Emacs Configuration

Jed Northridge

September 4, 2013

Contents

1 Overview

I use a simple mechanism for breaking down a single emacs configuration across several component files. Any one of these component files is written in a format that enables that file to simultaneously be more than one thing. Each file surely contains elisp that is used to configure emacs, but that same file also contains a prose explanation behind the motivation and impact of the associated elisp. The same mechanism combines these component pieces to form an overall configuration, ultimately within a single file an in pure elisp, that emacs is capable of interpreting. In the end, the elisp from each file provides a discrete section of the total, generated configuration, just as the prose explanation from each file comes to represent a section within this document.

The value of this mechanism draws from the belief that complexity is reduced when each file can bear a single responsibility for describing one smaller part of a large configuration. This follows from the similar expectation that any one chapter within a larger document can rightly be expected to describe a self-contained idea.

The notion of first creating a prose explanation of how "code" works and then embedding that same code within that explanation is known as Literate Programming. This document describes two such systems. The first, and larger of the two, is a literately programmed emacs configuration. The second is a description of how literate programming is achieved within emacs and then meaningfully used to configure emacs.

1.1 Background: Literate Programming in Emacs

Emacs provides support for literate programming most directly through org-mode, and, within org-mode, babel. When practicing literate programming in this style, you author org-mode documents as normal and you additionally embed source code within these documents. Org-mode gives you strong support for authoring a document of any type: you can export to various formats, structure documents with semantic headings, generate a table of contents, enjoy hyperlink syntax, and more. In addition to basic org-mode functionality, babel then allows you to include source code within these documents. Source code in this context enjoys the same support that emacs provides for normal editing. You can choose any single language or a mix of languages and you work in the mode of the language.

Embedded source code and its easy extraction is the foundation that enables literate programming. Briefly, as you consider programming "a thing," you are given all of the tools you need to write a stand-alone document about that "thing." You are free to layout and follow a narrative that you see fit. This document is my prose explanation of my emacs configuration.

1.2 An Emacs Configuration

When Emacs is started, it normally tries to load a Lisp program from an initialization file, or init file for short. The mechanism I use for managing my configuration is centered around two elisp files. A smaller initialization file, `init.el`, lives in `~/.emacs.d`, a location that is well known to emacs and this file is read on start-up. `init.el` presumes that it will be able to load a much larger elisp file located at `~/emacs.d/emacs-setup/emacs-setup.el`.

These two files, `init.el` and `emacs-setup.el`, are extracted from source code blocks within .org files using babel. Appendix A contains `init.el` in its entirety. The contents of `emac-setup.el` are spread across the remainder of the .org files and can be constructed by concatenating the extract results of all files.

In summary, I use Org-mode and .org files to serve two purposes:

- .org files contain documentation of my configuration. This documentation can be viewed on Github or translated into another format such as LaTeX or HTML.
- .org files contain embedded elisp behind the same configuration. Babel, a feature of Org-mode, can parse a .org file and extract the associated elisp.

Appendix A details how I install and setup emacs. It also contains information about how `init.el` and the start up process works. Appendix B defines a process that is capable of creating `emacs-setup.el` from distinct .org files.

When it comes to elisp, functions and variables associated with my emacs setup will begin with `jedcn-es`. The first variable we define sets the expecation that the directory `emacs-setup/` can be placed (or linked to) underneath `~/.emacs.d`:

```
(setq jedcn-es/dir (concat
                    user-emacs-directory
                    "emacs-setup"))
```

2 General

2.1 Package Repository

Extensions for emacs are known as "packages," and emacs has a built in package management system. Emacs lisp packages are stored in archives (elpas) and, initially, emacs knows about a single such archive: http://elpa.gnu.org. This archive has approximately 50 packages. However, there are additional elpas out there, and I have had good luck finding up-to-date packages in http://melpa.milkbox.net/.

That said, each time I open up emacs I make sure that the package management system is initialized before I configure it to use melpa:

```
(package-initialize)

(add-to-list 'package-archives
             '("melpa" . "http://melpa.milkbox.net/packages/") t)
```

My observation is that packages cannot be installed until a repository is contacted, at least once, and an overview of the contents are downloaded and cached locally.

