2016 STC Technical Communication Summit
15–18 May 2016 • Anaheim, CA
© 2016 Society for Technical Communication

PROGRAMMING LANGUAGE STORYTIME

Sarah Kiniry

Contrary to common opinion, not everyone needs to write code to be able to read it. "Digging into" the code is often the best way to extract information when writing software documentation, but the idea of reading all that code can be daunting. However, a few simple tricks and an understanding of the patterns involved can allow technical communicators to get started reading code in no time.

Communicating about software, whether for user-oriented instructions, developer-oriented API documentation, or interactive walkthrough content, is a huge part of the technical communication industry. More often than not, the technical communicator on this type of project acts as a de facto translator between software developers and the rest of the world.

To craft this communication effectively, technical communicators need to learn how to gather information independently, without constant input from the developers themselves. The logical answer to this problem, of course, is to dive into the software itself to gather information and develop a strong vocabulary in the developer's native language, code. Too often, this is where a disconnect arises.

Reading Is Not Writing

When faced with a screen full of text in an unfamiliar language, the learning curve is so daunting that many simply decide not to bother. Those who do persevere often spend long hours learning how to be a competent developer even though they do not really need that skill set.

We often forget a very important factor in the equation: reading is not the same as writing. Technical communicators do not always need to become skillful, or even novice, developers. Instead, they need to become adept readers of the appropriate programming language. As with any form of communication, being a talented writer is not the same as being a good reader. A tourist in Rome probably does not need to be able to write an essay in fluent Italian in order to read the city's signage and find the right streets. Similarly, looking for answers in code does not require the ability to craft it. If you can learn to read, you can learn to read code.

With the skills to answer their own questions, technical communicators can often operate independently from developers to gather their own information. Three questions are all that a reader needs to gather information from code and find the "story" that the code tells:

- What does the application do?
- What text will the user see?
- Which files or settings are involved?

Technical review by subject matter experts will always be a part of the writing process. However, bringing them into the process later, to review existing content rather than fill in myriad informational gaps, saves everyone time. It can even help to forge stronger relationships between team members. In Agile environments, this modified flow of information allows documentation writers to escape from all-too-common miniature-waterfall scenarios and operate alongside the team.

What Does the Application Do?

Writing a computer program is merely writing a set of instructions, not unlike a recipe. By extension, reading code allows one to figure out what the final product will be, based on those instructions. (Wang, 9) The most important question for a technical communicator to answer is, "What does this application do?" Answering this one question generally provides most of the necessary information. The application's action will also dictate which other questions should follow, depending on the audience. If an application stores configuration information to a database, the end user might need to know the specifics about the information stored, but not how the program writes it to the database. Conversely, if writing for a highly technical audience, those database details may be far more important than the individual settings.

What Text Will the User See?

The text that users see in graphical interfaces, or even in a command line interface, is always an important part of communicating about any application. User text provides clues for the code's reader just as it does for the end user. It can also serve as a signpost to locate other elements in an interface. For example, knowing that a button displays with a certain label, or knowing that a certain warning message displays after a user action, could allow the reader to more easily find that place in the code.

In many cases, identifying this text will also allow writers to update and improve those messages during the development process. Looking for this type of text may also reveal the need for labels or help text that are missing in other areas of an interface, improving user experience.

Which Files or Settings?

Files and settings are also useful clues, particularly when identified alongside the action that uses or affects them. A setting's name within the code often indicates its function. The fact that a setting saves to a specific configuration file might indicate more about the setting than its name alone. Additionally, finding the locations involved in code allows you to track down and examine related settings or sections of code. In complex applications, developers almost always spread functionality across many files.

It may seem surprising to anyone new to reading code, but the more familiar one becomes with a specific codebase and the style that a specific group of developers uses, the easier it is to draw helpful inferences from these conventions. An avid reader of a codebase might quickly learn, for example, that any new settings in a particular configuration file will correlate to a setting in a specific user interface. In products that include an externally-accessible API, knowing the specific files that contain those API functions means the reader can quickly "weed out" the need to investigate new functions in other files, which will be for internal use only.

Clues for Reading Code

When looking at code, questions can be answered through looking at comments, function and setting names, and variables and their values. The entire story of the code's actions, from start to finish, is all there waiting to be read. Because modern computers can use "high-level" programming languages, which more closely resemble human languages, reading code does not mean reading long lists of binary 1s and 0s (Wang, 15).

The key to finding these answers is to remember that a large portion of the code can often be "glossed over." In many cases, recognizing a few key clues is enough to piece together the puzzle, without understanding every single character of the code. For example, in a user interface that contains a password field, understanding the specifics of how the page validates that password against a database of users is not likely to be important, but recognizing that the password field exists and uses a specific label is.

Of course, because so much of the programming process is at the discretion of the developer, the usefulness of these items depends on the specific developer who wrote the code. Some developers include detailed comments and helpful, human-readable function and setting names. Other code, in comparison, may be quite obscure to all but the most skilled developers. Sometimes, though, a few nudges from readers of code is enough to encourage developers to find their way back to best practices and more helpful, human-readable code.

