The Clojure Style Guide

Table of Contents

Preamble

> Role models are important.
>
> -- Officer Alex J. Murphy / RoboCop

This Clojure style guide recommends best practices so that real-world Clojure programmers can write code that can be maintained by other real-world Clojure programmers. A style guide that reflects real-world usage gets used, and a style guide that holds to an ideal that has been rejected by the people it is supposed to help risks not getting used at all — no matter how good it is.

The guide is separated into several sections of related rules. I've tried to add the rationale behind the rules (if it's omitted, I've assumed that it's pretty obvious).

I didn't come up with all the rules out of nowhere; they are mostly based on my extensive career as a professional software engineer, feedback and suggestions from members of the Clojure community, and various highly regarded Clojure programming resources, such as "Clojure Programming" and "The Joy of Clojure".

The guide is still a work in progress; some sections are missing, others are incomplete, some rules are lacking examples, some rules don't have examples that illustrate them clearly enough. In due time these issues will be addressed — just keep them in mind for now.

Please note, that the Clojure developing community maintains a list of coding standards for libraries, too.

Builds

You may build document versions of this guide using various `make` targets.

Dependencies

In order use these, your system will need to have various dependencies installed. You may do that with the following:

```
$ make ubuntu-deps
$ make python-deps
```

Right now the Makefile installs dependencies for Ubuntu only. If you are using a different OS, viewing the dependency targets in the Makefile will provide you with the clues to either install them for your OS or submit a patch so that your OS is supported.

Supported Formats

You can generate copies of this document in PDF, HTML, EPUB, or MOBI formats using the following:

```
$ make html
$ make pdf
$ make epub
$ make mobi
```

Optionally, you may generate all of them with either `make` or `make all`.

Translations

Translations of the guide are available in the following languages:

- Japanese
- Korean

Source Code Organization

> Nearly everybody is convinced that every style but their own is ugly and unreadable. Leave out the "but their own" and they're probably right...
>
> -- Jerry Coffin (on indentation)

- Use **spaces** for indentation. No hard tabs.
- Use 2 spaces to indent the bodies of forms that have body parameters. This covers all `def` forms, special forms and macros that introduce local bindings (e.g. `loop`, `let`, `when-let`) and many macros like `when`, `cond`, `as->`, `cond->`, `case`, `with-*`, etc.

```
;; good
(when something
  (something-else))

(with-out-str
  (println "Hello, ")
  (println "world!"))

;; bad - four spaces
(when something
    (something-else))

;; bad - one space
(with-out-str
 (println "Hello, ")
 (println "world!"))
```

- Vertically align function (macro) arguments spanning multiple lines.

```
;; good
(filter even?
        (range 1 10))

;; bad
(filter even?
  (range 1 10))
```

- Use a single space indentation for function (macro) arguments when there are no arguments on the same line as the function name.

```
;; good
(filter
 even?
 (range 1 10))

(or
 ala
 bala
 portokala)

;; bad - two-space indent
(filter
  even?
  (range 1 10))

(or
  ala
  bala
  portokala)
```

- Vertically align `let` bindings and map keywords.

```
;; good
(let [thing1 "some stuff"
      thing2 "other stuff"]
  {:thing1 thing1
   :thing2 thing2})

;; bad
(let [thing1 "some stuff"
  thing2 "other stuff"]
  {:thing1 thing1
  :thing2 thing2})
```

- Optionally omit the new line between the function name and argument vector for `defn` when there is no docstring.

```
;; good
(defn foo
  [x]
  (bar x))

;; good
(defn foo [x]
  (bar x))

;; bad
(defn foo
  [x] (bar x))
```

- Place the `dispatch-val` of a multimethod on the same line as the function name.

```
;; good
(defmethod foo :bar [x] (baz x))

(defmethod foo :bar
  [x]
  (baz x))

;; bad
(defmethod foo
  :bar
  [x]
  (baz x))

(defmethod foo
  :bar [x]
  (baz x))
```

- When adding a docstring – especially to a function using the above form – take care to correctly place the docstring after the function name, not after the argument vector. The latter is not invalid syntax and won't cause an error, but includes the string as a form in the function body without attaching it to the var as documentation.

```
;; good
(defn foo
  "docstring"
  [x]
  (bar x))

;; bad
(defn foo [x]
  "docstring"
  (bar x))
```

- Optionally omit the new line between the argument vector and a short function body.

```
;; good
(defn foo [x]
  (bar x))

;; good for a small function body
(defn foo [x] (bar x))

;; good for multi-arity functions
(defn foo
  ([x] (bar x))
  ([x y]
   (if (predicate? x)
     (bar x)
     (baz x))))

;; bad
(defn foo
  [x] (if (predicate? x)
        (bar x)
        (baz x)))
```