The following elisp will run `package-list-packages` if `~/.emacs.d/elpa` does not exist, and is rooted in the belief that the first time you run `package-list-packages` the contents of the archive are cached within `~/.emacs.d/elpa`.

```
(unless
    (file-directory-p "~/.emacs.d/elpa")
  (package-list-packages))
```

I am interested in ensuring that the elpa cache has been created so that I can programatically install packages. I first read about this in Sacha Chua's excellent blog post on her configuration: Literate programming and my Emacs configuration file. She defines a function (copied below) that will install the package if it is not present:

```
(defun sacha/package-install (package &optional repository)
  "Install PACKAGE if it has not yet been installed.
If REPOSITORY is specified, use that."
  (unless (package-installed-p package)
    (let ((package-archives (if repository
                                 (list (assoc repository package-archives))
                               package-archives)))
    (package-install package))))
```

In combination, these facilities are the foundation of my package management strategy: initialize the subsystem, configure the repositories, and then define a means to programatically install missing packages. Doing this early on in my initialization process means that code which follows can state, "I expect to have package XYZ," by saying, `(sacha/package-install "XYZ")` and then presume that XYZ is present.

2.2 PATH

Emacs can run shell commands on your behalf. When it does this, it needs to know about the equivalent of your PATH so it can find commands.

I am not sure how this works. There is something that is an environment variabled named `PATH` that is reachable via `(getenv "PATH")` and there is something else that is a elisp variable named `exec-path`.

Rather than interact with my shell and have Emacs learn values from a `$PATH` proper, I am explicit about setting both:

```
(setq jedcn-env-path "/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/texbin")

(defun jedcn-sync-env-path-and-exec-path (desired-path)
  "Sets exec-path and env 'PATH' based on DESIRED-PATH"
  (setenv "PATH" desired-path)
  (setq exec-path (split-string desired-path ":")))

(jedcn-sync-env-path-and-exec-path jedcn-env-path)
```

2.3 UTF-8

I picked this up from Magnars in his sane-defaults.el.

```
(setq locale-coding-system 'utf-8)
(set-terminal-coding-system 'utf-8)
(set-keyboard-coding-system 'utf-8)
(set-selection-coding-system 'utf-8)
(prefer-coding-system 'utf-8)
```

2.4 Start Server

Emacs is often run for hours (or days, or weeks) at a time. One of the benefits of such a long-lived process is that you can build a small tool, like `emacsclient` that can connect to a running emacs and request that something be edited. For example, you can tell git that it should use `emacsclient` whenever it needs to edit something:

```
git config --global core.editor /usr/local/bin/emacsclient
```

That said, here's the elisp that starts up an emacs server:

```
(server-start)
```

2.5 Save Place

I got this one from Magnars: init.el-03.

```
(require 'saveplace)
(setq-default save-place t)
(setq save-place-file (expand-file-name ".places" user-emacs-directory))
```

2.6 Appearance

2.6.1 Color Theme

```
(sacha/package-install 'zenburn-theme)
(load-theme 'zenburn t)
```

2.6.2 Font

I like a bigger font (say, 18) and I vary between "Monaco-18" or "Menlo-18".

```
(set-face-attribute 'default nil :font "Menlo-18")
```

3 Personal Information

```
(setq user-full-name "Jed Northridge"
      user-mail-address "northridge@gmail.com")
```

4 Key Bindings

My main inspiration for keybindings have come from ESK and from Magnars.

Documenting (and configuring) keybindings is somewhat strange. These things appear out of no where, and do not always follow an obvious order.

