Mastering the Vim Language

Chris Toomey

@thoughtbot

@christoomey

https://ctoomey.com

I love Vim because I've yet to hit the ceiling

Typing is not the bottleneck

-- Michael Hill (GeePawHill)[1]

[1] http://anarchycreek.com/2009/05/26/how-tdd-and-pairing-increase-production/

Vim's killer feature is the language it provides for making changes

Syntax of the Language

Verb + Noun

`d` for delete

`w` for word,

combine to be "delete word"

Commands are Repeatable & Undoable

Verbs in Vim

The operation you want to take on the text

- `d` => **Delete**
- `c` => **Change** (delete and enter insert mode)
- `>` => **Indent**
- `v` => **Visually select**
- `y` => **Yank** (copy)

Nouns in Vim -- Motions

- `w` => **word** (forward by a "word")
- `b` => **back** (back by a "word")
- `2j` => down 2 lines

Nouns in Vim -- Text Objects

- `iw` => "inner word" (works from anywhere in a word)
- `it` => "inner tag" (the contents of an HTML tag)
- `i"` => "inner quotes"
- `ip` => "inner paragraph"
- `as` => "a sentence"

Nouns in Vim -- Parameterized Text Objects

- `f`, `F` => "find" the next character
- `t`, `T` => "find" the next character
- `/` => Search (up to the next match)

Combinatorics of Commands

```
  5 operators * 10 motions
+ 5 operators * 10 text objects
+ 5 operators * 35 characters * 4 (for `f`, `F`, `t`, `T`)
+ 5 operators * ~100 (for `/`)
```

Combinatorics of Commands

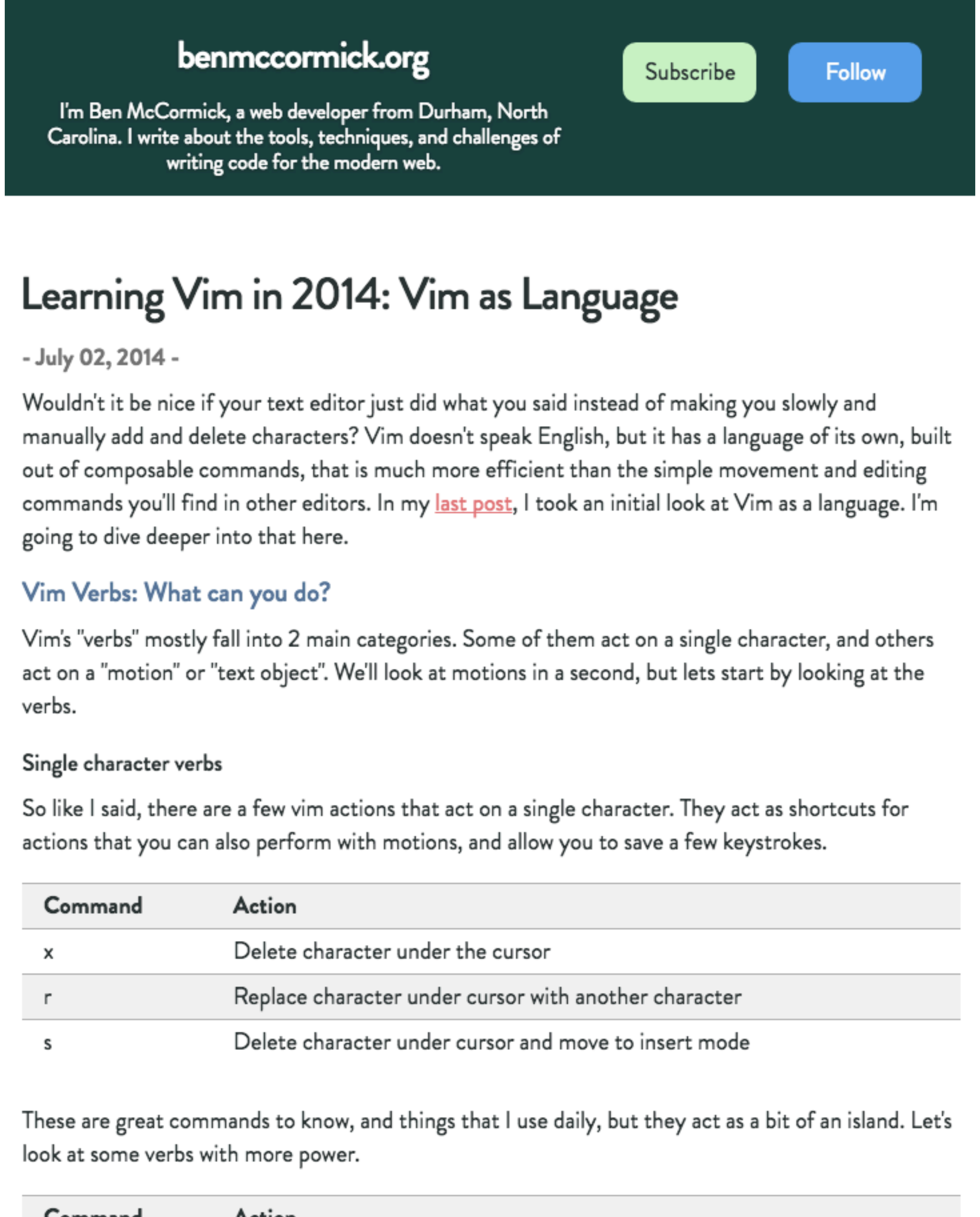
```
  5 commands * 10 motions
+ 5 commands * 10 text objects
+ 5 commands * 70 characters * 4 (for `f`, `F`, `t`, `T`)
+ 5 commands * ~100 (for `/`)
```

