Smart Documents

Marius Peter
<2021-02-09 Tue>

Contents

List of Figures

List of Tables

Abstract

The idea of *Smart Documents* came to me as I was reflecting on how to improve the document creation process in my workplace. The GNU Emacs editor had captured my imagination, and I wanted to create an accessible and highly productive text editor to benefit my organization. In this paper, I'll lay out my vision for the *Smart Document*, a file containing both text destined to the reader, and code describing how to update, validate, and present this text; then, I'll weave my personal GNU Emacs customizations with a tutorial. This paper is a *Smart Document* itself!

1 Introduction

The following sections were laid out very deliberately. When we start Emacs, the source code blocks contained in this document are evaluated sequentially — our editing environment is constructed in real time as we execute the blocks in order. For instance, we only begin loading packages once we ensured `use-package` is working properly.[1]

Customizing Emacs goes far, far beyond rewriting sections of this document — feel free to experiment and discover. Here are three commands that will help you understand all the symbols in this file, if you are browsing this paper within Emacs itself:

`C-h f` describe function

`C-h v` describe variable

`C-h k` describe key

You can press `f1` at any time to access Emacs built-in help.

1.1 TODO User details

One advantage of working with *Smart Documents* is that they can automatically be populated with our details in the header, footer, or other appropriate element.

```
(setq user-full-name "Marius Peter")

(defun my/user-details-get ()
  "Get user details."
  (interactive)
  (setq user-full-name (read-string "Enter full user name:"))
  (setq user-mail-address (read-string "Enter user e-mail address:"))
  (message "Successfully captured user details."))

(defun my/tokenize-user-details ()
  "Tokenize user details."
```

[1] For more information on the detailed steps Emacs takes upon starting, refer to `https://www.gnu.org/software/emacs/manual/html_node/elisp/Startup-Summary.html`.

```
  (cons 'user-full-name user-full-name))

(unless (file-exists-p (concat user-emacs-directory
       "meta/user-details"))
  (setq user-details '(user-full-name
       user-mail-address))
  (append-to-file "Foobar\n" nil "~/.emacs.d/meta/foobar"))
```

1.2 File system paths

In this subsection, we tell Emacs about relevant paths to resources.

On my MS Windows machine, I add the path to Portable Git.[2]

```
(when (string-equal system-type "windows-nt")
  (add-to-list 'exec-path "C:/Users/marius.peter/PortableGit/bin/"))
```

2 Early setup

2.1 The first file to load

The contents of this Section was automatically moved to `~/.emacs.d/init.el`. Use 'sd-pack-section' to copy the contents back into this section.

This is the very first user-editable file loaded by Emacs.[3] In it, we disable GUI elements that would otherwise be loaded and displayed once Emacs is ready to accept user input.

It can be found here: `early-init.el`

2.2 The second file to load

> Traditionally, file `~/.emacs` is used as the init file, although Emacs also looks at the following locations:
>
> - `~/.emacs.el`
> - `~/.emacs.d/init.el`
> - `~/.config/emacs/init.el`.
>
> From the GNU website[4]

This file can be found here: `init.el`

If no file is found, Emacs then loads in its purely vanilla state.

[2] Download from `https://git-scm.com/download/win`

[3] This feature became available in version 27.1.

[4] `https://www.gnu.org/software/emacs/manual/html_node/emacs/Init-File.html`

2.3 Profiling — start

We start the profiler now , and will interrupt it in Section 9.1. We will then present profiling report in Section 9.2.

```
(profiler-start)
```

2.4 Usual paths to files & directories

This section defines the paths that structure the overall *Smart Document* logic.

File A collection of information recognized by a computer — to us, a normal file appears as programs (source code or *executables*) or data (an image, a video, a document...). It is usually stored in a computer's memory or on a storage device (hard drive, USB drive...)

Path A description of a file's location from the user's perspective. A path is program searches for a file or executable program.

Directory Synonymous with *folder*. From the computer's — and Emacs' — perspective, a file which contains other files. These files may themselves be directories.

2.4.1 Meta files — files about files

In this section, we'll be tidying up the `.emacs.d/` directory — by default, many Emacs packages create files useful for themselves in our `user-emacs-directory`. This leads to undesirable clutter. Certain packages create files that log recently visited files (1); log location of known projects (3); log location in recently visited files (5) The commonality between all these files is that they tend to reference... other files. Thus, I decided to refer to them as meta files.

A meta files should follow the following best principles:

Be located at sd-path-meta This ensures a tidy

Be explicit Meta filenames should not begin with a period: they would be hidden by default on GNU/Linux systems. Novices must see all files by default.

```
(defcustom sd-path-meta
  (concat user-emacs-directory "meta/")
  "Directory containing files about files.")
```

1. Recently visited files

   ```
   (setq recentf-save-file
         (concat sd-path-meta "recentf"))
   ```

2. File bookmarks

```
(setq bookmark-default-file
      (concat sd-path-meta "bookmarks"))
```

3. Projects' bookmarks

```
(setq projectile-known-projects-file
      (concat sd-path-meta "projectile-bookmarks.eld"))
```

4. Org id locations

> We track IDs through files, so that links work globally. The file defined at org-id-locations-file maintains a hash table for IDs and writes this table to disk when exiting Emacs. Because of this, it works best if you use a single Emacs process, not many.
> Paraphrased from the Emacs help interface.

```
(setq org-id-locations-file
      (concat sd-path-meta "org-id-locations"))
;; The leading period is removed because no files are hidden in the
;; metafiles' directory.
```

5. Location in previously visited file

```
(setq save-place-file
      (concat sd-path-meta "places"))
```

6. Auto save file lists

```
(setq auto-save-list-file-prefix
      (concat sd-path-meta "auto-save-list/.saves-"))
```

2.4.2 Resources

All third-party resources are saved at the following location.

```
(defcustom sd-path-resources
  (concat user-emacs-directory "resources/")
  "Directory containing the third-party resources.
Resources may be any data that is not auto-generated during Emacs
startup.")
```

1. Packages

```
(setq package-user-dir
      (concat sd-path-resources "elpa/"))
```

2. Themes

```
(setq custom-theme-directory
      (concat sd-path-resources "themes/"))
```

3. Snippets

   This path, specifically, is required to be in list form.

```
(setq yas-snippet-dirs
      (list (concat sd-path-resources "snippets/")))
```

4. Templates

```
(setq sd-path-templates
      (concat sd-path-resources "templates/"))
```

5. Emojis

```
(setq emojify-emojis-dir
      (concat sd-path-resources "emojis/"))
```

6. Custom file

   Load settings created automatically by GNU Emacs Custom. (For example, any clickable option/toggle is saved here.) Useful for fooling around with `M-x customize-group <package>`.

```
(setq custom-file (concat sd-path-resources "custom.el"))
(load custom-file)
```