4.1 See Occurrences while Searching

If you are searching for something, and you press `C-o`, you can see all of the occurrences of that something within the file. Once that **Occur** window comes up, you can press `e` to start editing. You can press `C-c C-c` to get out of it.

```
(define-key isearch-mode-map (kbd "C-o")
  (lambda () (interactive)
    (let ((case-fold-search isearch-case-fold-search))
      (occur (if isearch-regexp isearch-string (regexp-quote isearch-string))))))
```

4.2 Running Methods

When it comes to running methods explicitly, I always use C-x C-m. I picked this up from Steve Yegge's Effective Emacs. He says use `execute-extended-command`, but I always use smex.

```
(global-set-key "\C-x\C-m" 'smex)
```

4.3 Text Size

Making text larger or smaller with ease is something I use every day, several times a day. This happens most commonly when I am showing someone something in emacs (say, pairing or running a meeting), but also when I am at home and do not have my glasses. These particular keybindings are all about the `+` and the `-`.

```
(define-key global-map (kbd "C-+") 'text-scale-increase)
(define-key global-map (kbd "C--") 'text-scale-decrease)
```

4.4 Goto Line

The following makes it so that when I press `C-x g` I can expect to be prompted to enter a line number to jump to it.

```
(global-set-key (kbd "C-x g") 'goto-line)
```

And the elisp below makes it so that whatever goto-line was bound to is now bound to a new function: goto-line-with-feedback.

In turn, goto-line-with-feedback modifies the buffer you are working in to show line numbers but only when you are actively looking to pick a number.

The point of showing line numbers is to give you an idea of where you will end up.

The point of **only** showing them while going to a line is to keep the screen free of distractions (line numbers) unless it is helpful.

This comes from this post within "what the emacs.d."

```
(global-set-key [remap goto-line] 'goto-line-with-feedback)

(defun goto-line-with-feedback ()
  "Show line numbers temporarily, while prompting for the line number input"
  (interactive)
  (unwind-protect
      (progn
        (linum-mode 1)
        (goto-line (read-number "Goto line: ")))
    (linum-mode -1)))
```

4.5 Magit

I like to think "C-x m"agit.

```
(global-set-key (kbd "C-x m") 'magit-status)
```

4.6 MacOS's "Command"

I think keys called 'super' and 'hyper' used to appear on the keyboards of fabled 'Lisp Machines,' as described in this ErgoEmacs post about Super and Hyper Keys. I may take advantage of these some day, but for now I am happy to have both the 'alt/option' key and the 'command' key on my Mac do the same thing: meta.

Given the default setup of my brew installed emacs, the following change makes it so that "command does meta"

If I am back this way in the future again, I'd like to remind myself to consider the following variables: mac-option-modifier, mac-command-modifier, and ns-function-modifer.

```
(setq mac-command-modifier 'meta)
```

4.7 Movement

I rely on standard emacs commands to move around, with the following enhancements:

4.7.1 Using shift makes standard movement 5x faster

This comes from Magnars in this post of whattheemacsd.com.

```
(global-set-key (kbd "C-S-n")
                (lambda ()
                  (interactive)
                  (ignore-errors (next-line 5))))

(global-set-key (kbd "C-S-p")
                (lambda ()
                  (interactive)
                  (ignore-errors (previous-line 5))))

(global-set-key (kbd "C-S-f")
                (lambda ()
                  (interactive)
                  (ignore-errors (forward-char 5))))

(global-set-key (kbd "C-S-b")
                (lambda ()
                  (interactive)
                  (ignore-errors (backward-char 5))))
```

4.7.2 Move current line up or down

This matches what Magnars says in this post, except I also use META.

```
(defun move-line-down ()
  (interactive)
  (let ((col (current-column)))
    (save-excursion
      (forward-line)
      (transpose-lines 1))
    (forward-line)
    (move-to-column col)))

(defun move-line-up ()
  (interactive)
  (let ((col (current-column)))
    (save-excursion
      (forward-line)
      (transpose-lines -1))
    (move-to-column col)))
(global-set-key (kbd "<C-M-S-down>") 'move-line-down)
(global-set-key (kbd "<C-M-S-up>") 'move-line-up)
```

5 Behaviors

. . .
be·hav·ior *bihvyr* Noun

- The way in which one acts or conducts oneself, esp. toward others: "his insulting behavior towards me".
- The way in which an animal or person acts in response to a particular situation or stimulus: "the feeding behavior of predators".

...