2000

Distinct commands based on memorizing ~30 key mappings (that are very memorable)

Learning Vim as a Language

Ben McCormick



benmccormick.org

Subscribe

Follow

I'm Ben McCormick, a web developer from Durham, North Carolina. I write about the tools, techniques, and challenges of writing code for the modern web.

Learning Vim in 2014: Vim as Language

- July 02, 2014 -

Wouldn't it be nice if your text editor just did what you said instead of making you slowly and manually add and delete characters? Vim doesn't speak English, but it has a language of its own, built out of composable commands, that is much more efficient than the simple movement and editing commands you'll find in other editors. In my last post, I took an initial look at Vim as a language. I'm going to dive deeper into that here.

Vim Verbs: What can you do?

Vim's "verbs" mostly fall into 2 main categories. Some of them act on a single character, and others act on a "motion" or "text object". We'll look at motions in a second, but lets start by looking at the verbs.

Single character verbs

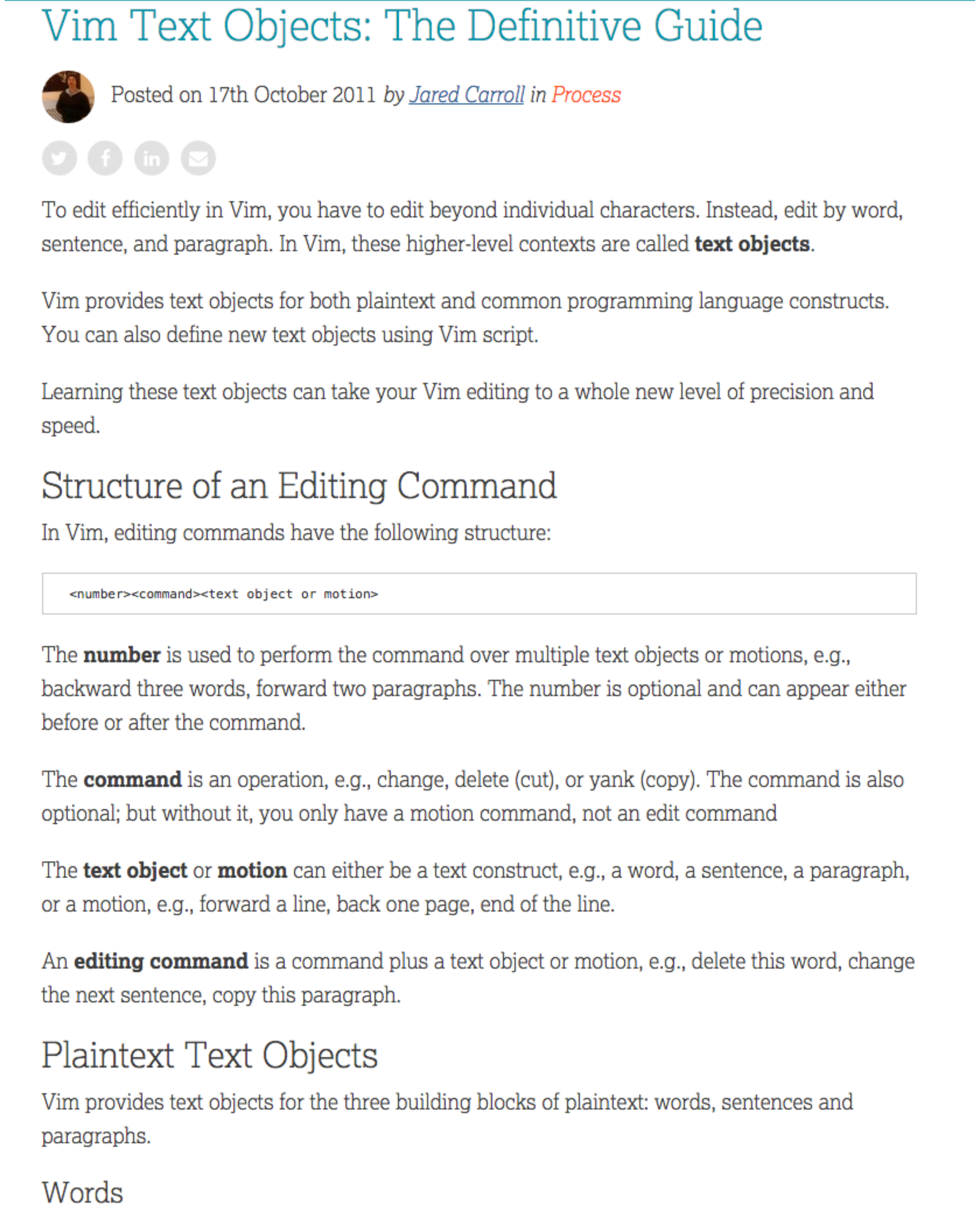
So like I said, there are a few vim actions that act on a single character. They act as shortcuts for actions that you can also perform with motions, and allow you to save a few keystrokes.

