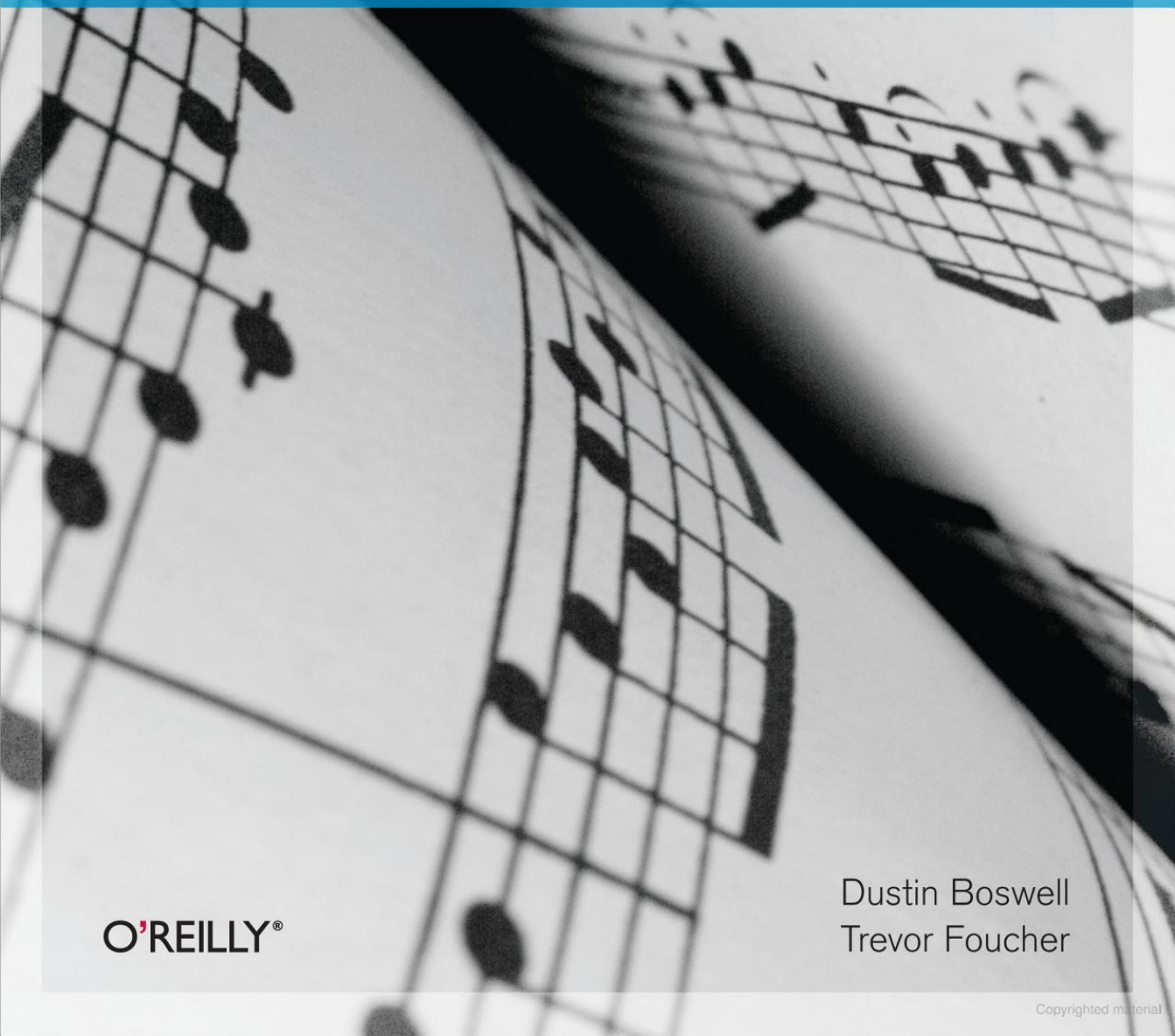
/THEORY/IN/PRACTICE

The Art of Readable Code

Simple and Practical Techniques for Writing Better Code



Dustin Boswell
Trevor Foucher

O'REILLY®

The Art of Readable Code

Dustin Boswell and Trevor Foucher

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

The Art of Readable Code
by Dustin Boswell and Trevor Foucher

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mary Treseler
Production Editor: Teresa Elsey
Copyeditor: Nancy Wolfe Kotary
Proofreader: Teresa Elsey

Indexer: Potomac Indexing, LLC
Cover Designer: Susan Thompson
Interior Designer: David Futato
Illustrators: Dave Allred and Robert Romano

November 2011: First Edition.

Revision History for the First Edition:
2011-11-01 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9780596802295* for release details.

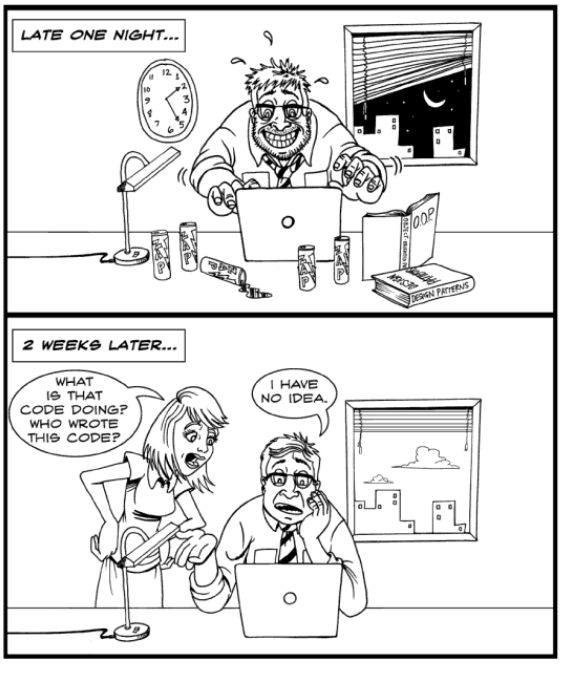
ISBN: 978-0-596-80229-5

[LSI]

1320175254

CONTENTS

PREFACE



We've worked at highly successful software companies, with outstanding engineers, and the code we encounter still has plenty of room for improvement. In fact, we've seen some really ugly code, and you probably have too.