The clues to use to find an application's "story" can be defined in three broad categories:

- Comments, the human-readable notes that developers include in code for their own use.
- Functions, the reusable actions that can be defined by either developers or the programming language itself.
- Variables, the settings and values that the code uses to run, or makes changes to.

Comments

Every programming language uses a set of special characters ("comment characters") that set descriptions or reference text apart from the actual code. The computer does not use these sections of code, skipping over them when the application runs. Instead, they remain in the code for developers to read when they make changes. These comments can act as developers' to-do lists, notes about harder-to-read code, or perhaps justifications of particular choices that the original developer made.

```
// Set a string.
var str = "This is a string of text.";

// Retrieve only the word "string"
var res = str.slice(10,16);
```

Figure 1. Comments within simple JavaScript code.

Developers also use comment characters to "comment out" code, causing the computer to temporarily ignore a certain section of code without the need to delete it from the file and then add it again later. While no programming language requires these comments to be present for a functional program, they are an important "best practice" that most developers almost always follow (Wang, 277).

- For readers of code, these comments can act as a roadmap to the code's actions. Some applications even include full in-code documentation to describe the application. In most programming languages, two styles of comments exist: single-line comments and multi-line comments.

//	Java, JavaScript, C, C#, PHP
#	PHP, Perl, Ruby, Python

Figure 2. Single-line comment characters in popular programming languages.

- Single-line comments, sometimes referred to as "line comments," use a comment character at the beginning of the comment. The comment ends at the line break, and the computer will begin to use the code normally at the beginning of the next line. Single-line comments often begin on the same line as code, noting information about that particular line.

/* Comment */	Java, JavaScript, C, C#, PHP
// Comment \ Comment	C
=pod =cut	Perl
=begin =end	Ruby
""" """	Python

Figure 3. Multi-line comment characters in popular programming languages.

- Multi-line comments, sometimes referred to as "block comments," use comment characters to mark the beginning and end of the comment. The computer ignores everything between the two comment characters, even if that section includes otherwise-functional code. Developers generally use this style of comment for larger sections of in-code documentation, or to comment out large sections of code.

Functions

The term "function" can refer to any reusable code that performs one or more actions. In many programming languages, however, the term for these actions may differ; "subroutine," "method," "subprogram," and "procedure" are just a few of the other terms that can describe variations on this concept.

The names of functions within code can act as keywords to give clues about the code's actions and the information on which the actions are performed. A function might be an externally-usable API function, or it could be completely internal. All programming languages include at least some pre-determined functions, such as the ability to open and read files, or the ability to display text to a user. They may also contain predefined control structures, which allow for certain actions to take place only under certain conditions. For example, in Perl, the "while" control structure causes the specified action to be repeated as long as a certain condition exists (Schwartz, 37).

```
// Count down from 10.
my $number = 10;

while ($number > 0) {
  print '$number…';
  $number -= 1;
}
print 'Countdown complete!';
```

Figure 4. The "while" control structure in Perl.

Developers can also craft their own reusable code to allow them to perform specific actions multiple times without the need to write the full code for the action each time. These custom functions act as miniature programs within the larger application (Wang, 132).

```
def my_fn(name)
    var = "Howdy, " + name
    return var
    end
```

Figure 5. Defining a function in Ruby.

When the code creates, or "defines," a function, a delineated section of code contains the function's actions. Many languages, included in the "curly brace family" of languages, use curly braces to delineate these sections (Wang, 63). Other languages, like Ruby, use other methods to mark the end of the function. While the format used by each programming language varies, function declarations generally perform several basic actions:

- Naming the function.
- Accepting input data.
- Performing one or more actions on the input data, or using it to modify other data or files.
- Returning a response, which might include data or a message of success or failure.

```
sub do_something {
    my $value = @_;
    if $value {
        return "Yay!";
    }
    …

do_something($value);
```

Figure 6. Defining and then using a Perl function.

When code uses a predefined function, it only includes the function name and any data that the function will use. Again, the specific formatting for each programming language varies, though not as widely as for function definitions.

```
# Print a string to the user.
print("Welcome to my Python program!")
```

Figure 7. A print function in Python.

All programming languages include at least one predefined function that displays text to the end user. Learning to identify a programming language's display (or "print") functions can also make it easier to find user text within the application code. To identify user text, the best method is to learn to identify the programming language's built-in print or display functions. Some applications may also use proprietary output functions for this purpose. For example, if an application uses a localization system to allow translation of its user interface, it may be important to learn how to recognize and read the functions for that localization system.

Variables

Often, functions accept data as input and perform actions on that data. Functions may also include hardcoded data, or may pull data from other locations, like configuration files, databases, or even the output of other functions. Variables store each piece of data that the code will use, allowing programs to hold as many pieces of data at once as the program needs (Wang, 142). Like functions, many terms exist to describe different types of variables, and some programming languages prefer certain terms over others. Variables may represent a single piece of data, an array of many individual values, or a hash, which contains multiple key and value pairs.

