Extracted from:

The Way of the Web Tester

A Beginner's Guide to Automating Tests

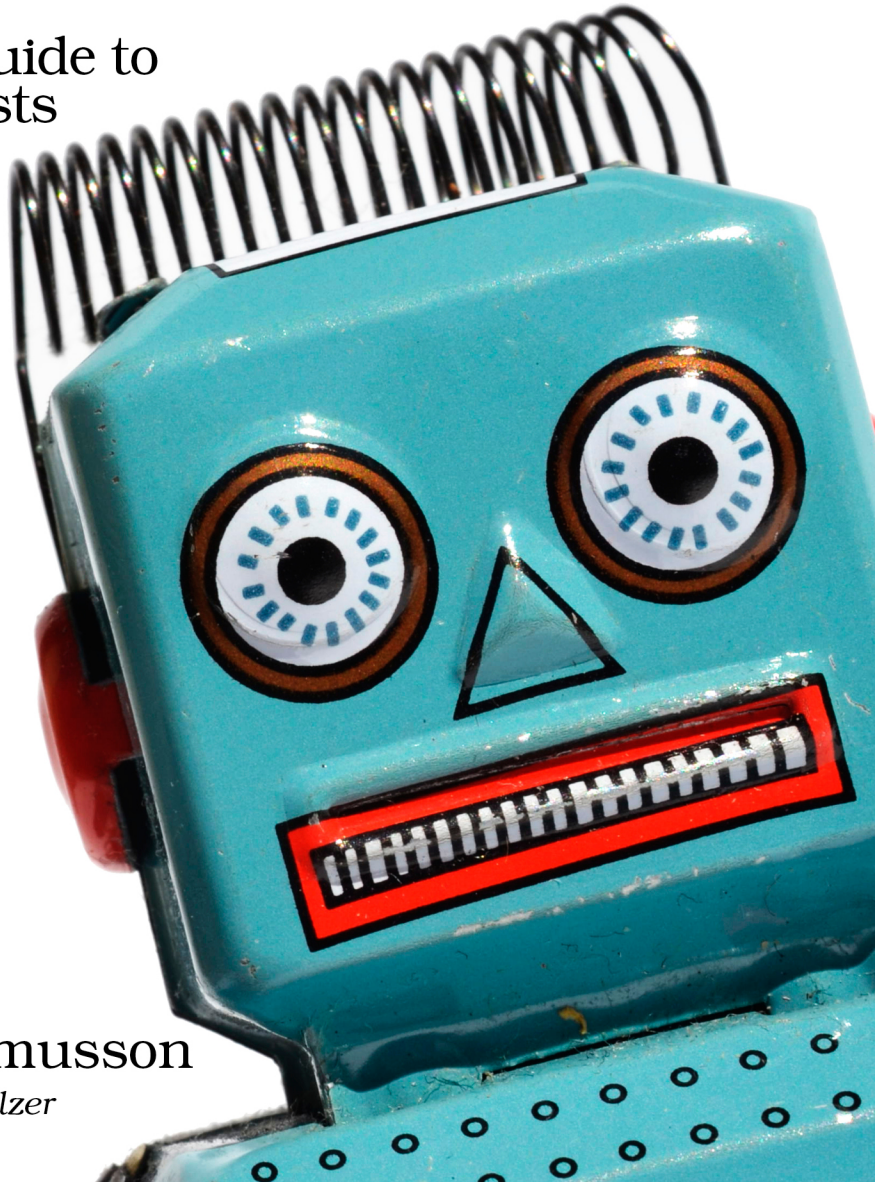
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

The Way of the Web Tester

A Beginner's Guide to Automating Tests



Jonathan Rasmusson

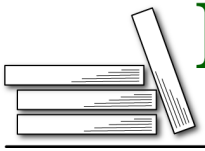
edited by Susannah Pfalzer

The Way of the Web Tester

A Beginner's Guide to Automating Tests

Jonathan Rasmusson

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The yellow adhesive note graphic in Chapter 11 is designed by Layerace from Freepik.com.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (index)
Nicole Abramowitz (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-183-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2016

The Importance of Style

It was a dark and stormy night.
The Wumpus is nearby!

Type 's' to shoot your arrow
or 'r' to run away.

```
def process(command)
  if command = "s"
    print "You've killed the Wumpus!"
  else if command = "r"
    print "Sir Robin bravely runs away."
end
```

Just like in writing, style makes a big difference in the clarity of your programming. Style is important because while computers run the programs, it's people like you and me who read and maintain them. So the clearer we can make our code, the easier it's going to be to modify, change, and support (not to mention contain fewer bugs).

To help you with your style, we are going to look at three things programmers constantly do to increase the quality of their code. Specifically, we are going to look at naming, spacing, and removing duplication.

Naming

Names really matter in programming. When we get the name of something right, understanding the program becomes a breeze. Get the name of something wrong, however, and understanding even our own code can be a bit of a nightmare.