Command	Action
x	Delete character under the cursor
r	Replace character under cursor with another character
s	Delete character under cursor and move to insert mode

These are great commands to know, and things that I use daily, but they act as a bit of an island. Let's look at some verbs with more power.

Vim Text Objects: The Definitive Guide

Carbon Five blog



Vim Text Objects: The Definitive Guide

Posted on 17th October 2011 *by* *Jared Carroll* *in* *Process*

To edit efficiently in Vim, you have to edit beyond individual characters. Instead, edit by word, sentence, and paragraph. In Vim, these higher-level contexts are called **text objects**.

Vim provides text objects for both plaintext and common programming language constructs. You can also define new text objects using Vim script.

Learning these text objects can take your Vim editing to a whole new level of precision and speed.

Structure of an Editing Command

In Vim, editing commands have the following structure:

```
<number><command><text object or motion>
```

The **number** is used to perform the command over multiple text objects or motions, e.g., backward three words, forward two paragraphs. The number is optional and can appear either before or after the command.

The **command** is an operation, e.g., change, delete (cut), or yank (copy). The command is also optional; but without it, you only have a motion command, not an edit command

The **text object** or **motion** can either be a text construct, e.g., a word, a sentence, a paragraph, or a motion, e.g., forward a line, back one page, end of the line.

An **editing command** is a command plus a text object or motion, e.g., delete this word, change the next sentence, copy this paragraph.