5.1 Miscellaneous

Do not "ding" all of the time, and instead flash the screen. Do not show the Emacs "splash" screen.

```
(setq visible-bell t
      inhibit-startup-message t)
```

5.2 Whitespace Cleanup

The following creates a function that cleans up whitespace, and then adds a hook that makes this happen each time you save. It comes from a post within "what the emacs.d," specifically titled buffer defuns.

```
(defun cleanup-buffer-safe ()
  "Perform a bunch of safe operations on the whitespace content of a buffer."
  (interactive)
  (untabify (point-min) (point-max))
  (delete-trailing-whitespace)
  (set-buffer-file-coding-system 'utf-8))

(add-hook 'before-save-hook 'cleanup-buffer-safe)
```

5.3 Yes or No?

Emacs often asks you to type "yes or no" to proceed. As an example, consider when you are in magit, and you press "k" to kill off a hunk. I am happy to have a confirmation before something is deleted, but I prefer to just press "y" instead of "y-e-s-<RETURN>"

```
(defalias 'yes-or-no-p 'y-or-n-p)
```

5.4 Autofill

By observation alone, `auto-fill-mode` makes it so that words wrap around the screen by inserting a new line once you go past a certain spot. I want to auto-fill if I am working on text. When I am programming, I only want to auto-fill if I am writing a comment.

Both of these come from technomancy in v2 of the emacs-starter-kit.

```
(defun esk-local-comment-auto-fill ()
  (set (make-local-variable 'comment-auto-fill-only-comments) t)
  (auto-fill-mode t))
(add-hook 'prog-mode-hook 'esk-local-comment-auto-fill)

(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

5.5 Display Column Numbers when Programming

Show column numbers when programming. This comes from technomancy in v2 of the emacs-starter-kit.

```
(defun esk-local-column-number-mode ()
  (make-local-variable 'column-number-mode)
  (column-number-mode t))

(add-hook 'prog-mode-hook 'esk-local-column-number-mode)
```

5.6 Highlight Current Line when Programming

Highlight the current line. This comes from technomancy in v2 of the emacs-starter-kit.

```
(defun esk-turn-on-hl-line-mode ()
  (when (> (display-color-cells) 8)
    (hl-line-mode t)))

(add-hook 'prog-mode-hook 'esk-turn-on-hl-line-mode)
```

5.7 Use instead of lambda

If you see “lambda” replace it with a . This comes from technomancy in v2 of the emacs-starter-kit.

```
(defun esk-pretty-lambdas ()
  (font-lock-add-keywords
   nil `(("(?\\(lambda\\>\\)"
          (0 (progn (compose-region (match-beginning 1) (match-end 1)
                                    ,(make-char 'greek-iso8859-7 107))
                    nil))))))

(add-hook 'prog-mode-hook 'esk-pretty-lambdas)
```

6 Modes and Packages

Modes bring significant functionality into Emacs. These are the modes (and any associated configuration) that I use.

As I understand it, modes are delivered via packages. You can browse available packages by typing `M-x package-list-packages`. As you move around (just like a regular buffer), if you see something you like you can press `i` and the package on the same line as your cursor will be marked for an upcoming installation. When you are ready, press 'x' to install each package that has been marked in this way.

If I like a package, I'll revisit this file and formally add it to the list of packages I use. In this file, my intent is to provide notes about a mode, why I am using it, what I am doing with it, etc. Further, I want to hook the list of packages that I am using into a system by which they are automatically installed. I am looking to do this to make sure that I can recreate my emacs installation if I move to a new computer. The means by which packages are automatically installed is with a function named `sacha/package-install`. This was defined previously.

6.1 General Package Listing

6.1.1 better-defaults

I started with Emacs Starter Kit, and am following its progression from v1 to v2 and, now, v3. In v3 the esk becomes prose only, and identifies `better-defaults` as a single package with "universal appeal."

```
(sacha/package-install 'better-defaults)
```

6.1.2 smex

When you want to run a command (say, via M-x) smex provides instant feedback by displaying available commands and remembering ones you have recently invoked.