But when we see beautifully written code, it's inspiring. Good code can teach you what's going on very quickly. It's fun to use, and it motivates you to make your own code better.

The goal of this book is help you make your code better. And when we say "code," we literally mean the lines of code you are staring at in your editor. We're not talking about the overall architecture of your project, or your choice of design patterns. Those are certainly important, but in our experience most of our day-to-day lives as programmers are spent on the "basic" stuff, like naming variables, writing loops, and attacking problems down at the function level. And a big part of this is reading and editing the code that's already there. We hope you'll find this book so helpful to your day-to-day programming that you'll recommend it to everyone on your team.

What This Book Is About

This book is about how to write code that's highly readable. The key idea in this book is that **code should be easy to understand**. Specifically, your goal should be to minimize the time it takes someone else to understand your code.

This book explains this idea and illustrates it with lots of examples from different languages, including C++, Python, JavaScript, and Java. We've avoided any advanced language features, so even if you don't know all these languages, it should still be easy to follow along. (In our experience, the concepts of readability are mostly language-independent, anyhow.)

Each chapter dives into a different aspect of coding and how to make it "easy to understand." The book is divided into four parts:

Surface-level improvements
: Naming, commenting, and aesthetics—simple tips that apply to every line of your codebase

Simplifying loops and logic
: Ways to refine the loops, logic, and variables in your program to make them easier to understand

Reorganizing your code
: Higher-level ways to organize large blocks of code and attack problems at the function level

Selected topics
: Applying "easy to understand" to testing and to a larger data structure coding example

How to Read This Book

Our book is intended to be a fun, casual read. We hope most readers will read the whole book in a week or two.

The chapters are ordered by "difficulty": basic topics are at the beginning, and more advanced topics are at the end. However, each chapter is self-contained and can be read in isolation. So feel free to skip around if you'd like.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*The Art of Readable Code* by Dustin Boswell and Trevor Foucher. Copyright 2012 Dustin Boswell and Trevor Foucher, 978-0-596-80229-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://shop.oreilly.com/product/9780596802301.do

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We'd like to thank our colleagues who donated their time to review our entire manuscript, including Alan Davidson, Josh Ehrlich, Rob Konigsberg, Archie Russell, Gabe W., and Asaph Zemach. Any errors in the book are entirely their fault (just kidding).

We're grateful to the many reviewers who gave us detailed feedback on various drafts of our book, including Michael Hunger, George Heineman, and Chuck Hudson.

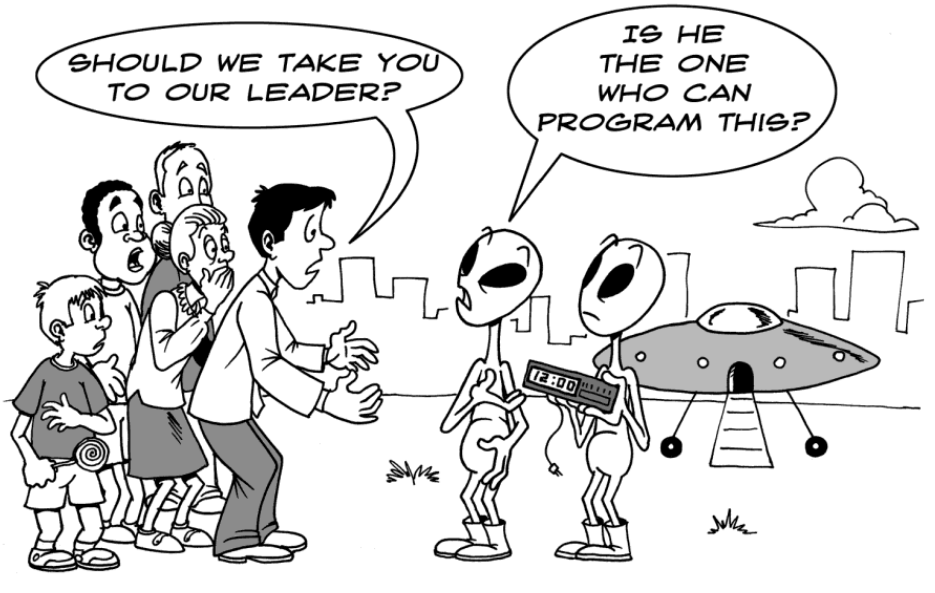
We also got numerous ideas and feedback from John Blackburn, Tim Dasilva, Dennis Geels, Steve Gerding, Chris Harris, Josh Hyman, Joel Ingram, Erik Mavrinac, Greg Miller, Anatole Paine, and Nick White. Thanks to the numerous online commenters who reviewed our draft on O'Reilly's OFPS system.