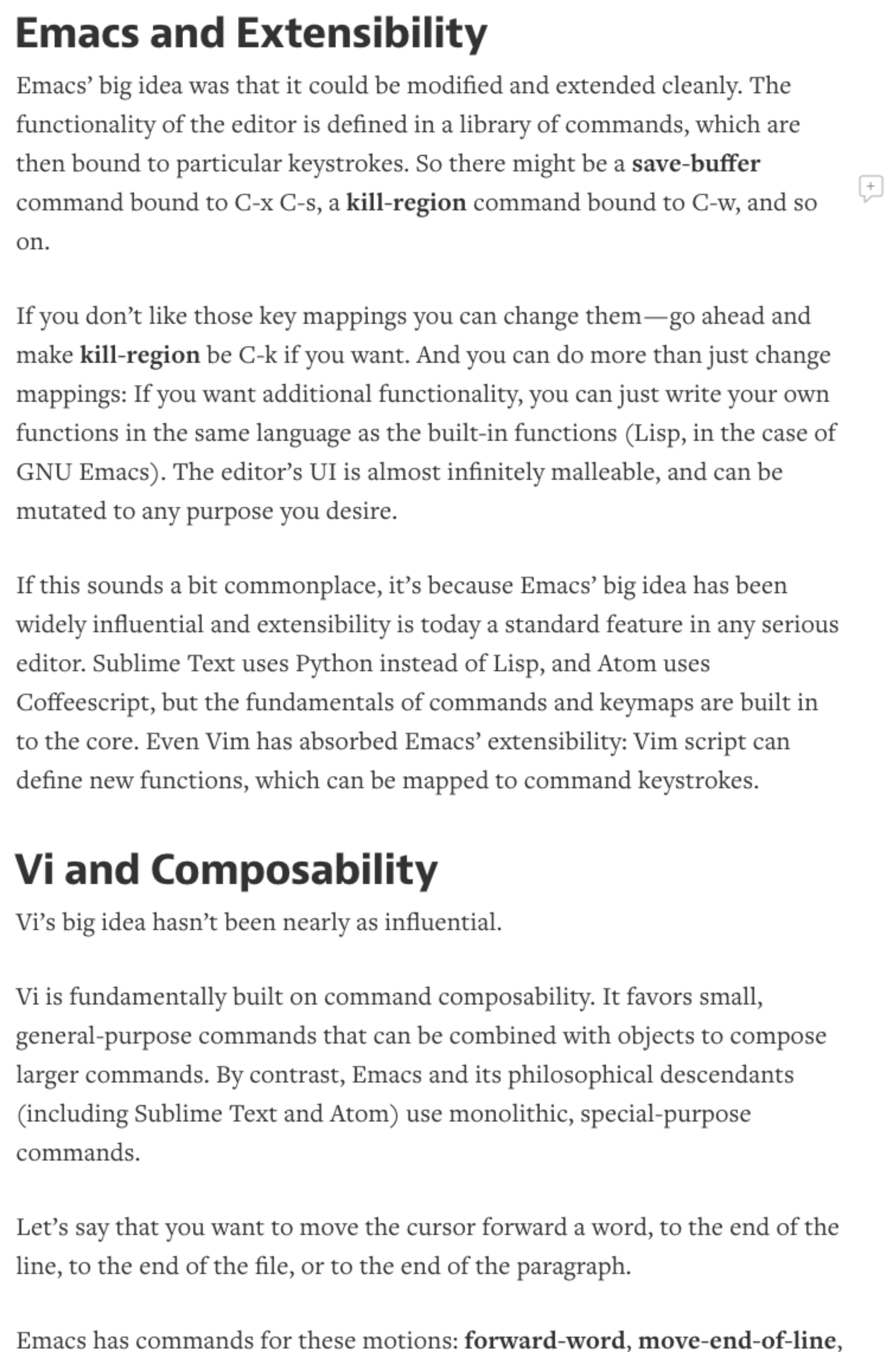
Plaintext Text Objects

Vim provides text objects for the three building blocks of plaintext: words, sentences and paragraphs.

Words

Why Atom Can't Replace Vim

Mike Kozlowski



Emacs and Extensibility

Emacs' big idea was that it could be modified and extended cleanly. The functionality of the editor is defined in a library of commands, which are then bound to particular keystrokes. So there might be a **save-buffer** command bound to C-x C-s, a **kill-region** command bound to C-w, and so on.

If you don't like those key mappings you can change them—go ahead and make **kill-region** be C-k if you want. And you can do more than just change mappings: If you want additional functionality, you can just write your own functions in the same language as the built-in functions (Lisp, in the case of GNU Emacs). The editor's UI is almost infinitely malleable, and can be mutated to any purpose you desire.

If this sounds a bit commonplace, it's because Emacs' big idea has been widely influential and extensibility is today a standard feature in any serious editor. Sublime Text uses Python instead of Lisp, and Atom uses Coffeescript, but the fundamentals of commands and keymaps are built in to the core. Even Vim has absorbed Emacs' extensibility: Vim script can define new functions, which can be mapped to command keystrokes.