Take this little method. You gotta feel sorry for whoever's responsible for maintaining this. It's not at all clear what the author was thinking when they wrote it.


```
if (val(b))
  redirect :wlcm_pg
else
  redirect :lgn_pg
end
```

Yet when we change a few words and rename a couple variables—bam! The intent suddenly becomes clear.


```
if valid(password)
  redirect :welcome_page
else
  redirect :login_page
end
```

It's hard to give super concrete advice on naming, because so much of what makes a good name is contextual. The perfect word on one project can be confusing and overloaded on another.

But here are some general guidelines to think about when choosing names.

Good names...	Good	Bad
Are easy to understand	salary	s
Make intent clear	brand_color	red
Explain	seconds_per_minute	60
Aren't too long	nasa	nasa_aeronautics_space_administration
Are descriptive	workDays	days
Avoid double negatives	isValid	isNotValid

It comes down to treating code like an author would treat the words and paragraphs in a good short story. You want to be clear with what you are saying, you want the program to be easy to read, and you don't want to make the reader work too hard to see and understand what it is you are doing.

And another element of style that can help with that is spacing.

Spacing

Spacing? That's right. Believe it or not, how you space and indent your code makes a big difference in its readability. Just like reading paragraphs in a book, understanding code gets hard if things aren't spaced and indented properly.

And it's not just for readability that spacing matters. Some authors of languages like Python and early versions of Fortran thought spacing was so important that your program wouldn't even run if you didn't space things correctly!

Naming Conventions: Go with the Flow

Every computer language has a convention for how it likes people to name things. Java, for example, uses a convention called CamelCase, where you alternate the capitalization of the letters when combining words.

```
int highScore = 1000;
String firstName = "Steve";
float myBankAccountAfterComingBackFromVacation = 0.0;
```

Ruby uses CamelCase for defining classes, but when it comes to naming variables and methods, the convention then is to separate them with underscores.

```
int max_number_of_songs_in_playlist = 1000
float currenct_exchange_rate = 2.4;
int average_age_of_hockey_player_in_nhl = 27
```

Whatever language you end up writing your automated tests in, it's probably a good idea if you stick with the coding convention for that language and go with the flow. It will make your tests easier to read and will be less confusing to others following in your footsteps.


```
def display_high_score
    if (new_score > old_score)
 enter_initials
 else
redirect       :game_over
      end
        end
```


```
def display_high_score
  if (new_score > old_score)
→   enter_initials
  else
→   redirect :game_over
  end
end
```

The other thing that helps with making your programs easier to understand is grouping related things together.

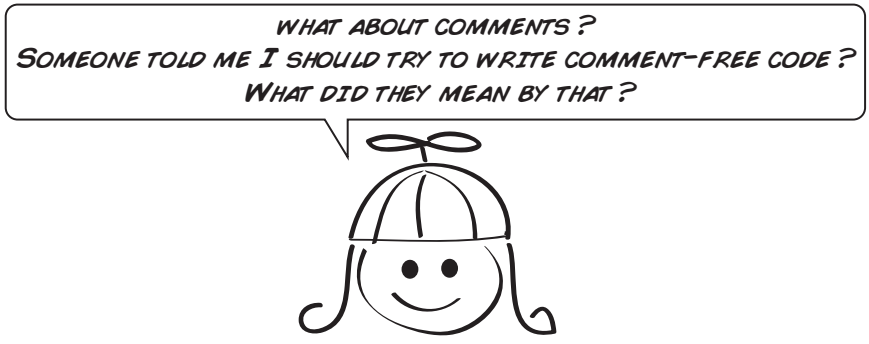

```
def some_hard_to_read_test
  get new_password_reset_path
  assert_template 'password_resets/new'
  post password_resets_path, password_reset: { email: "" }
  assert_not flash.empty?
  assert_template 'password_resets/new'
  post password_resets_path, password_reset: { email: @user.email }
  assert_not flash.empty?
  assert_redirected_to root_url
end
```

When you group related things together and add a little something we call *whitespace* (blank lines between paragraphs of code), a big jumble of code can suddenly become a lot clearer. Now when you read the code, you don't have to think as much. You can scan it at a glance and see instantly what's going on.


```
def a_nicely_spaced_test

  # Go to page
  get new_password_reset_path
  assert_template 'password_resets/new'

  # Try invalid email
  post password_resets_path, password_reset: { email: "" }
  assert_not flash.empty?
  assert_template 'password_resets/new'

  # Try valid email
  post password_resets_path, password_reset: { email: @user.email }
  assert_not flash.empty?
  assert_redirected_to root_url

end
```

whitespace

comments are OK too!