```
(sacha/package-install 'smex)
(setq smex-save-file (concat user-emacs-directory ".smex-items"))
(smex-initialize)
(global-set-key (kbd "M-x") 'smex)
```

6.1.3 markdown-mode

I write in Markdown all the time, and sometimes I use the "compilation" facility of this mode.

If you do start using the compilation aspect, you'll need a command line "markdown" to execute.

I got markdown with `brew install markdown`.

My notes indicate that:

You can change the markdown executable, or read more about the mode, here: http://jblevins.org/projects/markdown-mode/

Also, Highlights:

- `C-c C-c p`: Run markdown on buffer contents. Open result in browser.

I started using markdown-mode+ recently, and I did so after doing a bunch of work to get pandoc installed and working with Emacs.

```
(sacha/package-install 'markdown-mode)
(sacha/package-install 'markdown-mode+)
(add-to-list 'auto-mode-alist '("\\.md$" . markdown-mode))
```

6.1.4 puppet-mode

```
(sacha/package-install 'puppet-mode)
(add-to-list 'auto-mode-alist '("\\.pp$" . puppet-mode))
```

6.1.5 haml-mode

```
(sacha/package-install 'haml-mode)
```

6.1.6 yaml-mode

```
(sacha/package-install 'yaml-mode)
(add-to-list 'auto-mode-alist '("\\.yml$" . yaml-mode))
```

6.1.7 coffee-mode

```
(sacha/package-install 'coffee-mode)
```

6.2 Ruby Packages

I really enjoy writing ruby.

At a high level, my MacOS has RVM installed from http://rvm.io.

Then, my emacs uses a package named rvm that understands how http://rvm.io works, and can direct emacs to use any of the various rubies that rvm provides.

I explicitly use the default ruby from RVM, but Emacs also updates the ruby I'm using each time I start editing a file in ruby-mode. I think this works by looking at the location of the file I'm editing, looking "up" to find the associated .rvmrc or .ruby-version, and then activating it.

With all of that said, my main flow is to run rspec and cucumber from within emacs. This capability is provided by feature-mode and rspec-mode.

The main key bindings I use are:

- `C-c , v`

  Run rspec or cucumber against the file I'm editing

- `C-c , s`

  Run rspec or cucumber against the single line of the spec or feature I'm editing.

6.2.1 rvm

```
(sacha/package-install 'rvm)
```

For emacs, on a MacOS, I believe the following configures my setup so that I'll use the default ruby provided by RVM when I need ruby.

```
(rvm-use-default)
```

I was reading a blog post by Avdi Grimm about how he was using RVM the other day, and that's where I picked up the following helpful snippet that works with the emacs rvm subsystem to activate the correct version of ruby each time you open a ruby-based file:

```
(add-hook 'ruby-mode-hook
          (lambda () (rvm-activate-corresponding-ruby)))
```

6.2.2 feature-mode

I don't often write Gherkin at work, but I do try to use Cucumber whenever I get the chance on side projects. So far I've been using this mode mainly for syntax highlighting.

```
(sacha/package-install 'feature-mode)
```

6.2.3 rspec-mode

I **love** rspec.

```
(sacha/package-install 'rspec-mode)
```

I also have been using ZSH, and when I was getting rspec-mode up and running a few months ago, I ran into trouble. Thankfully, the author of rspec mode had a solution for using rspec mode with ZSH.

```
(defadvice rspec-compile (around rspec-compile-around)
  "Use BASH shell for running the specs because of ZSH issues."
  (let ((shell-file-name "/bin/bash"))
    ad-do-it))
(ad-activate 'rspec-compile)
```

6.2.4 ruby-mode

For now, the main thing I do is turn on ruby-mode when I'm editing well known file types:

```
(add-to-list 'auto-mode-alist '("\\.rake$" . ruby-mode))
(add-to-list 'auto-mode-alist '("\\.gemspec$" . ruby-mode))
(add-to-list 'auto-mode-alist '("\\.ru$" . ruby-mode))
(add-to-list 'auto-mode-alist '("Rakefile$" . ruby-mode))
(add-to-list 'auto-mode-alist '("Gemfile$" . ruby-mode))
```

```
(add-to-list 'auto-mode-alist '("Capfile$" . ruby-mode))
(add-to-list 'auto-mode-alist '("Vagrantfile$" . ruby-mode))
(add-to-list 'auto-mode-alist '("\\.thor$" . ruby-mode))
(add-to-list 'auto-mode-alist '("Thorfile$" . ruby-mode))
(add-to-list 'auto-mode-alist '("Guardfile" . ruby-mode))
```

6.2.5 ruby-electric

This minor mode automatically inserts a right brace when you enter a left brace, or an "end" when you define a def.