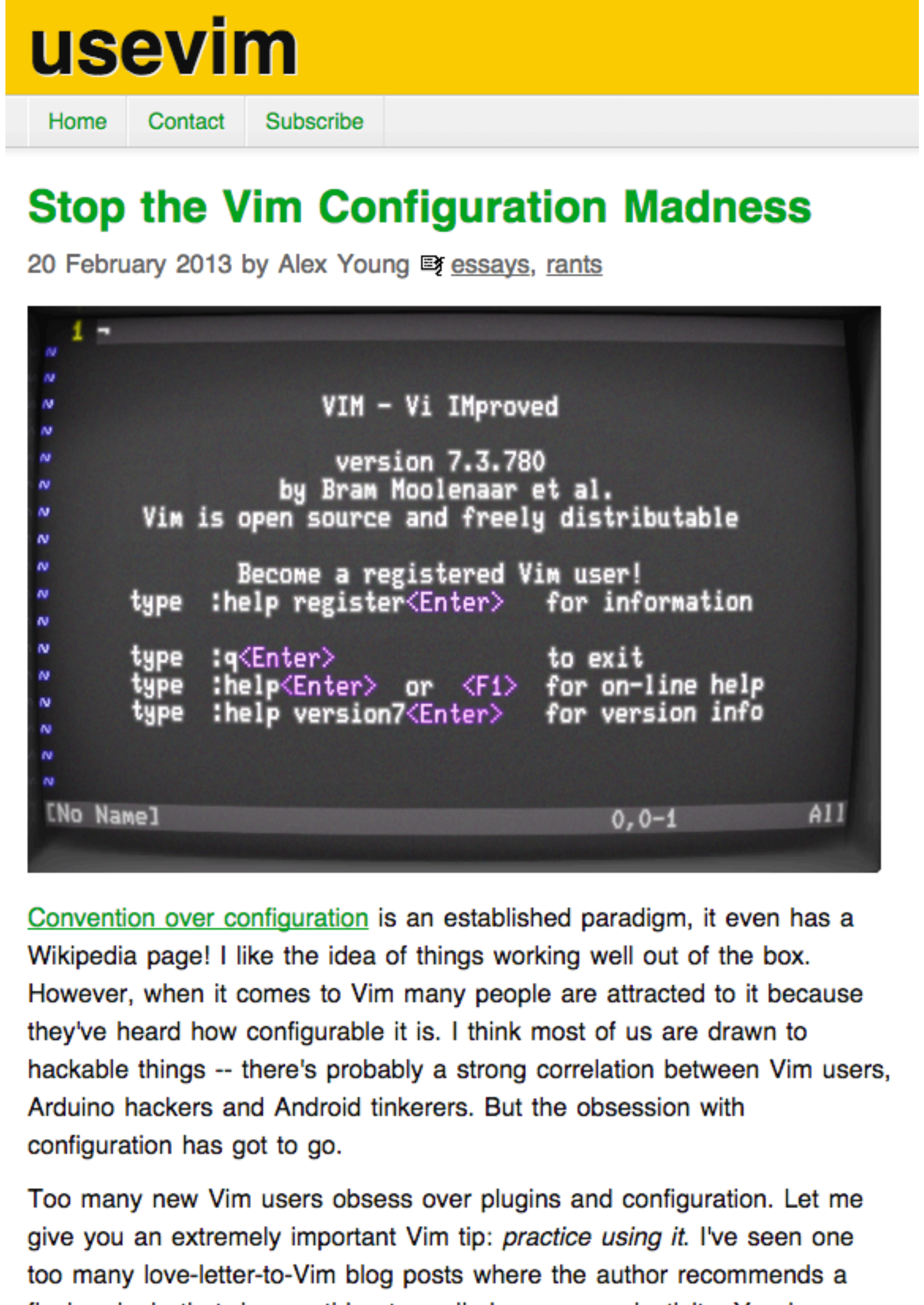
Vi and Composability

Vi's big idea hasn't been nearly as influential.

Vi is fundamentally built on command composability. It favors small, general-purpose commands that can be combined with objects to compose larger commands. By contrast, Emacs and its philosophical descendants (including Sublime Text and Atom) use monolithic, special-purpose commands.

Let's say that you want to move the cursor forward a word, to the end of the line, to the end of the file, or to the end of the paragraph.

Emacs has commands for these motions: **forward-word**, **move-end-of-line**,



usevim

Home Contact Subscribe

Stop the Vim Configuration Madness

20 February 2013 by Alex Young essays, rants

```
  1 ~
~
~                          VIM - Vi IMproved
~
~                           version 7.3.780
~                      by Bram Moolenaar et al.
~             Vim is open source and freely distributable
~
~                     Become a registered Vim user!
~            type  :help register<Enter>   for information
~
~            type  :q<Enter>               to exit
~            type  :help<Enter>  or  <F1>  for on-line help
~            type  :help version7<Enter>   for version info
~
~
[No Name]                                       0,0-1          All
```

Convention over configuration is an established paradigm, it even has a Wikipedia page! I like the idea of things working well out of the box. However, when it comes to Vim many people are attracted to it because they've heard how configurable it is. I think most of us are drawn to hackable things -- there's probably a strong correlation between Vim users, Arduino hackers and Android tinkerers. But the obsession with configuration has got to go.