Thanks to the team at O'Reilly for their endless patience and support, specifically Mary Treseler (editor), Teresa Elsey (production editor), Nancy Kotary (copyeditor), Rob Romano (illustrator), Jessica Hosman (tools), and Abby Fox (tools). And also to our cartoonist, Dave Allred, who made our crazy cartoon ideas come to life.

Lastly, we'd like to thank Melissa and Suzanne, for encouraging us along the way and putting up with incessant programming conversations.

CHAPTER ONE

Code Should Be Easy to Understand



Over the past five years, we have collected hundreds of examples of "bad code" (much of it our own), and analyzed what made it bad, and what principles/techniques were used to make it better. What we noticed is that all of the principles stem from a single theme.

KEY IDEA

Code should be easy to understand.

We believe this is the most important guiding principle you can use when deciding how to write your code. Throughout the book, we'll show how to apply this principle to different aspects of your day-to-day coding. But before we begin, we'll elaborate on this principle and justify why it's so important.

What Makes Code "Better"?

Most programmers (including the authors) make programming decisions based on gut feel and intuition. We all know that code like this:

```
for (Node* node = list->head; node != NULL; node = node->next)
    Print(node->data);
```

is better than code like this:

```
Node* node = list->head;
if (node == NULL) return;

while (node->next != NULL) {
    Print(node->data);
    node = node->next;
}
if (node != NULL) Print(node->data);
```

(even though both examples behave exactly the same).

But a lot of times, it's a tougher choice. For example, is this code:

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

better or worse than:

```
if (exponent >= 0) {
    return mantissa * (1 << exponent);
} else {
    return mantissa / (1 << -exponent);
}
```

The first version is more compact, but the second version is less intimidating. Which criterion is more important? In general, how do you decide which way to code something?

The Fundamental Theorem of Readability

After studying many code examples like this, we came to the conclusion that there is one metric for readability that is more important than any other. It's so important that we call it "The Fundamental Theorem of Readability."

> KEY IDEA
>
> **Code should be written to minimize the time it would take for someone else to understand it.**

What do we mean by this? Quite literally, if you were to take a typical colleague of yours, and measure how much time it took him to read through your code and understand it, this "time-till-understanding" is the theoretical metric you want to minimize.

And when we say "understand," we have a very high bar for this word. For someone to *fully understand* your code, they should be able to make changes to it, spot bugs, and understand how it interacts with the rest of your code.

Now, you might be thinking, *Who cares if someone else can understand it? I'm the only one using the code!* Even if you're on a one-man project, it's worth pursuing this goal. That "someone else" might be *you* six months later, when your own code looks unfamiliar to you. And you never know—someone might join your project, or your "throwaway code" might get reused for another project.

Is Smaller Always Better?

Generally speaking, the less code you write to solve a problem, the better (see Chapter 13, *Writing Less Code*). It probably takes less time to understand a 2000-line class than a 5000-line class.

But fewer lines isn't always better! There are plenty of times when a one-line expression like:

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

takes more time to understand than if it were two lines:

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

Similarly, a comment can make you understand the code more quickly, even though it "adds code" to the file:

```
// Fast version of "hash = (65599 * hash) + c"
hash = (hash << 6) + (hash << 16) - hash + c;
```

So even though having fewer lines of code is a good goal, minimizing the time-till-understanding is an even better goal.

Does Time-Till-Understanding Conflict with Other Goals?

You might be thinking, *What about other constraints, like making code efficient, or well-architected, or easy to test, and so on? Don't these sometimes conflict with wanting to make code easy to understand?*

We've found that these other goals don't interfere much at all. Even in the realm of highly optimized code, there are still ways to make it highly readable as well. And making your code easy to understand often leads to code that is well architected and easy to test.

The rest of the book discusses how to apply "easy to read" in different circumstances. But remember, when in doubt, the Fundamental Theorem of Readability trumps any other rule or principle in this book. Also, some programmers have a compulsive need to fix any code that isn't perfectly factored. It's always important to step back and ask, *Is this code easy to understand?* If so, it's probably fine to move on to other code.

The Hard Part

Yes, it requires extra work to constantly think about whether an imaginary outsider would find your code easy to understand. Doing so requires turning on a part of your brain that might not have been on while coding before.

But if you adopt this goal (as we have), we're certain you will become a better coder, have fewer bugs, take more pride in your work, and produce code that everyone around you will love to use. So let's get started!

PART I

Surface-Level Improvements

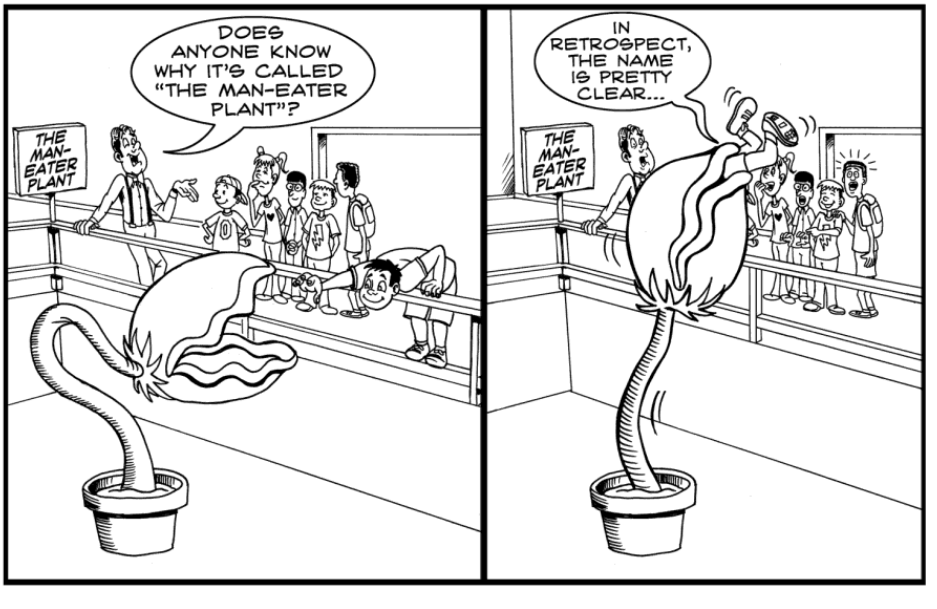
We begin our tour of readability with what we consider "surface-level" improvements: picking good names, writing good comments, and formatting your code neatly. These types of changes are easy to apply. You can make them "in place," without having to refactor your code or change how the program runs. You can also make them incrementally, without a huge time investment.