```
(sacha/package-install 'ruby-electric)
```

6.3 Magit

I **love** magit.

Beyond cosmetics, here are two great blog posts about magit: Setup Magit #1 and Setup Magit #2. The main points are:

- Give Magit full screen when you start it.
- Setup Magit so that pressing "q" gets rid of full screen.
- Setup Magit so that pressing "W" toggles paying attention to whitespace.

I happen to have `emacsclient` installed in two places, one at `/usr/bin` and another at `/usr/local/bin`. The one at `/usr/bin` cannot find my emacs server and this causes Magit to freeze whenever I try to commit. This is why I explicitly set `magit-emacsclient-executable`.

```
(sacha/package-install 'magit)

(require 'magit)

(defadvice magit-status (around magit-fullscreen activate)
  (window-configuration-to-register :magit-fullscreen)
  ad-do-it
  (delete-other-windows))

(defun magit-quit-session ()
  "Restores the previous window configuration and kills the magit buffer"
  (interactive)
  (kill-buffer)
  (jump-to-register :magit-fullscreen))

(define-key magit-status-mode-map (kbd "q") 'magit-quit-session)

(defun magit-toggle-whitespace ()
  (interactive)
  (if (member "-w" magit-diff-options)
      (magit-dont-ignore-whitespace)
    (magit-ignore-whitespace)))

(defun magit-ignore-whitespace ()
  (interactive)
  (add-to-list 'magit-diff-options "-w")
  (magit-refresh))

(defun magit-dont-ignore-whitespace ()
  (interactive)
  (setq magit-diff-options (remove "-w" magit-diff-options))
  (magit-refresh))

(define-key magit-status-mode-map (kbd "W") 'magit-toggle-whitespace)

(setq magit-emacsclient-executable "/usr/local/bin/emacsclient")
```

6.4 yasnippet

My favorite snippet to use is `dbg`, which I found in Jim Weirich's emacs setup here.

```
(sacha/package-install 'yasnippet)
(require 'yasnippet)
(setq yas-snippet-dirs (concat jedcn-es/dir "/snippets"))
```

When I was setting up yasnippet, I saw the following in the official documentation:

```
(yas-global-mode 1)
```

6.5 org-mode

OrgMode is a wonderful thing.

6.5.1 Defaults

When I open a .org file, I like to see all of the headlines but none of the text:

```
(setq org-startup-folded 'content)
```

Hiding the stars looks cleaner to me:

```
(setq org-hide-leading-stars 'hidestars)
```

6.5.2 Code Blocks

These emacs configuration files (.org, .el) use org's "code blocks" extensively, and the following has Emacs pay attention to the type of code within the blocks.

```
(setq org-src-fontify-natively t)
```

Editing Code Blocks With your cursor over one of these code blocks you can type C-c ' and a new buffer will open for editing just that content.

Executing Code Blocks With your cursor over one of these code blocks you can type C-c C-c and, if the code block is one of the languages that has been configured to be run, the block will be executed and the results printed nearby.

By default, only emacs-lisp is configured to be executed. The following block makes it so that ruby is too.

Here's the documentation for this: babel/languages.

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((emacs-lisp . t)
   (ruby . t)))
```

7 Various and Sundry

7.1 Jim Weirich's eval-buffer

I saw Jim Weirich give a great talk at one of the keynotes of Ruby Conf 2012. The way he used buffer evaluation was just awesome!

His setup (which I think is described below) allows him to consistently show you one piece of code and then pair that code up with the output that comes from executing it.

Unlike using an inferior-ruby process, the resulting code output has very little noise.

You can find the original code that he wrote right here.