Too many new Vim users obsess over plugins and configuration. Let me give you an extremely important Vim tip: *practice using it*. I've seen one too many love-letter-to-Vim blog posts where the author recommends a

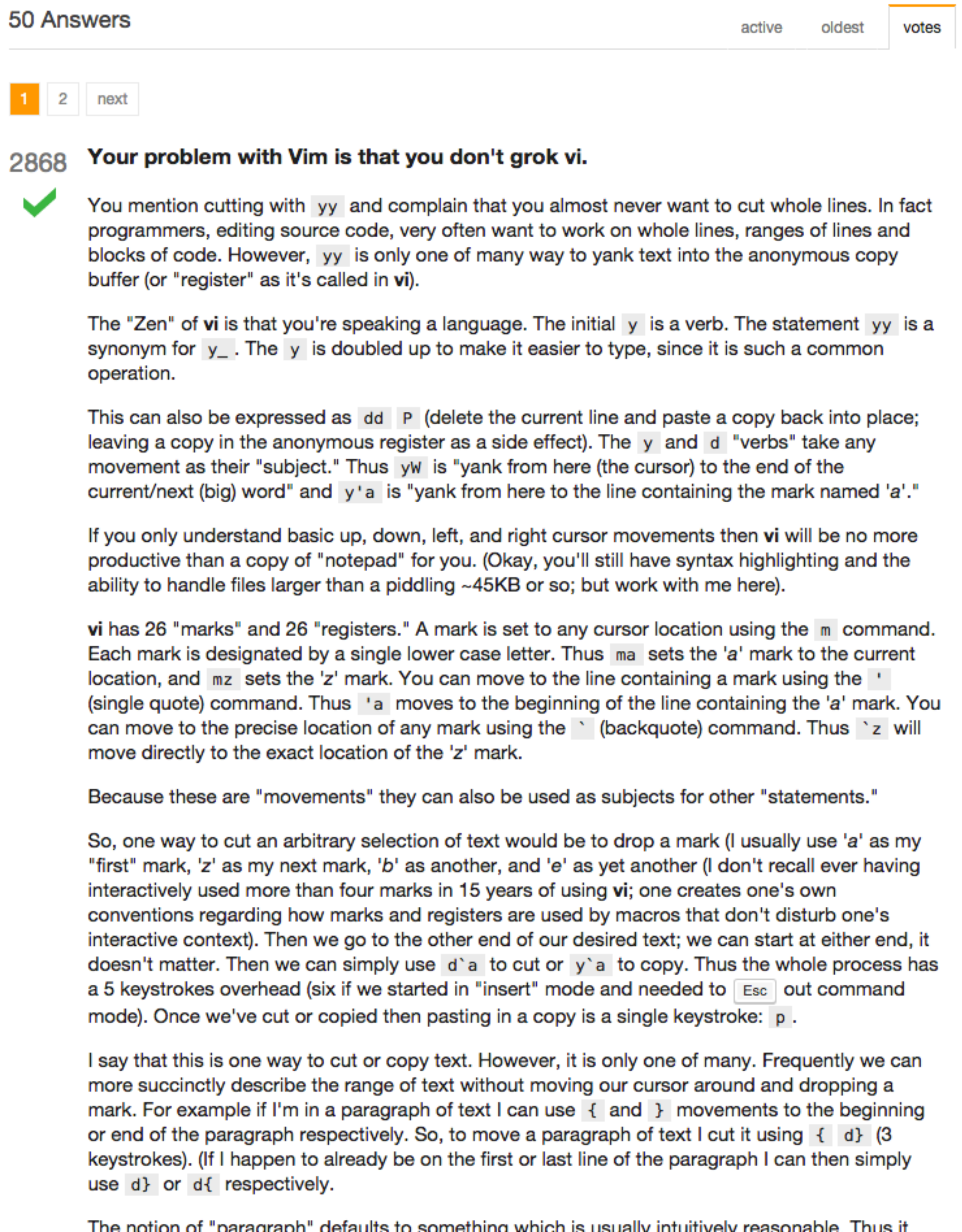
Stop the Vim Configfiguration Madness

Contains this great, succinct quote:

> You know what improves productivity? Mastering motions and operators.