These topics are very important because **they affect every line of code in your codebase**. Although each change may seem small, in aggregate they can make a huge improvement to a codebase. If your code has great names, well-written comments, and clean use of whitespace, your code will be *much* easier to read.

Of course, there's a lot more beneath the surface level when it comes to readability (and we'll cover that in later parts of the book). But the material in this part is so widely applicable, for so little effort, that it's worth covering first.

CHAPTER TWO

Packing Information into Names



Whether you're naming a variable, a function, or a class, a lot of the same principles apply. We like to think of a name as a tiny comment. Even though there isn't much room, you can convey a lot of information by choosing a good name.

KEY IDEA

Pack information into your names.

A lot of the names we see in programs are vague, like `tmp`. Even words that may seem reasonable, such as `size` or `get`, don't pack much information. This chapter shows you how to pick names that do.

This chapter is organized into six specific topics:

- Choosing specific words
- Avoiding generic names (or knowing when to use them)
- Using concrete names instead of abstract names
- Attaching extra information to a name, by using a suffix or prefix
- Deciding how long a name should be
- Using name formatting to pack extra information

Choose Specific Words

Part of "packing information into names" is choosing words that are very specific and avoiding "empty" words.

For example, the word "get" is very unspecific, as in this example:

```
def GetPage(url):
    ...
```

The word "get" doesn't really say much. Does this method get a page from a local cache, from a database, or from the Internet? If it's from the Internet, a more specific name might be `FetchPage()` or `DownloadPage()`.

Here's an example of a `BinaryTree` class:

```
class BinaryTree {
    int Size();
    ...
};
```

What would you expect the `Size()` method to return? The height of the tree, the number of nodes, or the memory footprint of the tree?

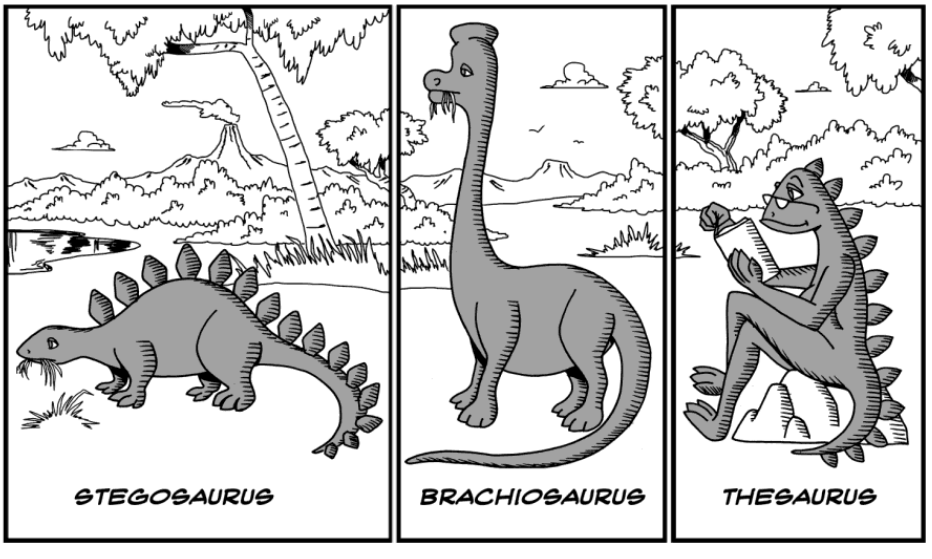
The problem is that `Size()` doesn't convey much information. A more specific name would be `Height()`, `NumNodes()`, or `MemoryBytes()`.

As another example, suppose you have some sort of `Thread` class:

```
class Thread {
    void Stop();
    ...
};
```

The name `Stop()` is okay, but depending on what exactly it does, there might be a more specific name. For instance, you might call it `Kill()` instead, if it's a heavyweight operation that can't be undone. Or you might call it `Pause()`, if there is a way to `Resume()` it.

Finding More "Colorful" Words



Don't be afraid to use a thesaurus or ask a friend for better name suggestions. English is a rich language, and there are a lot of words to choose from.

Here are some examples of a word, as well as more "colorful" versions that might apply to your situation:

Word	Alternatives
send	deliver, dispatch, announce, distribute, route
find	search, extract, locate, recover
start	launch, create, begin, open
make	create, set up, build, generate, compose, add, new

Don't get carried away, though. In PHP, there is a function to `explode()` a string. That's a colorful name, and it paints a good picture of breaking something into pieces, but how is it any different

from `split()`? (The two functions are different, but it's hard to guess their differences based on the name.)