The only thing I've changed is the variable `jw-eval-buffer-commands` and instead I've created `jedcn-eval-buffer-commands` just because I do not have xruby.

```
(defconst jedcn-eval-buffer-commands
  '(("js" . "/usr/local/bin/node")
    ("rb" . "ruby")
    ("coffee" . "/usr/local/bin/coffee")
    ("clj" . "/Users/jim/local/bin/clojure")
    ("py" . "/usr/bin/python")))

(defconst jw-eval-buffer-name "*EVALBUFFER*")

(defun jw-eval-buffer ()
  "Evaluate the current buffer and display the result in a buffer."
  (interactive)
  (save-buffer)
  (let* ((file-name (buffer-file-name (current-buffer)))
         (file-extension (file-name-extension file-name))
         (buffer-eval-command-pair (assoc file-extension jedcn-eval-buffer-commands)))
    (if buffer-eval-command-pair
        (let ((command (concat (cdr buffer-eval-command-pair) " " file-name)))
          (shell-command-on-region (point-min) (point-max) command jw-eval-buffer-name
          (pop-to-buffer jw-eval-buffer-name)
          (other-window 1)
          (jw-eval-buffer-pretty-up-errors jw-eval-buffer-name)
          (message ".."))
      (message "Unknown buffer type"))))
```

```
(defun jw-eval-buffer-pretty-up-errors (buffer)
  "Fix up the buffer to highlight the error message (if it contains one)."
  (save-excursion
    (set-buffer buffer)
    (goto-char (point-min))
    (let ((pos (search-forward-regexp "\\.rb:[0-9]+:\\(in.+:\\)? +" (point-max) t)))
      (if pos (progn
                (goto-char pos)
                (insert-string "\n\n")
                (end-of-line)
                (insert-string "\n"))))))

(defun jw-clear-eval-buffer ()
  (interactive)
  (save-excursion
    (set-buffer jw-eval-buffer-name)
    (kill-region (point-min) (point-max))))

(defun jw-eval-or-clear-buffer (n)
  (interactive "P")
  (cond ((null n) (jw-eval-buffer))
        (t (jw-clear-eval-buffer))))
```

8 Appendix A: Installation Details

This appendix covers both how I install Emacs on MacOS and how I get up and running with `emacs-setup`.

8.1 Emacs Installation

On MacOS I install Emacs using Homebrew. I run the following at my shell prompt:

```
brew install emacs --cocoa
```

This takes some time to complete, and when finished I take another step to make Emacs appear as one of my Applications:

```
ln -s /usr/local/Cellar/emacs/24.3/Emacs.app /Applications
```

Now I can start emacs by selecting it graphically in the Applications area.

8.2 Using emacs-setup

Once I have Emacs 24+ running, I use git to clone my `emacs-setup` to my machine, cd into the cloned directory, and source the file `install.sh`:

```
git clone https://github.com/jedcn/emacs-setup.git
cd emacs-setup
source install.sh
```

The contents of the `install.sh` file achieve the following:

- They allow you to supply a `HOME` and will create an `.emacs.d` if needed.
- They create a link within this `.emacs.d` back to the cloned `emacs-setup`.
- They create a single line of elisp that loads up the composite `emacs-setup.el`.

```
emacs_setup_dir=`pwd`

echo "Creating $HOME/.emacs.d (if needed)"
mkdir -p $HOME/.emacs.d

echo "Creating $HOME/.emacs.d/emacs-setup as link to $emacs_setup_dir"
ln -s $emacs_setup_dir $HOME/.emacs.d/emacs-setup

echo "Creating $HOME/.emacs.d/init.el"
echo '(load (concat user-emacs-directory "emacs-setup/emacs-setup.el"))' >> $HOME/.ema
```

It is important to note that `HOME` can be given a temporary value and this lets me test my installation process. I can get a fresh copy of `emacs-setup` and clone it into a temporary directory, and then I can run the `install.sh` with a temporary value of `HOME` like so:

```
mkdir /tmp/emacs-setup && cd /tmp/emacs-setup
git clone https://github.com/jedcn/emacs-setup.git .

mkdir /tmp/emacs-home
HOME=/tmp/emacs-home source install.sh
HOME=/tmp/emacs-home /Applications/Emacs.app/Contents/MacOS/Emacs &
```

9 Appendix B: Babel and the Config

My configuration is stored as several .org files. This is done to optimize for editing and the production of documentation (via `org-export`). However, emacs does not read these .org files and instead it reads a single elisp file, `emacs-setup.el`.