2.5 Backups

Backups are very important!

```
(setq backup-directory-alist
      `((".*" . ,temporary-file-directory))
      auto-save-file-name-transforms
      `((".*" ,temporary-file-directory t))
      backup-by-copying t    ; Don't delink hardlinks
      version-control t      ; Use version numbers on backups
      delete-old-versions t  ; Automatically delete excess backups
      kept-new-versions 20   ; how many of the newest versions to keep
      kept-old-versions 5)   ; and how many of the old
```

2.6 Initial and default frames

We set the dimensions of the initial frame:

```
(add-to-list 'initial-frame-alist '(width  . 70))
(add-to-list 'initial-frame-alist '(height . 40))
```

We also set the dimensions of subsequent frames:

```
(add-to-list 'default-frame-alist '(width  . 70))
(add-to-list 'default-frame-alist '(height . 40))
```

Transparency.

```
(set-frame-parameter (selected-frame) 'alpha '(<active> . <inactive>))
(set-frame-parameter (selected-frame) 'alpha <both>)

(set-frame-parameter (selected-frame) 'alpha '(90 . 70))
(add-to-list 'default-frame-alist '(alpha . (90 . 70)))
```

2.7 Secrets

The code contained in the `secrets.org` file is loaded by Emacs, but not rendered in this PDF for the sake of privacy. It contains individually identifying information such as names and e-mail addresses, which are used to populate Org templates (Section 5). You need to create this `secrets.org` file, as it is ignored by `git` by default.

```
(let ((secrets (concat user-emacs-directory "secrets.org")))
  (when (file-exists-p secrets) (org-babel-load-file secrets)))
```

3 Keyboard shortcuts

What follows are the most useful keybindings, as well as the keybindings to the functions we defined ourselves. It doesn't matter if we haven't defined the functions themselves yet; Emacs will accept a keybinding for any symbol and does not check if the symbol's function definition exists, until the keybinding is pressed.

3.1 CUA mode

The following bindings strive to further enhance CUA mode.[5]

```
(cua-mode)
```

[5] Common User Access. This is a term coined by IBM which has influenced user navigation cues on all modern desktop OSes. From IBM's CUA, we get the `Ctrl-c` and `Ctrl-v` keyboard shortcuts.

3.2 Files

3.2.1 Save a file

```
(global-set-key (kbd "C-s") 'save-buffer)
```

3.2.2 Open a file

```
(global-set-key (kbd "C-o") 'find-file)
```

3.2.3 List open files

```
(global-set-key (kbd "C-b") 'ivy-switch-buffer)
```

3.2.4 Open this very file

(Function defined in Section 6.1.1)

```
(global-set-key (kbd "C-c c") 'sd-find-literate-config)
```

3.2.5 Open the Org diary

```
(global-set-key (kbd "C-c d") 'sd-find-org-diary)
```

3.2.6 Capture Org content

```
(global-set-key (kbd "C-c k") 'org-capture)
```

3.2.7 Open a recently visited file

```
(global-set-key (kbd "C-r") 'counsel-recentf)
```

3.2.8 Locate a file

```
(global-set-key (kbd "C-c l") 'counsel-locate)
```

3.2.9 Open the agenda

```
(global-set-key (kbd "C-c a") 'org-agenda)
```

3.2.10 Open the diary

```
(global-set-key [f9]
'(lambda ()
   "Load `org-agenda-diary-file'."
   (interactive)
   (find-file org-agenda-diary-file)))
```

3.2.11 Open Org mode document properties

```
(global-set-key [f8] 'sd-document-properties)
```

3.3 Windows

3.3.1 Close window and quit

The following bindings lead to more natural window & frame exit behaviors.

```
(global-set-key (kbd "C-w") 'sd-delete-window-or-previous-buffer)
```

3.4 Frame

3.4.1 Make new frame

```
(global-set-key (kbd "C-n") 'make-frame)
```

3.4.2 Make only frame

```
(global-set-key (kbd "C-`") 'delete-other-windows)
```

3.4.3 Delete frame or kill Emacs

```
(global-set-key (kbd "C-q") 'sd-delete-frame-or-kill-emacs)
```

3.4.4 Remap `C-z` when using a graphical interface

By default, `C-z` suspends the editor. This is extremely handy when the editor is started with the `-nw` option (no window, i.e. launched in a terminal), because it returns control to the terminal command line without quitting Emacs — it simply places the Emacs process in the background. The user may then use the Linux job management tools to return inside the Emacs process.

However, when using a graphical display, we have no need for suspending the frame, so we remap `C-z` to the much more sensible `undo` behaviour.

```
(when (display-graphic-p)
  (global-set-key (kbd "C-z") 'undo))
```

3.5 Text display

3.5.1 Zoom

The typical binding on both GNU/Linux and MS Windows is adequate here: `C-=` to zoom in, `C--` to zoom out.

It seems that starting with Emacs 27.1, Control + mousewheel works.

```
(global-set-key (kbd "C--") 'text-scale-decrease)
(global-set-key (kbd "C-=") 'text-scale-increase)
(global-set-key (kbd "C-+") 'text-scale-increase)
```

3.6 Accessing customization

3.6.1 Customize a variable

```
(global-set-key (kbd "C-c v") 'customize-variable)
```

3.6.2 Customize a face

```
(global-set-key (kbd "C-c f") 'customize-face)
```

3.7 One-click workflows

A major advantage of the Emacs document production system: arbitrarily complicated functions can be assigned to very simple keybindings. This means we can automate workflows up to a pretty absurd level.

3.7.1 Export to PDF

PDF is probably the most prevalent file format for sharing static documents.

1. Document

   ```
   (global-set-key (kbd "C-p") 'sd-quick-export)
   ```

2. **TODO** Presentation

3.7.2 Clean up buffer

Clean up buffer in every mode.

```
(global-set-key [f12] 'sd-beautify-buffer)
```

4 Packages

Packages are collections of `.el` files providing added functionality to Emacs.

4.1 Meta

How do we bootstrap packages? First, let's figure out:

1. Where we get our packages from
2. How we upgrade packages
3. How we ensure our required packages are installed

4.1.1 Package archives

List of package archives.

```
(require 'package)
(add-to-list 'package-archives '("melpa" . "https://melpa.org/packages/") t)
(add-to-list 'package-archives '("org" . "https://orgmode.org/elpa/") t)
(package-initialize)
```

4.1.2 TODO Convenient package update

One-function rollup of upgradeable package tagging, download and lazy install.