KEY IDEA

It's better to be clear and precise than to be cute.

Avoid Generic Names Like tmp and retval

Names like `tmp`, `retval`, and `foo` are usually cop-outs that mean "I can't think of a name." Instead of using an empty name like this, **pick a name that describes the entity's value or purpose**.

For example, here's a JavaScript function that uses `retval`:

```
var euclidean_norm = function (v) {
    var retval = 0.0;
    for (var i = 0; i < v.length; i += 1)
        retval += v[i] * v[i];
    return Math.sqrt(retval);
};
```

It's tempting to use `retval` when you can't think of a better name for your return value. But `retval` doesn't contain much information other than "I am a return value" (which is usually obvious anyway).

A better name would describe the purpose of the variable or the value it contains. In this case, the variable is accumulating the sum of the squares of `v`. So a better name is `sum_squares`. This would announce the purpose of the variable upfront and might help catch a bug.

For instance, imagine if the inside of the loop were accidentally:

```
retval += v[i];
```

This bug would be more obvious if the name were `sum_squares`:

```
sum_squares += v[i];  // Where's the "square" that we're summing? Bug!
```

ADVICE

The name `retval` doesn't pack much information. Instead, use a name that describes the variable's value.

There are, however, some cases where generic names do carry meaning. Let's take a look at when it makes sense to use them.

tmp

Consider the classic case of swapping two variables:

```
if (right < left) {
    tmp = right;
    right = left;
    left = tmp;
}
```

In cases like these, the name `tmp` is perfectly fine. The variable's sole purpose is temporary storage, with a lifetime of only a few lines. The name `tmp` conveys specific meaning to the reader—that this variable has no other duties. It's not being passed around to other functions or being reset or reused multiple times.

But here's a case where `tmp` is just used out of laziness:

```
String tmp = user.name();
tmp += " " + user.phone_number();
tmp += " " + user.email();
...
template.set("user_info", tmp);
```

Even though this variable has a short lifespan, being temporary storage isn't the most important thing about this variable. Instead, a name like `user_info` would be more descriptive.

In the following case, `tmp` should be in the name, but just as a *part* of it:

```
tmp_file = tempfile.NamedTemporaryFile()
...
SaveData(tmp_file, ...)
```

Notice that we named the variable `tmp_file` and not just `tmp`, because it is a file object. Imagine if we just called it `tmp`:

```
SaveData(tmp, ...)
```

Looking at just this one line of code, it isn't clear if `tmp` is a file, a filename, or maybe even the data being written.

ADVICE

The name `tmp` should be used only in cases when being short-lived and temporary is the most important fact about that variable.

Loop Iterators

Names like `i`, `j`, `iter`, and `it` are commonly used as indices and loop iterators. Even though these names are generic, they're understood to mean "I am an iterator." (In fact, if you used one of these names for some *other* purpose, it would be confusing—so don't do that!)

But sometimes there are better iterator names than `i`, `j`, and `k`. For instance, the following loops find which users belong to which clubs:

```
for (int i = 0; i < clubs.size(); i++)
    for (int j = 0; j < clubs[i].members.size(); j++)
        for (int k = 0; k < users.size(); k++)
            if (clubs[i].members[k] == users[j])
                cout << "user[" << j << "] is in club[" << i << "]" << endl;
```

In the `if` statement, `members[]` and `users[]` are using the wrong index. Bugs like these are hard to spot because that line of code seems fine in isolation:

```
if (clubs[i].members[k] == users[j])
```

In this case, using more precise names may have helped. Instead of naming the loop indexes (`i`,`j`,`k`), another choice would be (`club_i`, `members_i`, `users_i`) or, more succinctly (`ci`, `mi`, `ui`). This approach would help the bug stand out more:

```
if (clubs[ci].members[ui] == users[mi])  # Bug! First letters don't match up.
```

When used correctly, the first letter of the index would match the first letter of the array:

```
if (clubs[ci].members[mi] == users[ui])  # OK. First letters match.
```

The Verdict on Generic Names

As you've seen, there are some situations where generic names are useful.

> ADVICE
>
> **If you're going to use a generic name like `tmp`, `it`, or `retval`, have a good reason for doing so.**

A lot of the time, they're overused out of pure laziness. This is understandable—when nothing better comes to mind, it's easier to just use a meaningless name like `foo` and move on. But if you get in the habit of taking an extra few seconds to come up with a good name, you'll find your "naming muscle" builds quickly.

Prefer Concrete Names over Abstract Names

When naming a variable, function, or other element, describe it concretely rather than abstractly.

For example, suppose you have an internal method named `ServerCanStart()`, which tests whether the server can listen on a given TCP/IP port. The name `ServerCanStart()` is somewhat abstract, though. A more concrete name would be `CanListenOnPort()`. This name directly describes what the method will do.

The next two examples illustrate this concept in more depth.

Example: DISALLOW_EVIL_CONSTRUCTORS

Here's an example from the codebase at Google. In C++, if you don't define a copy constructor or assignment operator for your class, a default is provided. Although handy, these methods

can easily lead to memory leaks and other mishaps because they're executed "behind the scenes" in places you might not have realized.