How is a single elisp file generated from several .org files? The .org files are concatenated together in a specific order to create a composite .org file named `emacs-setup.org`. This composite file can be used to generate `emacs-setup.el`, and it can also generate complete documentation in various formats: HTML or LaTeX/PDF.

Emacs has built in support for extracting and loading elisp within .org files via `org-babel-load-file`. Why not just use this on each .org file individually rather than orchestrating a process by which they are concatenated into a single, larger document? I want to focus on woven documentation. Why not just operate on just a larger .org file? I want to work towards modularity. Putting these two concepts together, I think of each .org file as a stand-alone entity that is both chapter in a larger story and section in a larger program.

The remainder of this appendix details how this orchestration works. All of the functions and variables in this section begin with `jedcn-es/` to indicate their logical association with my (`jedcn`) emacs setup (`es`).

9.1 Composite File

The name of the composite .org file is `emacs-setup.org`, and its location is stored for future reference in `composite-org`.

```
(setq jedcn-es/composite-org (concat
                              jedcn-es/dir
                              "/emacs-setup.org"))
```

9.2 Component Files

The list of files that will be included in the is stored in `files`. Order is significant. These files are presumed to be within `files-dir`.

```
(setq jedcn-es/files-dir (concat
                          jedcn-es/dir
                          "/org"))
```

```
(setq jedcn-es/files '("introduction.org"
                       "general-setup.org"
                       "personal-information.org"
                       "key-bindings.org"
                       "behaviors.org"
                       "modes.org"
                       "various-and-sundry.org"
                       "appendix-a.org"
                       "appendix-b.org"))
```

9.3 Concatenation

The composite file is created with `create-composite-org`, which in turn relies on `concat-files`, `files-dir`, and `files`, and `composite-org`.

```
(defun jedcn-es/concat-files (the-files target-file)
  "Concatenate a list of THE-FILES into a TARGET-FILE"
  (let* ((original-buffer (current-buffer))
         (result-file target-file)
         (files the-files)
         (file (car files)))
    ;; do..
    (find-file file)
    (write-region (point-min) (point-max) result-file)
    (setq files (cdr files))
    (setq file (car files))
    ;; while
    (while files
      (find-file file)
      (write-region (point-min) (point-max) result-file t)
      (setq files (cdr files))
      (setq file (car files)))
    (switch-to-buffer original-buffer)))

(defun jedcn-es/create-composite-org ()
  "Create a composite org file based on my list of config files"
  (jedcn-es/concat-files
   (mapcar (lambda (file)
             (concat jedcn-es/files-dir "/" file))
           jedcn-es/files)
  jedcn-es/composite-org))
```

9.4 Extracting elisp

Literate programming uses the verb “tangling” to describe the extraction of pure source code from its annotated source. We’ll aim to extract the elisp from `composite-org` and place it into `composite-el`.

```
(setq jedcn-es/composite-el (concat jedcn-es/dir "/emacs-setup.el"))
```

Babel supports code extraction with a function named `org-babel-tangle-file`, and we can hook into the process described above as follows:

```
(defun jedcn-es/tangle-composite-org ()
  (org-babel-tangle-file jedcn-es/composite-org jedcn-es/composite-el))
```

When this has completed, we will have the tangled result residing at `composite-el`. The next logical step is to load it up and try it out:

```
(defun jedcn-es/load-composite-el ()
  (load-file jedcn-es/composite-el))
```

Stepping back, we can bundle up the creation of the composite .org file from its component pieces, then tangle it, and then load the result with `jedcn-es/rebuild-and-reload`:

```
(defun jedcn-es/rebuild-and-reload ()
  "Rebuild the composite .org file, extract the elisp, and reload"
  (interactive)
  (jedcn-es/create-composite-org)
  (jedcn-es/tangle-composite-org)
  (jedcn-es/load-composite-el))
```

This is the only function I make interactive. In practice, this means that I can fool around with my .org files, and then `M-x jedcn-es/rebuild-and-reload` to try out the latest changes.