4.1.3 `use-package`

We ensure `use-package` is installed, as well as all packages described in this configuration file.

```
(unless (package-installed-p 'use-package)
  (package-refresh-contents)
  (package-install 'use-package)
  (eval-when-compile (require 'use-package)))
(setq use-package-always-ensure t)
(require 'use-package)
(require 'bind-key)
```

4.2 Spelling, completion, and snippets

The following customizations open the doors to vastly increased typing speed and accuracy.

4.2.1 Syntax checking

We require a package to highlight syntax errors and warnings. The `flycheck` package ensures we are aware of all our code's syntactical shortcomings.

```
(use-package flycheck)
(global-flycheck-mode)
```

4.2.2 Spelling

```
(use-package flyspell)
(add-hook 'text-mode-hook 'flyspell-mode)
```

4.2.3 Completion

```
(use-package company)
(add-hook 'after-init-hook 'global-company-mode)
```

4.2.4 Insert template from keyword

Thanks to `yasnippet`, we can type certain keywords, then press `TAB`, to automatically insert a predefined text snippet. We can then navigate through the snippet by using `<tab>` (next field) and `<backtab>` (previous field).[6]

For instance: typing `src` then pressing `TAB` will expand the keyword to the following text:

```
#+BEGIN_SRC emacs-lisp :tangle yes

#+END_SRC
```

We notice that emacs-lisp is highlighted — this is the first modifiable field. Many clever programming tricks can be performed with `yasnippet` to save us a ton of time with boilerplate text!

```
(use-package yasnippet)
(yas-global-mode 1)
```

4.2.5 Delete all consecutive whitespaces

```
(use-package hungry-delete)
(global-hungry-delete-mode)
```

4.3 Utilities

4.3.1 Versioning of files

Wonderful Git porcelain for Emacs. Enables the administration of a Git repository in a pain-free way.

```
(use-package magit
  :bind ("C-c g" . magit-status))
```

4.3.2 Navigate between projects

This enables us to better manage our `.git` projects.

```
(use-package projectile
  :bind ("C-c p" . 'projectile-command-map)
  :init (projectile-mode 1)
  (setq projectile-completion-system 'ivy))
```

4.3.3 Display keyboard shortcuts on screen

```
(use-package which-key
  :init (which-key-mode))
```

[6] `<backtab>` is synonymous with pressing shift-tab.

4.3.4 Jump to symbol's definition

`dumb-jump` is a reliable symbol definition finder. It uses different matching algorithms and heuristics to provide a very educated guess on the location of a symbol's definition.

```
(use-package dumb-jump)
(add-hook 'xref-backend-functions #'dumb-jump-xref-activate)
```

4.3.5 Graphical representation of file history

```
(use-package undo-tree)
(global-undo-tree-mode)
```

4.3.6 Auto-completion framework

```
(use-package ivy
  :config (setq ivy-use-virtual-buffers t
ivy-count-format "%d/%d "
enable-recursive-minibuffers t))
(ivy-mode t)
```

1. Smartly suggesting interactive search matches

   And he will be called Wonderful **Counselor**, Mighty God, Everlasting Father, Prince of Peace.

   ```
   (use-package counsel
     :bind ("M-x" . counsel-M-x)
     :config (counsel-mode t))
   ```

2. Searching for items in current buffer

   ```
   (use-package swiper
     :bind (("C-f" . swiper)))
   ```

4.3.7 IRC

Emacs ships with an IRC client called `erc`.

```
(use-package erc
  :custom
  (erc-autojoin-channels-alist '(("freenode.net"
  "#linux"
  "#archlinux"
  "#emacs"
  "#bitcoin"
  "#latex"
```

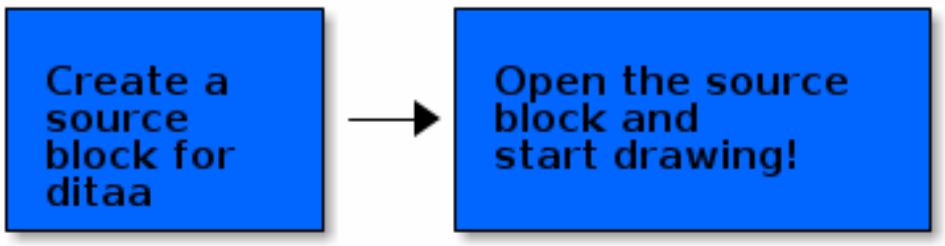
```
  "#org-mode"
  "#python")))
  (erc-autojoin-timing 'ident) ; Autojoin after NickServ identification.
  (erc-fill-function 'erc-fill-static)
  (erc-fill-static-center 16)
  ;; (erc-hide-list '("JOIN" "PART" "QUIT"))
  (erc-lurker-hide-list '("JOIN" "PART" "QUIT"))
  (erc-lurker-threshold-time (* 3600 4)) ; Four hours
  (erc-prompt-for-nickserv-password nil)
  (erc-server-reconnect-attempts 5)
  (erc-server-reconnect-timeout 3)
  :config
  (add-to-list 'erc-modules 'spelling)
  (erc-services-mode 1)
  (erc-update-modules))
```

4.3.8 TODO Telegram

Yeah, a Telegram client exists for Emacs.

```
(use-package telega
  :load-path "~/telega.el/telega.el"
  :commands (telega)
  :defer t)
```