As a result, Google has a convention to disallow these "evil" constructors, using a macro:

```
class ClassName {
  private:
    DISALLOW_EVIL_CONSTRUCTORS(ClassName);

  public:
    ...
};
```

This macro was defined as:

```
#define DISALLOW_EVIL_CONSTRUCTORS(ClassName) \
    ClassName(const ClassName&); \
    void operator=(const ClassName&);
```

By placing this macro in the `private:` section of a class, these two methods become private, so that they can't be used, even accidentally.

The name `DISALLOW_EVIL_CONSTRUCTORS` isn't very good, though. The use of the word "evil" conveys an overly strong stance on a debatable issue. More important, it isn't clear what that macro is disallowing. It disallows the `operator=()` method, and that isn't even a "constructor"!

The name was used for years but was eventually replaced with something less provocative and more concrete:

```
#define DISALLOW_COPY_AND_ASSIGN(ClassName) ...
```

Example: --run_locally

One of our programs had an optional command-line flag named `--run_locally`. This flag would cause the program to print extra debugging information but run more slowly. The flag was typically used when testing on a local machine, like a laptop. But when the program was running on a remote server, performance was important, so the flag wasn't used.

You can see how the name `--run_locally` came about, but it has some problems:

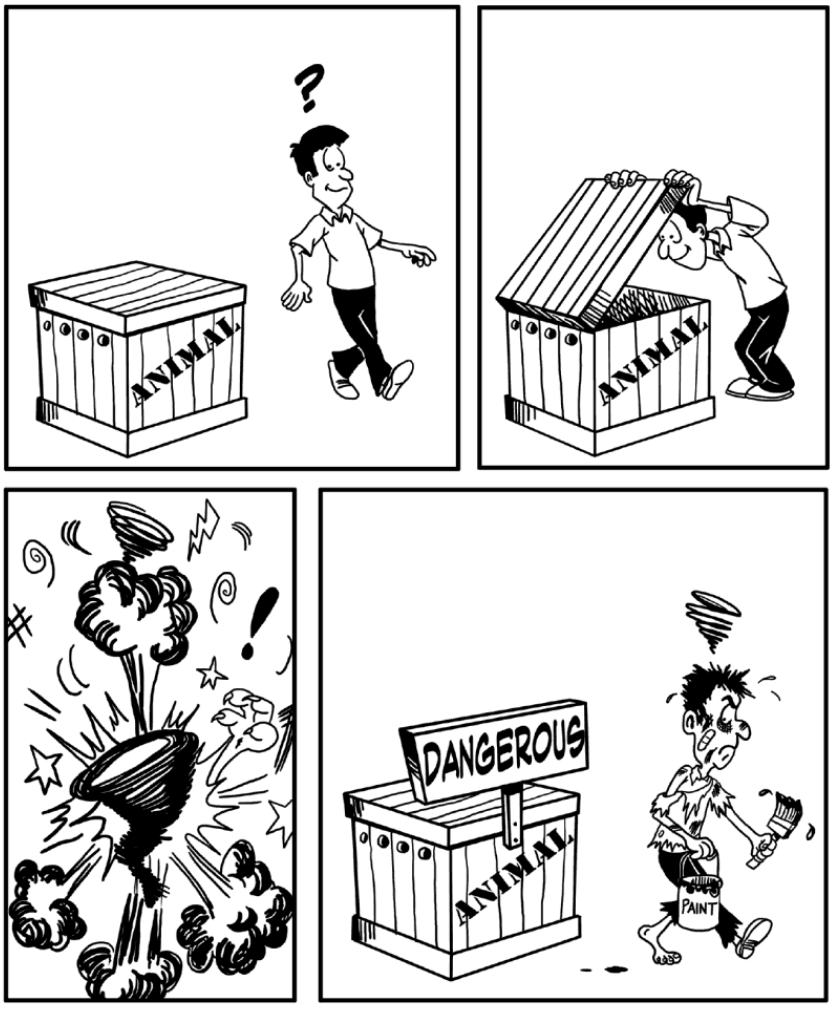
- A new member of the team didn't know what it did. He would use it when running locally (imagine that), but he didn't know why it was needed.
- Occasionally, we needed to print debugging information while the program ran remotely. Passing `--run_locally` to a program that is running remotely looks funny, and it's just confusing.
- Sometimes we would run a performance test locally and didn't want the logging slowing it down, so we wouldn't use `--run_locally`.

The problem is that `--run_locally` was named after the circumstance where it was typically used. Instead, a flag name like `--extra_logging` would be more direct and explicit.

But what if `--run_locally` needs to do more than just extra logging? For instance, suppose that it needs to set up and use a special local database. Now the name `--run_locally` seems more tempting because it can control both of these at once.

But using it for that purpose would be picking a name *because* it's vague and indirect, which is probably not a good idea. The better solution is to create a second flag named `--use_local_database`. Even though you have to use two flags now, these flags are much more explicit; they don't try to smash two orthogonal ideas into one, and they give you the option of using just one and not the other.

Attaching Extra Information to a Name



As we mentioned before, a variable's name is like a tiny comment. Even though there isn't much room, any extra information you squeeze into a name will be seen every time the variable is seen.

So if there's something very important about a variable that the reader must know, it's worth attaching an extra "word" to the name. For example, suppose you had a variable that contained a hexadecimal string:

```
string id;  // Example: "af84ef845cd8"
```

You might want to name it `hex_id` instead, if it's important for the reader to remember the ID's format.

Values with Units

If your variable is a measurement (such as an amount of time or a number of bytes), it's helpful to encode the units into the variable's name.