- Indent each arity form of a function definition vertically aligned with its parameters.

```
;; good
(defn foo
  "I have two arities."
  ([x]
   (foo x 1))
  ([x y]
   (+ x y)))

;; bad - extra indentation
(defn foo
  "I have two arities."
  ([x]
    (foo x 1))
  ([x y]
    (+ x y)))
```

- Sort the arities of a function from fewest to most arguments. The common case of multi-arity functions is that some K arguments fully specifies the function's behavior, and that arities N < K partially apply the K arity, and arities N > K provide a fold of the K arity over varargs.

```
;; good - it's easy to scan for the nth arity
(defn foo
  "I have two arities."
  ([x]
   (foo x 1))
  ([x y]
   (+ x y)))

;; okay - the other arities are applications of the two-arity
(defn foo
  "I have two arities."
  ([x y]
   (+ x y))
  ([x]
   (foo x 1))
  ([x y z & more]
   (reduce foo (foo x (foo y z)) more)))
```

```
;; bad - unordered for no apparent reason
(defn foo
  ([x] 1)
  ([x y z] (foo x (foo y z)))
  ([x y] (+ x y))
  ([w x y z & more] (reduce foo (foo w (foo x (foo y z))) more)))
```

- Indent each line of multi-line docstrings.

```
;; good
(defn foo
  "Hello there. This is
  a multi-line docstring."
  []
  (bar))

;; bad
(defn foo
  "Hello there. This is
a multi-line docstring."
  []
  (bar))
```

- Use Unix-style line endings. (*BSD/Solaris/Linux/OSX users are covered by default, Windows users have to be extra careful.)
  - If you're using Git you might want to add the following configuration setting to protect your project from Windows line endings creeping in:

```
bash$ git config --global core.autocrlf true
```

- If any text precedes an opening bracket(`(` , `{` and `[`) or follows a closing bracket(`)` , `}` and `]`), separate that text from that bracket with a space. Conversely, leave no space after an opening bracket and before following text, or after preceding text and before a closing bracket.

```
;; good
(foo (bar baz) quux)

;; bad
(foo(bar baz)quux)
(foo ( bar baz ) quux)
```

> Syntactic sugar causes semicolon cancer.
>
> -- Alan Perlis

- Don't use commas between the elements of sequential collection literals.

```
;; good
[1 2 3]
(1 2 3)

;; bad
[1, 2, 3]
(1, 2, 3)
```

- Consider enhancing the readability of map literals via judicious use of commas and line breaks.

```
;; good
{:name "Bruce Wayne" :alter-ego "Batman"}

;; good and arguably a bit more readable
{:name "Bruce Wayne"
 :alter-ego "Batman"}

;; good and arguably more compact
{:name "Bruce Wayne", :alter-ego "Batman"}
```

- Place all trailing parentheses on a single line instead of distinct lines.

```
;; good; single line
(when something
  (something-else))

;; bad; distinct lines
(when something
  (something-else)
)
```

- Use empty lines between top-level forms.

```
;; good
(def x ...)

(defn foo ...)

;; bad
(def x ...)
(defn foo ...)
```

An exception to the rule is the grouping of related `def` s together.

```
;; good
(def min-rows 10)
(def max-rows 20)
(def min-cols 15)
(def max-cols 30)
```

- Do not place blank lines in the middle of a function or macro definition. An exception can be made to indicate grouping of pairwise constructs as found in e.g. `let` and `cond`.
- Where feasible, avoid making lines longer than 80 characters.
- Avoid trailing whitespace.
- Use one file per namespace.
- Start every namespace with a comprehensive `ns` form, comprised of `refer` s, `require` s, and `import` s, conventionally in that order.

```
(ns examples.ns
  (:refer-clojure :exclude [next replace remove])
  (:require [clojure.string :as s :refer [blank?]]
            [clojure.set :as set]
            [clojure.java.shell :as sh])
  (:import java.util.Date
           java.text.SimpleDateFormat
           [java.util.concurrent Executors
                                 LinkedBlockingQueue]))
```

- In the `ns` form prefer `:require :as` over `:require :refer` over `:require :refer :all`. Prefer `:require` over `:use`; the latter form should be considered deprecated for new code.

```
;; good
(ns examples.ns
  (:require [clojure.zip :as zip]))

;; good
(ns examples.ns
  (:require [clojure.zip :refer [lefts rights]))

;; acceptable as warranted
(ns examples.ns
  (:require [clojure.zip :refer :all]))

;; bad
(ns examples.ns
  (:use clojure.zip))
```

- Avoid single-segment namespaces.

```clojure
;; good
(ns example.ns)

;; bad
(ns example)
```

- Avoid the use of overly long namespaces (i.e., more than 5 segments).
- Avoid functions longer than 10 LOC (lines of code). Ideally, most functions will be shorter than 5 LOC.
- Avoid parameter lists with more than three or four positional parameters.
- Avoid forward references. They are occasionally necessary, but such occasions are rare in practice.