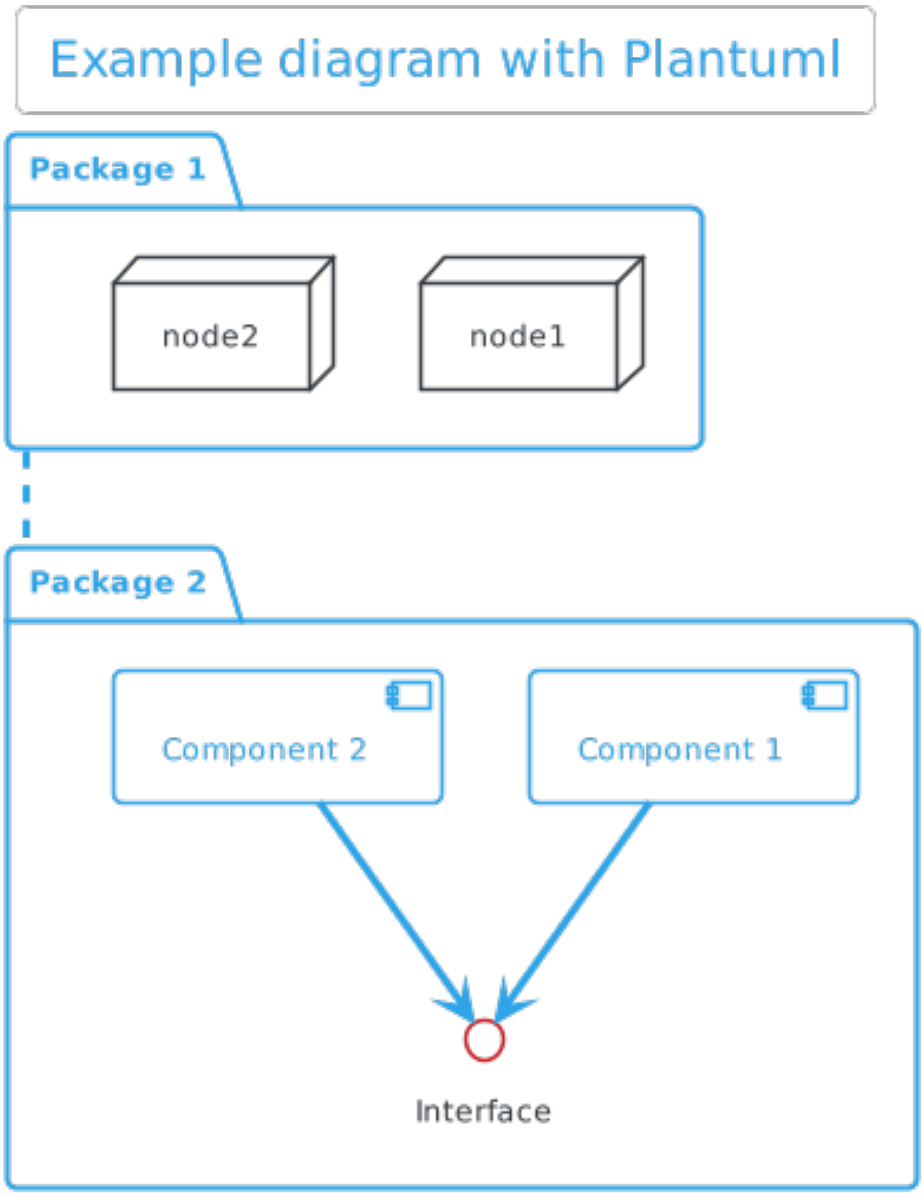
4.3.9 Drawings



4.3.10 TODO UML diagrams

```
;; (require 'plantuml-mode)
(use-package plantuml-mode)
```

```
(setq plantuml-default-exec-mode 'jar
      plantuml-jar-path (concat sd-path-resources
"executables/plantuml.jar")
      org-plantuml-jar-path (concat sd-path-resources
    "executables/plantuml.jar"))
```



4.4 Coding languages

4.4.1 TODO Emacs Lisp

4.4.2 Python

Python is included by default on most Linux distributions.

```
(use-package py-yapf)
(add-hook 'python-mode-hook 'py-yapf-enable-on-save)
```

4.4.3 OCaml

```
(use-package tuareg)
```

4.4.4 Haskell

```
(use-package haskell-mode)
```

4.4.5 Lua

```
(use-package lua-mode)
```

4.5 File formats

These aren't tied to a particular language per se.

4.5.1 `csv` and Excel

```
(use-package csv-mode)
```

4.5.2 Interacting with PDFs

Org mode shines particularly when exporting to PDF — Org files can reliably be shared and exported to PDF in a reproducible fashion.

```
(use-package pdf-tools)
(unless (string-equal system-type "windows-nt")
(pdf-tools-install))
```

4.5.3 Accounting

Ledger is a creation of John Wiegley's. It enables double-entry accounting in a simple plaintext format, and reliable verification of account balances through time.[7]

[7] For more information, visit `https://www.ledger-cli.org/`.

```
(use-package ledger-mode
  :bind
  ("C-c r" . ledger-report)
  ("C-c C" . ledger-mode-clean-buffer))
```

These reports can be generated within Emacs. It is quite useful to pipe their output to an automated "smart document".

```
(setq ledger-reports
      '(("bal" "%(binary) -f %(ledger-file) bal")
("bal-USD" "%(binary) -f %(ledger-file) bal --exchange USD")
("reg" "%(binary) -f %(ledger-file) reg")
("net-worth" "%(binary) -f %(ledger-file) bal ^Assets ^Liabilities --
exchange USD")
("net-income" "%(binary) -f %(ledger-file) bal ^Income ^Expenses --exchange USD -
depth 2 --invert")
("payee" "%(binary) -f %(ledger-file) reg @%(payee)")
("account" "%(binary) -f %(ledger-file) reg %(account)")
("budget" "%(binary) -f %(ledger-file) budget --exchange USD")))
```

4.5.4 Plotting & charting

```
(use-package gnuplot)
```

4.6 Cosmetics

4.6.1 Start page

We replace the standard welcome screen with our own.

4.6.2 Better parentheses

```
(use-package rainbow-delimiters
  :config (add-hook 'prog-mode-hook #'rainbow-delimiters-mode))
(show-paren-mode 1)
```

4.6.3 Highlight *color* keywords in that color

This highlights hexadecimal numbers which look like colors, in that same color.

```
(use-package rainbow-mode
  :init
  (add-hook 'prog-mode-hook 'rainbow-mode))
```

4.6.4 Minor modes in mode line

We hide minor modes in the mode line.

```
(use-package rich-minority)
(rich-minority-mode 1)
(setf rm-whitelist "projectile")
```

4.6.5 Emojis

Emojis are a symbol of modernity, and their tasteful use enables communicating with people from around the world — we're all for that! B-) ◡̈

```
(when (string-equal system-type "gnu/linux")
  (use-package emojify
    :hook (after-init . global-emojify-mode)))
```

5 org-mode

Org mode is so significant that this section of the paper deserves its own introduction.

5.1 Introduction

Phew, after all this initialization, I can finally introduce Org mode! I am so **excited**.

Org mode replaces a word processor, a presentation creator, and a spreadsheet editor. The spreadsheet ability captures more than 80% use cases wherein one wishes to include a table in a text document destined for physical publication. (It is clear that Excel spreadsheets are *not* destined for physical publication — simply attempt to print an Excel spreadsheet with the default settings.) In my opinion, Org mode matches all *useful* features of the Microsoft Office suite 1-to-1.

What follows are customizations designed to make Org mode behave more like Microsoft Word. The end goal is, once again, to draw as many new users to Emacs as possible!