For example, here is some JavaScript code that measures the load time of a web page:

```
var start = (new Date()).getTime();  // top of the page
...
var elapsed = (new Date()).getTime() - start;  // bottom of the page
document.writeln("Load time was: " + elapsed + " seconds");
```

There is nothing obviously wrong with this code, but it doesn't work, because `getTime()` returns milliseconds, not seconds.

By appending `_ms` to our variables, we can make everything more explicit:

```
var start_ms = (new Date()).getTime();  // top of the page
...
var elapsed_ms = (new Date()).getTime() - start_ms;  // bottom of the page
document.writeln("Load time was: " + elapsed_ms / 1000 + " seconds");
```

Besides time, there are plenty of other units that come up in programming. Here is a table of unitless function parameters, and better versions that include the units:

Function parameter	Renaming parameter to encode units
`Start(int delay)`	`delay` → **`delay_secs`**
`CreateCache(int size)`	`size` → **`size_mb`**
`ThrottleDownload(float limit)`	`limit` → **`max_kbps`**
`Rotate(float angle)`	`angle` → **`degrees_cw`**

Encoding Other Important Attributes

This technique of attaching extra information to a name isn't limited to values with units. You should do it any time there's something dangerous or surprising about the variable.

For example, many security exploits come from not realizing that some data your program receives is not yet in a safe state. For this, you might want to use variable names like `untrustedUrl` or `unsafeMessageBody`. After calling functions that cleanse the unsafe input, the resulting variables might be `trustedUrl` or `safeMessageBody`.

The following table shows additional examples of when extra information should be encoded in the name:

Situation	Variable name	Better name
A `password` is in "plaintext" and should be encrypted before further processing	`password`	`plaintext_password`
A user-provided `comment` that needs escaping before being displayed	`comment`	`unescaped_comment`
Bytes of `html` have been converted to UTF-8	`html`	`html_utf8`
Incoming `data` has been "url encoded"	`data`	`data_urlenc`

You shouldn't use attributes like `unescaped_` or `_utf8` for *every* variable in your program. They're most important in places where a bug can easily sneak in if someone mistakes what the variable is, especially if the consequences are dire, as with a security bug. Essentially, if it's a critical thing to understand, put it in the name.

IS THIS HUNGARIAN NOTATION?

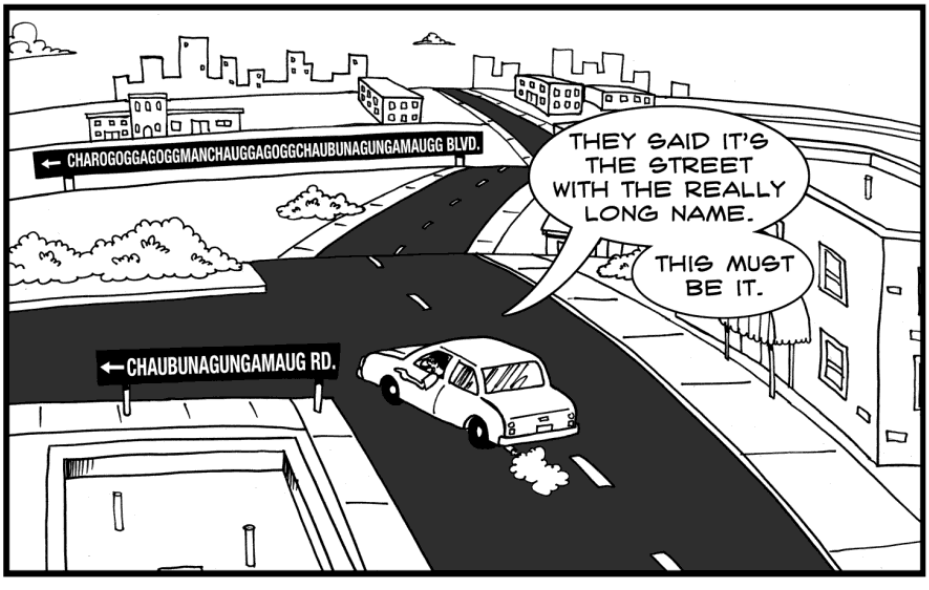
Hungarian notation is a system of naming used widely inside Microsoft. It encodes the "type" of every variable into the name's prefix. Here are some examples:

Name	Meaning
`pLast`	A pointer (p) to the last element in some data structure
`pszBuffer`	A pointer (p) to a zero-terminated (z) string (s) buffer
`cch`	A count (c) of characters (ch)
`mpcopx`	A map (m) from a pointer to a color (pco) to a pointer to an x-axis length (px)

It is indeed an example of "attaching attributes to names." But it's a more formal and strict system focused on encoding a specific set of attributes.

What we're advocating in this section is a broader, more informal system: identify any crucial attributes of a variable, and encode them legibly, if they're needed at all. You might call it "English Notation."

How Long Should a Name Be?



When picking a good name, there's an implicit constraint that the name shouldn't be too long. No one likes to work with identifiers like this:

```
newNavigationControllerWrappingViewControllerForDataSourceOfClass
```

The longer a name is, the harder it is to remember, and the more space it consumes on the screen, possibly causing extra lines to wrap.

On the other hand, programmers can take this advice too far, using only single-word (or single-letter) names. So how should you manage this trade-off? How do you decide between naming a variable `d`, `days`, or `days_since_last_update`?

This decision is a judgment call whose best answer depends on exactly how that variable is being used. But here are some guidelines to help you decide.

