents with commands such as cat /usr/share/dict/words.

curl

If you want to manage some network communication from the command line, curl is your best friend. It knows most popular protocols including HTTP and FTP. You can use this to check the availability of a network resource (curl -I http://rubyquiz.com), fetch the entire resource (curl http://rubyquiz.com), download files (curl -O http://media.pragprog.com/titles/textmate/code/textmate-code.tgz), or fill out web forms:

```
curl -d 'command=sum+1+2+3+4+5' -g -L http://yubnub.org/parser/parse
```

echo

Use this to generate a line of content just by passing the line as a command-line argument. For example, use echo 'A line to output'. This command is also a handy way to find the current value of a TextMate environment variable: echo $TM_FILEPATH.

find

This shell command will walk a file hierarchy and return the path to all files matching certain criteria. You could use this to get a list of all Ruby files below the current directory, for example, with a call such as find . -name *.rb.

fmt

You can use this formatter to wrap lines at a specified length. This can be helpful in TextMate to restrict command output to a given width: cat unwraped_document.txt | fmt -w 80.

grep

By feeding grep a regular expression, you can restrict output to only the lines of a file or STDIN that match the provided expression. For example, use cat todo_list.txt | grep -E '^ *TODO'.

Use the -v switch to invert the results to show the unmatched lines. This can be a slick way to prune document content with Filter Through Command (⌥ ⌘ R).

Unix Regular Expressions

Many shell commands can use regular expressions. I often use them with find, grep, and sed.

Be warned, though—the regular expressions these commands accept are not as powerful as TextMate's regular expression engine. Passing an -E flag to these commands will activate their "extended" syntax, which is pretty close to what I covered for TextMate.

Avoid using shortcut character classes such as \d and advanced features such as look-arounds and conditional replacements. The basic elements are the same, though.

You can learn more by feeding Terminal the command man re_format.

head
tail

These tools are for looking at the first *n* lines of a file or STDIN (cat email.txt | head -n 4) or the last *n* lines (cat error.log | tail -n 10). Just pass the number of lines needed after the -n switch. These commands are often used to examine document headers or the latest entries of log files.

When you are working with something like a log, you may be more interested in the newest lines, which are generally at the end of the stream. In this case, the -r switch supported by tail to reverse the lines is helpful to know.

iconv

Use this tool to convert files of one encoding to another. You pass iconv from and to encoding names with command-line switches: iconv -f ISO_8859-1 -t UTF-8 old_file.txt.

man

This command will open the manual pages for other shell commands. You probably won't use this in TextMate too much, but you can use it to look up documentation for all the commands covered here and more. Just name the command you would like to read the documentation for in the call: man curl.

mdfind
mdls

You can use this pair of commands to perform Spotlight searches from the command line. To see all the metadata associated with a given file, just hand the filename to mdls with a call like mdls stock_report.doc.

To perform full Spotlight searches, you need mdfind. You can use that tool to perform simple searches among all metadata fields, as the Spotlight tool in the menu bar does, with calls such as mdfind Rails. You can also perform searches targeting specific fields of metadata with calls such as mdfind "kMDItemFSName == 'test.rb'".

osascript

This tool will allow you to communicate with Apple's AppleScript environment and through that give instructions to many Mac applications. For example, you could play the current system sound with osascript -e beep.

pbcopy
pbpaste

You can use these commands to place data on and retrieve data from Mac OS X's paste board, known to most users as the *clipboard*. You could add a line to the clipboard with this:

```
echo 'http://www.pragmaticprogrammer.com/' | pbcopy
```

and later fetch it back with pbpaste.

Mac OS X has a separate clipboard for search patterns used in Find dialog boxes. These commands can affect that clipboard with the -pboard find option. You may want to use this to generate search patterns for the TextMate Find dialog box.

These commands default to the default encoding of the system (MacRoman for Western users), so you should switch to UTF-8 before using these commands for any non-ASCII content. You can make the change by having your command execute the following before you call pbcopy or pbpaste:

```
export __CF_USER_TEXT_ENCODING=$UID:0x8000100:0x8000100
```

Tempting though it may be, do not stick that line in your shell start-up scripts. It can cause some programs to misbehave.

perl
php
python

You've seen me using Ruby just about everywhere in this book because that's my scripting weapon of choice. I use it often when shelling out to introduce moderately complex logic. Of course, if you're a fan of another scripting language, such as those listed previously, you can use it to do the same.

sed

This is a terrific tool for quick data transformations. sed supports many options for changing the input passed through it including regular expression search and replace: `echo "I have three dogs and two cats." | sed -E 's/[AEIOUaeiou]/X/g'`.

sort

You can use this command to order a collection of lines from a file or STDIN. Commands may need this to provide a human with friendly ordering of command output. For example, use `sort names.txt`.

tee

A poor man's backup, this command can duplicate a Unix stream. It is usually used to dump some data to a file and continue processing: `echo '^config=.+' | tee find_pattern.txt | pbcopy -pboard find`.

touch

This tool is actually intended to update a file's modification time so tools such as compilers will examine the file again. It sees at least as much use, though, in creating blank files. This might be helpful to TextMate commands wanting to initialize some directory structure with files to be edited: `touch plugin.rb test.rb`.

uniq

This command will remove duplicate adjacent lines from a file or STDIN. This is helpful when you generate a lot of data but need only a single entry for each line. Since it catches only adjacent lines, you generally want to sort the data first to bring like lines together: `cat event_days.txt | sort | uniq`.

uuidgen

This command is helpful anytime you need to generate a unique ID. Different computers and different times of execution will affect the ID generated, so it's safe to count on them being unique.

xargs

It's common to programmatically build up arguments to a shell command and have xargs pass them on to the command in question. xargs will separate STDIN on whitespace and forward each chunk of data it finds as an argument to the named utility. For example, you could use this to build a find command that will match a name pattern currently on the clipboard with pbpaste -pboard find | xargs find . -name.

xxd

If you want to make sense of a binary data file, this tool is often invaluable. xxd will create a hex dump for the passed file, which you can then examine in an editor such as TextMate: xxd codex.umz | mate.