Check out how much information Org mode keeps concerning the most recent header:

```
(save-excursion
  (org-previous-visible-heading 1)
  (org-entry-properties))
```

(This block was evaluated on Microsoft Windows.)

```
(("CATEGORY" . "smart-documents")
 ("BLOCKED" . "")
 ("FILE" . "c:/Users/blend/AppData/Roaming/.emacs.d/smart-documents.org")
 ("PRIORITY" . "A")
 ("ITEM" . "Introduction"))
```

5.2 Basic customization

5.2.1 Base folder

Org base directory is in user home on GNU/Linux, or in `AppData` in MS Windows.

```
(setq org-directory (concat user-emacs-directory "~/org"))
```

5.2.2 Prevent/warn on invisible edits

```
(setq org-catch-invisible-edits t)
```

5.3 Org cosmetics

First, we ensure the display of markup symbols for **bold**, *italic*, underlined and ~~strikethrough~~ text, and ensure our document appears indented upon loading.[8]

We then set values for many other Org-related cosmetic symbols.

```
(setq org-hide-emphasis-markers nil
      org-startup-indented t
      org-src-preserve-indentation nil
      org-edit-src-content-indentation 2
      org-ellipsis (propertize " ⤵ " ; folding symbol
       'mouse-face 'highlight
       'help-echo "Unfold section."))
```

5.3.1 Pretty LaTeX symbols

We display LaTeX entities as UTF8 symbols ⇒ this is a slick idea to further make Emacs look like the exported PDF. Using symbols in tables is discouraged?

```
(setq org-pretty-entities t)
```

5.3.2 Dynamic numbering of headlines

We enable the dynamic numbering of headlines in an Org buffer. We also set the numbering face to `org-special-keyword`, which specifies a `:background white` attribute. This is necessary because otherwise, the background of the numbering may be overridden by the `TODO` face attribute `:background coral`.

```
(add-hook 'org-mode-hook 'org-num-mode)
(setq org-num-face 'org-special-keyword)
```

By default, we hide Org document properties such as `#+TITLE`, `#+AUTHOR`, and `#+DATE`, because those keywords are defined when the document template is populated. We can nevertheless always access those properties and edit them manually, with a simple keyboard shortcut (cf. Section 3.2.11).

[8] It *appears* indented, but the underlying plaintext file does not contain tab characters!

5.3.3 TODO Document properties

```
(defun org-property-value (property)
  "Return the value of a given Org document property."
  (interactive)
  (save-excursion
    (goto-char (point-min))
    (re-search-forward
     (concat
      "^[[:space:]]*#\\+"
      property
      ":[[:space:]]*\\(.*?\\)[[:space:]]*$")
     nil t)
    (nth 3 (car (cdr (org-element-at-point))))))

(defun sd-document-properties ()
  "Open separate buffer to edit Org mode properties."
  (interactive)
  (let ((title (car (org-property-value "TITLE")))
(date (org-property-value "DATE")))
    (with-output-to-temp-buffer "Smart Document Properties"
      (print title)
      (print date))))

(add-hook 'org-src-mode-hook
  '(lambda ()
     "Disable flycheck for `emacs-lisp-mode'."
     (setq-local flycheck-disabled-checkers
 '(emacs-lisp-checkdoc))))
```

5.3.4 Timestamps

More literary timestamps are exported to LaTeX using the following custom format:

```
(setq org-time-stamp-custom-formats
      '("%d %b. %Y (%a)" . "%d %b. %Y (%a), at %H:%M"))
```

5.3.5 Sequence of TODOs

```
(setq org-todo-keywords
      '((sequence "TODO" ; Vanilla sequence
  "|"
  "DONE")
(sequence "APPLY" ; Job applications
  "FOLLOW UP"
```

```
  "|"
  "REJECTED"
  "STOP"
  "OFFER")
(sequence "STUCK" ; Project mgmt
  "WAITING"
  "|"
  "N/A"
  "COMPLETED")))



(setq org-todo-keyword-faces
      '(("STUCK" . (:height 1.6 :background "red" :foreground "white" :weight bol
("WAITING" . (:height 1.6 :background "yellow"))
("N/A" . (:height 1.6 :background "LightSteelBlue3" :foreground "white"))
("COMPLETED" . (:height 1.6 :background "green" :foreground "white"))))
```

5.4 Agenda

The agenda displays a chronological list of headings across all agenda files for which the heading or body contain a matching `org-time-stamp`.[9]

5.4.1 Diary file

The diary file can be included in all agenda views.

```
(setq org-agenda-diary-file "~/org/PERSONAL/diary/diary.org")
```

5.4.2 List of agenda files

The list of agenda files is saved at the following location.

```
(setq org-agenda-files (concat sd-path-meta "org-agenda-files"))
```

5.5 Org Capture

```
(setq org-default-notes-file (concat sd-path-resources "org/default-
notes.org"))
```

[9] An `org-time-stamp` can be inserted with `C-c .` (period)

5.6 Programming a *Smart Document*

The following languages can be used inside SRC blocks, in view of being executed by the Org Babel backend upon document export.

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((shell . t)
   (python . t)
   (ditaa . t)
   (plantuml . t)
   (emacs-lisp . t)
   (awk . t)
   (ledger . t)
   (latex . t)
   (C . t)
   (gnuplot . t)))
```

5.7 Exporting *Smart Documents*

5.7.1 LaTeX export

We'll be compiling our documents with LuaTeX. This will afford us some future-proofing, since it was designated as the successor to pdfTeX by the latter's creators.

First, we define the command executed when an Org file is exported to LaTeX. We'll use `latexmk`, the Perl script which automagically runs binaries related to LaTeX in the correct order and the right amount of times.

Options and why we need them:

`-shell-excape` required by minted to color source blocks

`-pdflatex=lualatex` we use lualatex to generate our PDF

`-interaction=nonstopmode` go as far as possible without prompting user for input

```
(setq org-latex-pdf-process
      '("latexmk -pdf -f \
    -pdflatex=lualatex -shell-escape \
    -interaction=nonstopmode -outdir=%o %f"))
```

1. Exporting timestamps

   We customize the format for org time stamps to make them appear monospaced in our exported LaTeX documents. This makes it easy to distinguish time stamps from body text, and make them align nicely in definition lists, which I prefer when logging events:

   `[2021-10-03 Sun]` Did something

[2021-10-04 Mon] Did something else

[2021-10-05 Tue] Did yet another thing

```
(setq org-latex-active-timestamp-format
      "\\texttt{%s}")
(setq org-latex-inactive-timestamp-format
      "\\texttt{%s}")
```

2. LaTeX packages

   The following packages are loaded for every time we export to LaTeX.