`variable_name`	Java, JavaScript, C, C#, Python
`$variable_name`	PHP, Perl, Ruby
`@variable_name`	Perl, Ruby
`%variable_name`	Perl

Figure 8. Variables in several popular programming languages.

The names of these variables give more information about the data on which the application operates. For example, a variable with the name "filename" probably stores a file's name. Some styles of development even use structured naming conventions, such as Hungarian notation, in which the names of variables (or even functions) include abbreviated information about the type or purpose that precedes the programmer-selected name.

Files	`example.txt`, `example.jpg`, `example.php`
Directories	Linux: `/example/directory` or `example/directory/` Windows: `C:\example\directory` or `..\example\directory\`
Settings	`managed`, `unmanaged`; `blue`, `green`, `red`; `0`, `1`

Figure 9. Examples of important value types.

Variables can store three broad types of values:

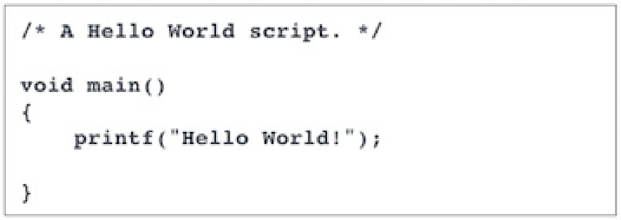
- Text values, which include any number of characters (Schwartz, 24).
- Numbers, including integers and decimal values.
- Boolean values, which are technically integer values, but use those numbers to store values that mean "yes" or "no" (Wang, 149).
- Which values provide you with the most information depends largely on the type of software. In many applications, the files or directories that the code operates on will give you many details.

For example, a file path may help you find the exact images that users will see or the specific configuration files that the code uses to retrieve or save settings. In other applications, settings that are specified in the code may be more important, like Boolean values to show whether or not a user has selected certain preferences, or variables that differentiate behavior between different user account types.

Getting Started With Hello World

When learning to read a specific programming language, one of the best places to start is with a "Hello World" script. "Hello World" is a sort of "See Spot Run" for code. Almost any beginning programming course will start with one of these. In every language, the story is the same. The script simply performs whatever actions are necessary to display the text "Hello World" to the user. Because the story is the same each time, the reader already knows the answers to their questions, which allows them to focus on recognizing the patterns for that programming language:

1. What does the application do? It displays a message to the user.
2. What text will the user see? "Hello World"
3. Which files or settings? No other files or settings are involved.



```
/* A Hello World script. */

void main()
{
    printf("Hello World!");

}
```

Figure 10. An example of a "Hello World" script in C#.

Of course, even a simple action can be performed in several ways in a single programming language. In fact, the idea that "there's always more than one way to do it" is the philosophy of the entire Perl community (Wang, 559). Because of this, it's always beneficial to find additional examples of simple scripts, to see as many of the variations as possible.

Resources

"C Programming Tutorial." Programiz (2016). http://www.programiz.com/c-programming.

"C# Programming Guide." Microsoft Developer Network (2016). https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx.

Harrington, Andrew N. "Hands-on Python Tutorial." Computer Science Department, Loyola University Chicago (08 September 2014). http://anh.cs.luc.edu/python/hands-on/3.1/handsonHtml/index.html.

Heddings, Lowell. "Ruby Function (method) Syntax." How-To Geek (2016). http://www.howtogeek.com/howto/programming/ruby/ruby-function-method-syntax/.

"JavaScript." Mozilla Developer Network (2016). https://developer.mozilla.org/en-US/docs/Web/JavaScript.

"JavaScript Tutorial." W3Schools (2016). http://www.w3schools.com/js/default.asp.

"PHP 5 Tutorial." W3Schools (2016). http://www.w3schools.com/php/default.asp.

"PHP Manual." The PHP Group (2016). http://php.net/manual/en/index.php.

Ponnappa, Sidu and Jasim A Basheer. "Ruby Primer." RubyMonk (2012). https://rubymonk.com/learning/books/1-ruby-primer.

"Trail: Learning the Java Language." Oracle Java Documentation (2015). https://docs.oracle.com/javase/tutorial/java/index.html.

References

Schwarts, Randal L., brian d foy, and Tom Phoenix. Learning Perl (Sebastopol, CA: O'Reilly), 2011.

Wang, Wallance. Beginning Programming All-In-One Desk Reference (Hoboken, NJ: Wiley), 2008.

Author Contact Information

Sarah Kiniry
Technical Writer II
cPanel, Inc.
2550 North Loop West Ste 4006
Houston, TX 77092
+1 713.742.3617

Author Biography

Sarah Kiniry is a technical writer for cPanel, Inc., a web hosting software company. In this role, she provides full documentation support for an Agile development team, as well as participating in content strategy and other initiatives for the company as a whole.

In a past life, Sarah was a stage manager and audience development manager at several of Houston's professional theaters and performing arts venues. She believes that technical communication is really just another performing art, and that documenting an interface isn't that much different from telling any other story.

You can read more about Sarah's journey into software documentation at technicolorwriter.com.