Your problem with Vim is that you don't grok vi.

The classic Stackoverflow post that sets the standard on talking about the Vim language.



50 Answers

active oldest votes

1 2 next

2868

Your problem with Vim is that you don't grok vi.

You mention cutting with `yy` and complain that you almost never want to cut whole lines. In fact programmers, editing source code, very often want to work on whole lines, ranges of lines and blocks of code. However, `yy` is only one of many way to yank text into the anonymous copy buffer (or "register" as it's called in **vi**).

The "Zen" of **vi** is that you're speaking a language. The initial `y` is a verb. The statement `yy` is a synonym for `y_`. The `y` is doubled up to make it easier to type, since it is such a common operation.

This can also be expressed as `dd` `P` (delete the current line and paste a copy back into place; leaving a copy in the anonymous register as a side effect). The `y` and `d` "verbs" take any movement as their "subject." Thus `yW` is "yank from here (the cursor) to the end of the current/next (big) word" and `y'a` is "yank from here to the line containing the mark named '*a*'."

If you only understand basic up, down, left, and right cursor movements then **vi** will be no more productive than a copy of "notepad" for you. (Okay, you'll still have syntax highlighting and the ability to handle files larger than a piddling ~45KB or so; but work with me here).

vi has 26 "marks" and 26 "registers." A mark is set to any cursor location using the `m` command. Each mark is designated by a single lower case letter. Thus `ma` sets the '*a*' mark to the current location, and `mz` sets the '*z*' mark. You can move to the line containing a mark using the `'` (single quote) command. Thus `'a` moves to the beginning of the line containing the '*a*' mark. You can move to the precise location of any mark using the `` ` `` (backquote) command. Thus `` `z `` will move directly to the exact location of the '*z*' mark.

Because these are "movements" they can also be used as subjects for other "statements."

So, one way to cut an arbitrary selection of text would be to drop a mark (I usually use '*a*' as my "first" mark, '*z*' as my next mark, '*b*' as another, and '*e*' as yet another (I don't recall ever having interactively used more than four marks in 15 years of using **vi**; one creates one's own conventions regarding how marks and registers are used by macros that don't disturb one's interactive context). Then we go to the other end of our desired text; we can start at either end, it doesn't matter. Then we can simply use `` d`a `` to cut or `` y`a `` to copy. Thus the whole process has a 5 keystrokes overhead (six if we started in "insert" mode and needed to `Esc` out command mode). Once we've cut or copied then pasting in a copy is a single keystroke: `p`.

I say that this is one way to cut or copy text. However, it is only one of many. Frequently we can more succinctly describe the range of text without moving our cursor around and dropping a mark. For example if I'm in a paragraph of text I can use `{` and `}` movements to the beginning or end of the paragraph respectively. So, to move a paragraph of text I cut it using `{` `d}` (3 keystrokes). (If I happen to already be on the first or last line of the paragraph I can then simply use `d}` or `d{` respectively.

The notion of "paragraph" defaults to something which is usually intuitively reasonable. Thus it

Tips for Mastering the Language

The "dot" command

- Use the more general text object (`iw` rather than `w` even if at beginning of word)
- Prefer text objects to motions when possible
- Repeat.vim for plugin repeating