```
(setq org-latex-packages-alist
      '(("AUTO" "babel" t
 ("pdflatex"))
("AUTO" "polyglossia" t ; Babel replacement for LuaLaTeX
 ("xelatex" "lualatex"))
("" "fontspec" t ; Fonts for LuaLaTeX
 ("lualatex"))
("" "booktabs" t ; Publication quality tables
 ("pdflatex" "lualatex"))
("" "wasysym" t ; Emojis and other symbols
 ("pdflatex" "lualatex"))
("" "lettrine" t
 ("pdflatex" "lualatex"))
("table,svgnames" "xcolor" t ; svgnames opens up ~150 new color keywords
 ("pdflatex" "lualatex"))
("skip=0.5\\baselineskip" "caption" t ; Increase space between floats and ca
 ("pdflatex" "lualatex"))))
```

3. Cleaning directory after export

   Now, we set the files to be deleted when a LaTeX → PDF compilation occurs. We only care about two files, in the end: the Org mode file for edition, and the PDF for distribution.

```
(setq org-latex-logfiles-extensions
      '("aux" "bcf" "blg" "fdb_latexmk"
"fls" "figlist" "idx" "log" "nav"
"out" "ptc" "run.xml" "snm" "toc" "vrb" "xdv"
"tex" "lot" "lof"))
```

4. Chronological diary entries

   By default, Org agenda inserts diary entries as the first under the selected date. It is preferable to insert entries in the order that they were recorded, i.e. chronologically.

```
(setq org-agenda-insert-diary-strategy 'date-tree-last)
```

5. Extra LaTeX class

   This *letter* template completes the other default LaTeX classes.

```
(require 'ox-publish)
(add-to-list 'org-latex-classes
     '("letter"
       "\\documentclass[11pt]{letter}"
       ("\\section{%s}" . "\\section*{%s}")
       ("\\subsection*{%s}" . "\\subsection*{%s}")
       ("\\subsubsection*{%s}" . "\\subsubsection*{%s}")))

(add-to-list 'org-latex-classes
     '("book-blendoit"
       "\\documentclass[11pt]{book}"
       ("\\chapter{%s}" . "\\chapter*{%s}")
       ("\\section{%s}" . "\\section*{%s}")
       ("\\subsection*{%s}" . "\\subsection*{%s}")
       ("\\subsubsection*{%s}" . "\\subsubsection*{%s}")))
```

6. AUCTEX

```
(use-package tex
  :defer t
  :ensure auctex
  :ensure auctex-latexmk)
(auctex-latexmk-setup)
```

7. Groff export

```
(require 'ox-groff)
```

5.8 TODO Org links

This is a mind-bending capacity of Org mode: we can assign arbitrary functions to be executed when a user follows an Org link. Org links appear as hyperlinks both in buffers and PDF exports — e.g. the following link to this very section, Section 5.8 — but their in-buffer behavior can be arbitrarily assigned.

```
(org-add-link-type
 "tag" 'endless/follow-tag-link)
```

```
(defun endless/follow-tag-link (tag)
  "Display a list of TODO headlines with tag TAG.
With prefix argument, also display headlines without a TODO keyword."
  (org-tags-view (null current-prefix-arg) tag))

[[tag:work+phonenumber-boss][Optional Description]]
```

6 One-click workflows

In this section, we'll implement useful one-click workflows. It comes later than the keybinding definitions for two reasons:

1. To a new user, keybindings are more relevant than the implementation of the bound function — it is more important to know how to drive a car than how a car works.
2. If the following subsections share the same name as the keybinding subsection (Section 3), the links to that name will resolve to the earliest heading in the document, i.e. the keybinding subsection, and not the subsection describing the "one-click workflow".

6.1 Opening files

First off, we identify files that we'd like to jump to conveniently.

6.1.1 This very file

We begin by defining a function to open this very file.

```
(defun sd-find-literate-config ()
  "Visit this very file."
  (interactive)
  (find-file sd-literate-config))
```

6.1.2 Org diary file

```
(defun sd-find-org-diary()
  "Visit the `org-agenda-diary-file'."
  (interactive)
  (find-file org-agenda-diary-file))
```

6.2 TODO Export to PDF

This series of `quick-export` functions have one objective: harmonize the export of Emacs buffers to PDF. Org mode does this by design; we describe additional exports for other modes, most notably Nroff mode and Ledger mode.

6.2.1 From Org mode

This reimplements the most common Org mode export: Org → LaTeX → PDF. The binding is defined in Section 3.7.1.

```
(defun sd-quick-export--org ()
  "Org mode async export to PDF and open.
     This basically reimplements `C-c C-e C-a l o'."
  (org-open-file (org-latex-export-to-pdf)))
```

6.2.2 From a Ledger report

```
(defun sd-quick-export--ledger-report ()
  "Quick export for `ledger-mode' report buffers."
  (let ((old-buffer (current-buffer)))
    (with-output-to-temp-buffer "**SD Export**"
      (print "#+SETUPFILE: ~/.emacs.d/resources/templates/documents/default.org")
      (newline)
      (insert-buffer-substring old-buffer)
      (forward-line 10)
      (org-table-convert-region (point) (goto-char (point-max)))
      (setq more-lines-p t)
      (while more-lines-p
(move-end-of-line 1)
(newline)
(setq more-lines-p (= 0 (forward-line 1))))
      (org-open-file (org-latex-export-to-pdf)))))
```

6.2.3 From Nroff mode

```
(defun sd-quick-export--nroff (macros)
  "Export Nroff/Groff buffer to PDF, with specified macro set."
  (let* ((file-exported-name
  (concat (file-name-sans-extension buffer-file-name)
  (format "-%s.pdf" macros)))
 (command-export
  (format "groff -%s -Tps %s | ps2pdf - > %s"
  macros
  (buffer-file-name)
  file-exported-name)))
    (shell-command command-export)
    (org-open-file file-exported-name)))
```

6.2.4 Quick export

```
(defun sd-quick-export ()
  "Quickly prettify and export current buffer to PDF."
  (interactive)
  (cond ((eq major-mode 'org-mode)
 (sd-quick-export--org))
((eq major-mode 'nroff-mode)
 (sd-quick-export--nroff
  (read-string "Macro set used (ms, me, mm...): ")))
((eq major-mode 'emacs-lisp-mode)
 (message "No quick-export implemented for emacs-lisp-mode yet."))
((eq major-mode 'ledger-report-mode)
 (sd-quick-export--ledger-report))
(t (message (format "No sd-quick-export backend for %s."
    major-mode)))))
```

6.3 Operate on whole buffer

6.3.1 Fix indentation

```
(defun sd-indent-buffer ()
  "Indent entire buffer."
  (interactive)
  (save-excursion
    (indent-region (point-min) (point-max) nil)))
```

6.3.2 Beautify

1. All types of buffers

   ```
   (defun sd-beautify-buffer ()
     "Clean up buffer in the most general sense.

   This means performing the following actions:
   1) indenting the buffer according to the major mode in force,
   2) deleting trailing whitespaces.