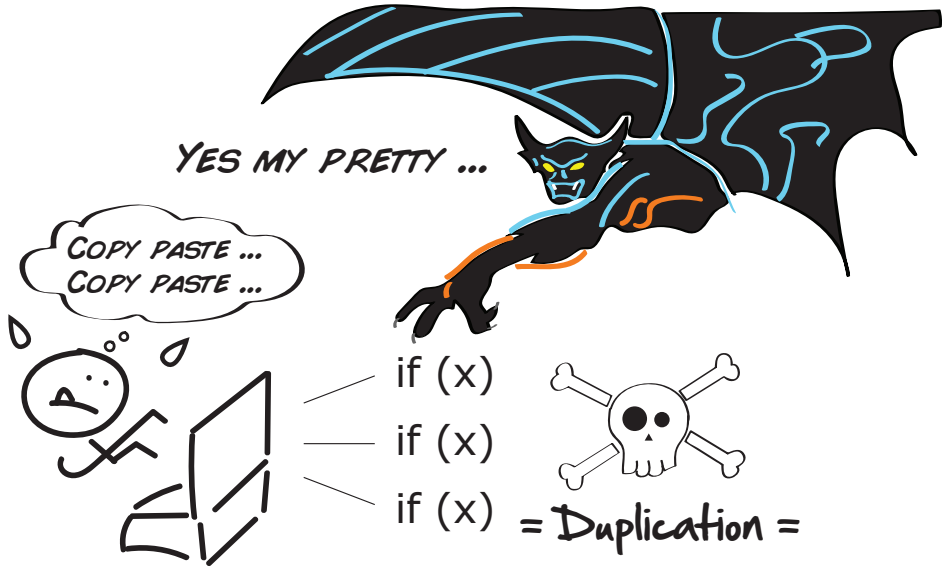
Comment-free code is a term developers use to challenge and remind themselves that the code they write should be so clear and easy to understand that no comments are required. It should all just make sense.

While comment-free code is a nice goal and something we should all definitely strive for, there is nothing wrong with dropping the occasional comment in your code to clarify or explain your thinking.

What we want to avoid with comments is redundancy. If the code already clearly explains what's going on, no additional comments should be required. But if there is some wrinkle, a hidden gotcha, or a non-obvious reason for why something might happen in the code, a well-placed comment is perfectly fine and desired.

Next let's look at the root of all evil in software—duplication.

Dealing with Duplication



Copying and pasting code is one of these double-edged swords in software. On the one hand, it's great for quickly getting things up and running. But on the other hand, it makes our code more fragile and harder to change.

To see what I mean, take a look at the following test code. A common pattern in testing is to get the first test working, and then copy and paste the same code for the other test cases afterward, resulting in code that looks like this:

```
test 'can access welcome page' do
  @user = users(:user1)
  get login_path
  post login_path, session: { email: 'user@test.com', password: 'password' }
  follow_redirect!
  assert_select 'h1', 'Welcome'
end

test 'can access company financials' do
  @user = users(:user1)
  get login_path
  post login_path, session: { email: 'user@test.com', password: 'password' }
  follow_redirect!
  get financials_path
  assert_select 'h1', 'Financials'
end

test 'can access plans for world domination' do
  @user = users(:user1)
  get login_path
  post login_path, session: { email: 'user@test.com', password: 'password' }
  follow_redirect!
  get world_domination_path
  assert_select 'h1', 'Step1: Take Saskatchewan'
end
```

The advantage of copying and pasting here is that it is simple and quick. We get immediate feedback with regards to whether our tests are working. That is good.

The downside of stopping here, however, is that if we ever decide to change anything about how these tests work (like logging in with a new password), we now need to do it in three places instead of one.

One way to clean this code up a bit would be to pull all the common code shared between the methods into one `setup` method, and then call that `setup` method at the beginning of each test. That code would look something like this:

```
def setup
  @user = users(:user1)
  get login_path
  post login_path, session: { email: 'user@test.com', password: 'password' }
  follow_redirect!
end

test 'can access welcome page' do
  assert_select 'h1', 'Welcome'
end

test 'can access company financials' do
  assert_select 'h1', 'Financials'
end
```

```
test 'can access plans for world domination' do
  get world_domination_path
  assert_select 'h1', 'Step1: Take Saskatchewan'
end
```

This is much cleaner, much simpler, and much easier to read. The reason you don't see `setup` being called directly from each method is that `setup` is a special test method that automated testing frameworks support for doing this sort of thing automatically for us. So in this case, we don't need to call it ourselves.

What we just did here (this small but important act of removing duplication), developers call *refactoring*. In layman's terms, refactoring is nothing more than going back and cleaning your code up. It can include things like renaming variables and picking better method names. But it usually boils down to removing duplication and making the code easier to read.

We want to do these kinds of things when we are writing our tests. Any duplication we see, we are going to want to pull out and get rid of. Doing so will not only make our tests easier to read, it will also make them way easier to change and understand.

Remove Duplication by Continuously Refactoring

Refactoring is the act of improving the design of your code without changing its underlying functionality. That may sound a little weird, but it is an important part of the programming processes.

You see, when we write code and tests, we are in two states of mind. One is to get the test or piece of code working. But the other, often missed step, is to go back and make sure that everything is as clean as possible, and that the code is as simple and easy to read as possible.

That's what refactoring is. It's that critical step that prevents code from decaying over time and collapsing under its own weight, and instead continuously improves it so it remains a joy to work with.

To learn more about this technique and other ways to improve your software, check out Martin Fowler's book on the subject, *Refactoring: Improving the Design of Existing Code [FBBO99]*.

OK. Those are some basic techniques for writing good code. Let's try them out now and see what they look like in action.