Shorter Names Are Okay for Shorter Scope

When you go on a short vacation, you typically pack less luggage than if you go on a long vacation. Similarly, identifiers that have a small "scope" (how many other lines of code can "see" this name) don't need to carry as much information. That is, you can get away with shorter names because all that information (what type the variable is, its initial value, how it's destroyed) is easy to see:

```
if (debug) {
    map<string,int> m;
    LookUpNamesNumbers(&m);
    Print(m);
}
```

Even though `m` doesn't pack any information, it's not a problem, because the reader already has all the information she needs to understand this code.

However, suppose `m` were a class member or a global variable, and you saw this snippet of code:

```
LookUpNamesNumbers(&m);
Print(m);
```

This code is much less readable, as it's unclear what the type or purpose of `m` is.

So if an identifier has a large scope, the name needs to carry enough information to make it clear.

Typing Long Names—Not a Problem Anymore

There are many good reasons to avoid long names, but "they're harder to type" is no longer one of them. Every programming text editor we've seen has "word completion" built in. Surprisingly, most programmers aren't aware of this feature. If you haven't tried this feature on your editor yet, please put this book down right now and try it:

1. Type the first few characters of the name.
2. Trigger the word-completion command (see below).
3. If the completed word is not correct, keep triggering the command until the correct name appears.

It's surprisingly accurate. It works on any type of file, in any language. And it works for any token, even if you're typing a comment.

Editor	Command
Vi	Ctrl-p
Emacs	Meta-/ (hit ESC, then /)
Eclipse	Alt-/
IntelliJ IDEA	Alt-/
TextMate	ESC

Acronyms and Abbreviations

Programmers sometimes resort to acronyms and abbreviations to keep their names small—for example, naming a class `BEManager` instead of `BackEndManager`. Is this shrinkage worth the potential confusion?

In our experience, project-specific abbreviations are usually a bad idea. They appear cryptic and intimidating to those new to the project. Given enough time, they even start to appear cryptic and intimidating to the authors.

So our rule of thumb is: **would a new teammate understand what the name means?** If so, then it's probably okay.

For example, it's fairly common for programmers to use `eval` instead of `evaluation`, `doc` instead of `document`, `str` instead of `string`. So a new teammate seeing `FormatStr()` will probably understand what that means. However, he or she probably won't understand what a `BEManager` is.

Throwing Out Unneeded Words

Sometimes words inside a name can be removed without losing any information at all. For instance, instead of `ConvertToString()`, the name `ToString()` is smaller and doesn't lose any real information. Similarly, instead of `DoServeLoop()`, the name `ServeLoop()` is just as clear.

Use Name Formatting to Convey Meaning

The way you use underscores, dashes, and capitalization can also pack more information in a name. For example, here is some C++ code that follows the formatting conventions used for Google open source projects:

```
static const int kMaxOpenFiles = 100;

class LogReader {
  public:
    void OpenFile(string local_file);

  private:
    int offset_;
    DISALLOW_COPY_AND_ASSIGN(LogReader);
};
```

Having different formats for different entities is like a form of syntax highlighting—it helps you read the code more easily.

Most of the formatting in this example is pretty common—using `CamelCase` for class names, and using `lower_separated` for variable names. But some of the other conventions may have surprised you.

For instance, constant values are of the form `kConstantName` instead of `CONSTANT_NAME`. This style has the benefit of being easily distinguished from `#define` macros, which are `MACRO_NAME` by convention.

Class member variables are like normal variables, but must end with an underscore, like `offset_`. At first, this convention may seem strange, but being able to instantly distinguish

members from other variables is very handy. For instance, if you're glancing through the code of a large method, and see the line:

```
stats.clear();
```

you might ordinarily wonder, *Does* `stats` *belong to this class? Is this code changing the internal state of the class?* If the `member_` convention is used, you can quickly conclude, *No,* `stats` *must be a local variable. Otherwise it would be named* `stats_`.

Other Formatting Conventions

Depending on the context of your project or language, there may be other formatting conventions you can use to make names contain more information.

For instance, in *JavaScript: The Good Parts* (Douglas Crockford, O'Reilly, 2008), the author suggests that "constructors" (functions intended to be called with `new`) should be capitalized and that ordinary functions should start with a lowercase letter:

```
var x = new DatePicker();  // DatePicker() is a "constructor" function
var y = pageHeight();      // pageHeight() is an ordinary function
```

Here's another JavaScript example: when calling the jQuery library function (whose name is the single character `$`), a useful convention is to prefix jQuery results with `$` as well:

```
var $all_images = $("img");  // $all_images is a jQuery object
var height = 250;            // height is not
```

Throughout the code, it will be clear that `$all_images` is a jQuery result object.

Here's a final example, this time about HTML/CSS: when giving an HTML tag an `id` or `class` attribute, both underscores and dashes are valid characters to use in the value. One possible convention is to use underscores to separate words in IDs and dashes to separate words in classes:

```
<div id="middle_column" class="main-content"> ...
```

Whether you decide to use conventions like these is up to you and your team. But whichever system you use, be consistent across your project.

Summary

The single theme for this chapter is: **pack information into your names**. By this, we mean that the reader can extract a lot of information just from reading the name.

Here are some specific tips we covered:

- **Use specific words**—for example, instead of `Get`, words like `Fetch` or `Download` might be better, depending on the context.
- **Avoid generic names** like `tmp` and `retval`, unless there's a specific reason to use them.

Symbols

A

B

C

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

D

E

M

N

O

P

Q

W

X