   As well as a couple other things."
     (interactive)
     (sd-indent-buffer)
     (delete-trailing-whitespace)
     (when (string-equal
    major-mode "org-mode")
       (sd-org-fix-headlines-spacing)))
   ```

6.4 Smart quitting

6.4.1 Window

```
(defun sd-delete-window-or-previous-buffer ()
  "Delete window; if sole window, previous buffer."
  (interactive)
  (if (> (length (window-list)) 1)
      (delete-window)
    (previous-buffer)))
```

6.4.2 Frame

```
(defun sd-delete-frame-or-kill-emacs ()
  (interactive)
  "Delete frame; if sole frame, kill Emacs."
  (if (> (length (frame-list)) 1)
      (delete-frame)
    (save-buffers-kill-terminal)))
```

7 Editing preferences

These customizations enhance editor usability. They also encompass cosmetic changes not brought about a specific package.

7.1 Editor

7.1.1 Coding standards

This is just a better default. Don't @ me.

```
(setq c-default-style "linux"
      c-basic-offset 4)
```

7.1.2 Recent files

The keybinding for opening a recently visited file is described in paragraph 3.2.7.

```
(recentf-mode 1)
(setq recentf-max-menu-items 100)
(setq recentf-max-saved-items 100)
(run-at-time nil (* 5 60) 'recentf-save-list)
```

7.1.3 Reload changed files silently

```
(global-auto-revert-mode)
```

7.2 Frame

7.2.1 Header & mode lines

1. **TODO** Icons We start by defining some icons we wish to include in our user interface. Emacs allows the usage of GIF images — this paves the way for UI elements which may be animated.

   ```
   (defcustom sd-icon-loading
     (create-image
      (concat user-emacs-directory "resources/images/icons/ellipsis.gif")
      'gif nil
      :scale 0.4)
     "The GIF representing \"loading\". Not animated by default."
     :type 'sexp
     :version "27.1"
     :group 'sd)

   (defun sd-icon-loading ()
     "Insert an animated blue ellipsis."
     (insert-image sd-icon-loading)
     (image-animate sd-icon-loading 0 t))
   ```

2. Header line

   In Org mode, the document header line will be the title of the document we are working on currently. We start by defining keybindings for our header line buttons for navigating through open windows.

   ```
   (defvar sd-header-line-previous-buffer-keymap
     (let ((map (make-sparse-keymap)))
       (define-key map [header-line mouse-1] 'previous-buffer)
       map)
     "Keymap for what is displayed in the header line, with a single
     window.")

   (defvar sd-header-line-kill-buffer-keymap
     (let ((map (make-sparse-keymap)))
       (define-key map [header-line mouse-1] 'kill-buffer-and-window)
       map)
     "Keymap for closing current window.")
   ```

```
(defvar sd-header-line-maximize-window-keymap
  (let ((map (make-sparse-keymap)))
    (define-key map [header-line mouse-1] 'delete-other-windows)
    map)
  "Keymap for maximizing the current window.")

(defvar sd-header-line-minimize-window-keymap
  (let ((map (make-sparse-keymap)))
    (define-key map [header-line mouse-1] 'delete-window)
    map)
  "Keymap for minimizing the current window.")
```

Now, we describe the actual format of the header line.

```
(use-package all-the-icons)

  (setq-default
   header-line-format
   '(:eval
     (list
      (if (eq (length (window-list)) 1)
  (propertize " 🗙 "
      'face 'org-meta-line
      'mouse-face 'highlight
      'keymap sd-header-line-previous-buffer-keymap
      'help-echo "Return to previous window.")
(list (propertize " 🗙 "
  'face 'org-meta-line
  'mouse-face 'org-todo
  'keymap sd-header-line-kill-buffer-keymap
  'help-echo "Close this window.")
      (propertize " 🗙"
  'face 'org-meta-line
  'mouse-face 'highlight
  'keymap sd-header-line-maximize-window-keymap
  'help-echo "Maximize this window.")
      (propertize "🗙 "
  'face 'org-meta-line
  'mouse-face 'highlight
  'keymap sd-header-line-minimize-window-keymap
  'help-echo "Minimize this window.")))
      mode-line-buffer-identification)))
```

```
  (image-animate sd-icon-loading 0 t)
```

3. Mode line

   This interpretation of the ideal mode line is the result of carefully studying the default `mode-line`, as well as studying various customizations online.

```
(defvar sd-mode-line-lock-buffer-keymap
  (let ((map (make-sparse-keymap)))
    (define-key map [mode-line mouse-1] 'read-only-mode)
    map)
  "Keymap for locking/unlocking the current buffer.")

(setq-default
 mode-line-format
 (list
  mode-line-front-space
  '(:eval (if buffer-read-only
      (propertize "
  'keymap sd-mode-line-lock-buffer-keymap
  'help-echo "C-x C-q: unlock buffer.")
    (propertize "
'keymap sd-mode-line-lock-buffer-keymap
'help-echo "C-x C-q: lock buffer.")))
  '(:eval (if (buffer-modified-p)
      (propertize "  "
  'help-echo "Buffer is modified.")
    (propertize "  "
'help-echo "Buffer is saved.")))
  mode-line-modes " "
  mode-line-end-spaces))
```

7.3 Window

7.4 Buffer

7.4.1 Save cursor location

Save cursor location in visited buffer after closing it or Emacs.

```
(save-place-mode 1)
```

7.4.2 Column filling

We leave the default `fill-column` unchanged, so as to minimally disrupt a user's existing documents. We automatically break lines longer than `fill-column`.

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

7.5 Text

7.5.1 Beautiful symbols

We want the Emacs Lisp keyword `lambda` to be rendered as λ within the editor. This is mostly for a subjective "cool" factor.

```
(global-prettify-symbols-mode 1)
```

7.5.2 Org mode sugar

Let's pimp out the appearance of our text in Org mode. First, we prettify checkbox lists when viewed on GNU/Linux systems.

```
(when (string-equal system-type "gnu/linux")
  (add-hook 'org-mode-hook
    (lambda ()
      "Beautify Org symbols."
      (push '("[ ]" .  "☐") ; Unchecked item
    prettify-symbols-alist)
      (push '("[X]" . "☑" ) ; Checked item
    prettify-symbols-alist)
      (push '("[-]" . "☒" ) ; Partially checked item
    prettify-symbols-alist)
      (push '("-" . "•" ) ; Plain list dash
    prettify-symbols-alist)
      (prettify-symbols-mode))))
```

- ☐ This first item is unticked
- ⊟ This second item is partially completed
  - ☒ This first sub-item is ticked
  - ☐ This sub-item is not ticked
- ☐ This third item is ticked

7.5.3 Electric modes

Electricity is a very important technology. In Emacs jargon, “electric” modes tend to automate behaviors or present some elegant simplification to a workflow.[10]

```
(electric-pair-mode) ; Certain character pairs are automatically completed.
(electric-indent-mode) ; Newlines are always intelligently indented.
```

7.6 Minibuffer

We replace the longer `yes-or-no-p` questions with more convenient `y-or-n-p`.

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

Disable minibuffer scroll bar.