9.4 Using TextMate's Document Parsing

You've already seen how TextMate can send all manner of input to the commands you write; but instead of getting raw text, you can also ask TextMate to tell you how it parses the text. This can make it easier to find the pieces of a document you need to see, since TextMate breaks them down for you. I'll now show you how this works.

First, let's create a command that just returns the input it receives and set that input to be the entire document. You can place the output in a new window. The body of the command is one word: cat. Open any document, run your new creation, and verify that TextMate makes a duplicate of the document in a new window.

Not impressed yet? Just wait until you see my next trick....

To activate parsed input, you must make a change to the actual command file on your hard disk. You'll use the same technique as in Section 7.1, *The Macro Editor*, on page 102.

I named my command Show Parsed Input, so I have to edit the file ~/Library/Application Support/TextMate/Bundles/Pragmatic Examples.tmbundle/Commands/Show Parsed Input.tmCommand. Just choose File → Open and navigate to the document.

To change the behavior of the command, add these two magic lines just before the closing </dict> tag:

```
<key>inputFormat</key>
<string>xml</string>
```

These lines adjust a hidden setting for the command. By setting it to xml, you tell TextMate that you are prepared to receive extra information with the passed text, in the form of XML markup.

Save the file, close it, and ask TextMate to reread it by choosing Bundles → Bundle Editor → Reload Bundles.

Open any document that TextMate will syntax highlight, and run your command one more time. This time you should receive your document content decorated in XMLish markup showing the scopes into which TextMate has divided the document. I say "XMLish" because the scope names don't make good XML tag names. However, the document content is properly escaped, and you can easily turn it into something you can work with using this Ruby code:

```
xml = ARGF.read.gsub(/<(.*?)>/) do |tag|
  if    tag.size == 2  then ""
  elsif tag[1]   == ?/ then "</scope>"
  else                      "<scope name='#{$1}'>"
  end
end
```

Once you have a structure like that, you can load an XML library and hunt down what you want with XPath searches. Here's a sample command that uses TextMate's ability to parse Ruby source code to present the user with an outline of classes and the methods they contain:

Download automation_tips_and_tricks/show_class_structure.rb

```
#!/usr/bin/env ruby -w

require "rexml/document"

xml = ARGF.read.gsub(/<(.*?)>/) do |tag|
  if    tag.size == 2  then ""
  elsif tag[1]   == ?/ then "</scope>"
  else                      "<scope name='#{$1}'>"
  end
end
doc = REXML::Document.new(xml)

met, cla = "entity.name.function.ruby", "entity.name.type.class.ruby"
doc.elements.each("//scope[@name='#{met}' or @name='#{cla}']") do |tag|
  if tag.attributes["name"] == cla
    puts tag.text
  else
    puts "  " + tag.text
  end
end
```

Open the Bundle Editor (^ ⌥ ⌘ B), create a new command with that code in the Command(s) field, set Input to Entire Document, and set Output to Create New Document. It needs to be scoped to source.ruby, and you will need to open the command file TextMate saves to the hard disk to add the XML input lines.

Once you have it set up, try running it on some Ruby libraries. Mac OS X ships with standard Ruby libraries you can use. For example, try running it on /usr/lib/ruby/1.8/set.rb.

9.5 bash Support Functions

Before TextMate runs a bash command, it triggers an internal script to set up the environment for you. This does some nice things, such as setting up the path as I described in Section 9.2, *TextMate's Environment Variables*, on page 121. It also defines a handful of functions you can use in your command.

First, require_cmd will allow you to check whether a shell command is in the path and thus available for use. If the shell command is not found, an error is reported to the user, and your command aborts. It's a good idea to call this before using a shell command that does not ship with Mac OS X so you can make sure the user has installed it. You may even want to check for commands that don't ship with all versions of the operating system, just in case the user has a different version. If you wanted to check for mysqldump, for example, before using it to dump a database table into an SQL file, you would enter this:

```
require_cmd mysqldump
```

Another family of useful functions are those that allow you to change the output type of your command. It's common for commands to check the conditions they were run under and then bail out with an error message if a requirement is missing. You don't need to create a new document just to show a small error message, even if that is the command's regular output. In such a case, you can change the output format with something like this:

```
exit_show_tool_tip "Sorry, this command only works between 8 AM and 5 PM."
```

The user will see your message, as a tooltip in this case, and the command will exit. This works for all commands except those set to HTML output, so remember to switch to HTML instead of away from it.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

TextMate
http://pragmaticprogrammer.com/titles/textmate
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragmaticprogrammer.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragmaticprogrammer.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragmaticprogrammer.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/textmate.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com