Relative Number

Visual Mode Is a Smell

Don't use two sentences where one will due

Breaks repeatability

Custom Operators

Surround
Commentary
ReplaceWithRegister
Titlecase
Sort-motion
System-copy

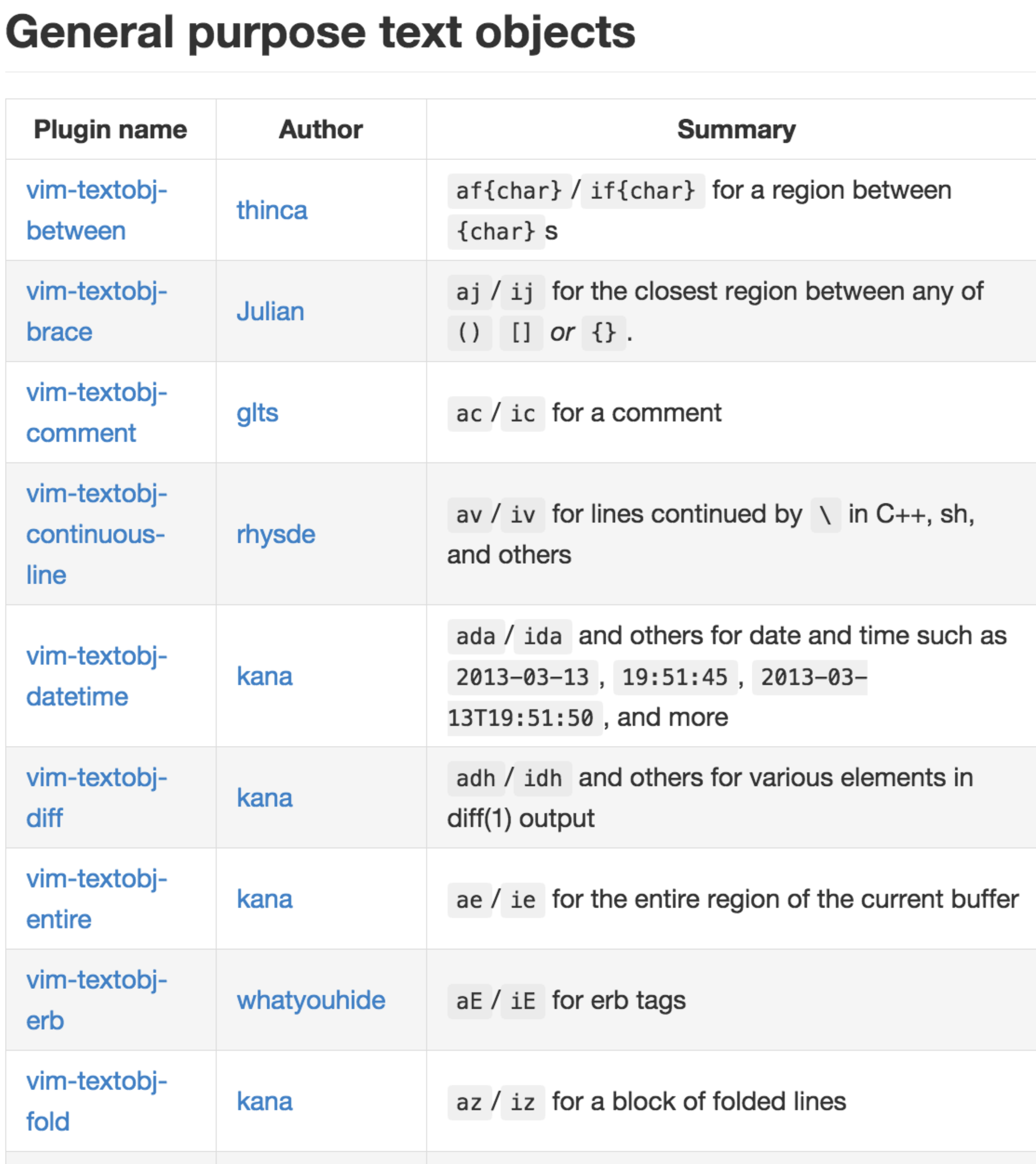
Custom Text Objects

Indent
Entire
Line
Ruby block

Custom Text Objects -- Finding More

Many many more available

textobj-user wiki



General purpose text objects

Plugin name	Author	Summary
vim-textobj-between	thinca	`af{char}` / `if{char}` for a region between `{char}` s
vim-textobj-brace	Julian	`aj` / `ij` for the closest region between any of `()` `[]` *or* `{}` .
vim-textobj-comment	glts	`ac` / `ic` for a comment
vim-textobj-continuous-line	rhysde	`av` / `iv` for lines continued by `\` in C++, sh, and others
vim-textobj-datetime	kana	`ada` / `ida` and others for date and time such as `2013-03-13` , `19:51:45` , `2013-03-13T19:51:50` , and more
vim-textobj-diff	kana	`adh` / `idh` and others for various elements in diff(1) output
vim-textobj-entire	kana	`ae` / `ie` for the entire region of the current buffer
vim-textobj-erb	whatyouhide	`aE` / `iE` for erb tags
vim-textobj-fold	kana	`az` / `iz` for a block of folded lines

In Conclusion

Having a composable language of operations and text objects is one honking great idea -- let's do more of those!