```
(set-window-scroll-bars (minibuffer-window) nil nil)
```

8 Themes

Without a carefully designed theme, our editor would become unusable. Thus, we *describe* two themes that were developed **purposefully** and iteratively.

```
(load-theme 'sd-compagnon-dark)

(load-theme 'molokai)

(load-theme 'sd-light)

(load-theme 'sd-dark)
```

8.1 My light and dark themes

A highly legible, unambiguous, and classic theme.

8.1.1 Colors

The default face is a black foreground on a white background, this matches MS Word. We are striving for a simple, intuitive color scheme.

Most of the affordance cues derived from color are identical in both light and dark themes (Table 1).

1. Red
2. Green

[10] More information can be found at `https://www.emacswiki.org/emacs/Electricity`.

Table 1: Light and dark themes' colors.

Color	`sd-light`	`sd-dark`
Black	default text	default background
Lighter shades	lesser headers	*n/a*
White	default background	default text
Darker shades	*n/a*	lesser headers
Red	negative	*same*
Tomato	timestamp 'TODO'	*same*
Green	positive	*same*
ForestGreen	timestamp 'DONE'	*same*
Blue	interactive content; links	*same*
SteelBlue	anything Org mode; anchor color	*same*
DeepSkyBlue	`highlight`	*same*
DodgerBlue	`isearch`	*same*
Purple	Code syntax highlighting	*same*

3. Blue
4. Purple

   Purple is Emacs' main logo color. Since we use Emacs for coding a lot, code syntax highlighting to could be in the pink/purple shades.

8.1.2 Cursors

In order to imitate other modern text editors, we resort to a blinking bar cursor. We choose red, the most captivating color, because the cursor is arguably the region on our screen:

1. most often looked at;
2. most often searched when lost.

In files containing only `fixed-pitch` fonts (i.e. files containing only code), the cursor becomes a high-visibility box.

In files containing a mix of `variable-pitch` and `fixed-pitch` fonts, the cursor is a more MS Word-like bar.

```
(setq-default cursor-type 'box)
```

8.1.3 TODO Fonts

Here are some fonts I discovered and enjoyed since I began learning Emacs.

1. Serif

Crimson Pro[11] `variable-pitch`, default body text font

- Inspired by Garamond

Linux Libertine[12] `variable-pitch`, default body text font

- Inspired by Garamond

2. Sans serif

Public Sans[13] `variable-pitch`, default body text font

- Very modern yet neutral
- Designed for the U.S. government
- Exceptional color on screen

Jost[14] `org-document-title` and `org-level-1`

- Ultra-modern
- Tasteful amount of geometric inspiration

Open Sans[15] `variable-pitch`

- Ooh geometric Bauhaus influences, look at me
- Tall leading height is `harmonious`

Liberation Sans[16] ~~`variable-pitch`~~

- Metrically compatible with *Arial* (ugh)
- Unoffensive, unambitious forms
- Pretty angular letters, it's like you're trying to read squares

3. Monospace

Hack[17] `default` and `fixed-pitch`, default code font

- Legible, modern monospace font
- Strict, sharp, uncompromising

Hermit[18] `org-block`, anything Org/meta in general

- Slightly wider than Hack
- More opinionated shapes
- Very legible parentheses, very useful for Emacs Lisp!

[11]

[12]

[13] `https://public-sans.digital.gov/`
[14] `https://indestructibletype.com/Jost.html`
[15] `https://www.opensans.com/`
[16] `https://en.wikipedia.org/wiki/Liberation_fonts`
[17] `https://sourcefoundry.org/hack/`
[18] `https://pcaro.es/p/hermit/`

Courier Prime[19] monospace in print

4. Using proportional fonts when needed

   We use `variable-pitch-mode` for appropriate modes.

   ```
   (add-hook 'org-mode-hook 'variable-pitch-mode)
   (add-hook 'info-mode-hook 'variable-pitch-mode)
   ```

5. **TODO** Default font size

   Make default font size larger on displays of which the resolution is greater than 1920×1080.

   ```
   (if (< screen-width 1920)
       (default-font)
     else)
   ```

8.2 TODO *Wealthy* document theme

Figure 1: Claude Garamont, an icon of font design. World-renowned for his elegant typefaces, which inspired many generations of typographers.

Good golly, nobody wishes for a *pedestrian* theme! Let your entourage know that you're rocking an editor fit for a king with this finely crafted `wealthy' theme. Selecting it shall enable the following fancitudes:

[19] `https://quoteunquoteapps.com/courierprime/index.php`

1. The default font shall be sublimed in the form of *EB Garamond*
2. Bullets will be tastefully replaced with pointing fingers
3. Heading stars will be replaced with Black Queen chess pieces

> CLAUDE Garamont (c. 1510--1561), known commonly as **Claude Garamond**, was a French type designer, publisher and punch-cutter based in Paris. Garamond worked as an engraver of punches, the masters used to stamp matrices, the moulds used to cast metal type. He worked in the tradition now called old-style serif design, which produced letters with a relatively organic structure resembling handwriting with a pen but with a slightly more structured and upright design. Considered one of the leading type designers of all time, he is recognised to this day for the elegance of his typefaces. Many old-style serif typefaces are collectively known as Garamond, named after the designer.
>
> From `https://en.wikipedia.org/wiki/Claude_Garamond`

8.2.1 Symbol substitution

```
(defun sd-wealthy ()
  "Beautify symbols for our wealthy theme."
  (push '("-" . "□" ) prettify-symbols-alist) ; unnumbered bullets
  (push '("*" . "□" ) prettify-symbols-alist) ; headings
  (prettify-symbols-mode))
```

8.3 TODO `minimal`

9 Late setup

At this point, our editor is almost ready to run. Phew! All that's left to do is to interrupt our profiling activities, and smartly store the result of our profiling.

9.1 Profiling — stop

```
(profiler-stop)
```

9.2 Profiling — report

```
(profiler-report)
```

10 Conclusion

In this configuration file, we described a series of customization steps taken to make Emacs more palatable to modern word processors users.

Local files variables

If the following variable is set to nil, `org-babel` will not ask to confirm the evaluation of source code blocks during export or tangling of this very file.

```
Local Variables:
org-confirm-babel-evaluate: t
End:
```