Syntax

- Avoid the use of namespace-manipulating functions like `require` and `refer`. They are entirely unnecessary outside of a REPL environment.
- Use `declare` to enable forward references when forward references are necessary.
- Prefer higher-order functions like `map` to `loop/recur`.
- Prefer function pre and post conditions to checks inside a function's body.

```clojure
;; good
(defn foo [x]
  {:pre [(pos? x)]}
  (bar x))

;; bad
(defn foo [x]
  (if (pos? x)
    (bar x)
    (throw (IllegalArgumentException. "x must be a positive number!")))
```

- Don't define vars inside functions.

```clojure
;; very bad
(defn foo []
  (def x 5)
  ...)
```

- Don't shadow `clojure.core` names with local bindings.

```clojure
;; bad - you're forced to use clojure.core/map fully qualified inside
(defn foo [map]
  ...)
```

- Use `alter-var-root` instead of `def` to change the value of a var.

```clojure
;; good
(def thing 1) ; value of thing is now 1
; do some stuff with thing
(alter-var-root #'thing (constantly nil)) ; value of thing is now nil

;; bad
(def thing 1)
; do some stuff with thing
(def thing nil)
; value of thing is now nil
```

- Use `seq` as a terminating condition to test whether a sequence is empty (this technique is sometimes called *nil punning*).

```clojure
;; good
(defn print-seq [s]
  (when (seq s)
```

```
    (prn (first s))
    (recur (rest s))))

;; bad
(defn print-seq [s]
  (when-not (empty? s)
    (prn (first s))
    (recur (rest s))))
```

- Prefer `vec` over `into` when you need to convert a sequence into a vector.

```
;; good
(vec some-seq)

;; bad
(into [] some-seq)
```

- Use `when` instead of `(if ... (do ...)`.

```
;; good
(when pred
  (foo)
  (bar))

;; bad
(if pred
  (do
    (foo)
    (bar)))
```

- Use `if-let` instead of `let` + `if`.

```
;; good
(if-let [result (foo x)]
  (something-with result)
  (something-else))

;; bad
(let [result (foo x)]
  (if result
    (something-with result)
    (something-else)))
```

- Use `when-let` instead of `let` + `when`.

```
;; good
(when-let [result (foo x)]
  (do-something-with result)
  (do-something-more-with result))

;; bad
(let [result (foo x)]
  (when result
    (do-something-with result)
    (do-something-more-with result)))
```

- Use `if-not` instead of `(if (not ...) ...)`.

```
;; good
(if-not pred
  (foo))

;; bad
(if (not pred)
  (foo))
```

- Use `when-not` instead of `(when (not ...) ...)`.

```
;; good
```

```
(when-not pred
  (foo)
  (bar))

;; bad
(when (not pred)
  (foo)
  (bar))
```

- Use `when-not` instead of `(if-not ... (do ...)`.

```
;; good
(when-not pred
  (foo)
  (bar))

;; bad
(if-not pred
  (do
    (foo)
    (bar)))
```

- Use `not=` instead of `(not (= ...))`.

```
;; good
(not= foo bar)

;; bad
(not (= foo bar))
```

- Use `printf` instead of `(print (format ...))`.

```
;; good
(printf "Hello, %s!\n" name)

;; ok
(println (format "Hello, %s!" name))
```

- When doing comparisons, keep in mind that Clojure's functions `<`, `>`, etc. accept a variable number of arguments.

```
;; good
(< 5 x 10)

;; bad
(and (> x 5) (< x 10))
```

- Prefer `%` over `%1` in function literals with only one parameter.

```
;; good
#(Math/round %)

;; bad
#(Math/round %1)
```

- Prefer `%1` over `%` in function literals with more than one parameter.

```
;; good
#(Math/pow %1 %2)

;; bad
#(Math/pow % %2)
```

- Don't wrap functions in anonymous functions when you don't need to.

```
;; good
(filter even? (range 1 10))
```

```
;; bad
(filter #(even? %) (range 1 10))
```

- Don't use function literals if the function body will consist of more than one form.

```
;; good
(fn [x]
  (println x)
  (* x 2))

;; bad (you need an explicit do form)
#(do (println %)
     (* % 2))
```

- Favor the use of `complement` versus the use of an anonymous function.

```
;; good
(filter (complement some-pred?) coll)

;; bad
(filter #(not (some-pred? %)) coll)
```

  This rule should obviously be ignored if the complementing predicate exists in the form of a separate function (e.g. `even?` and `odd?`).

- Leverage `comp` when it would yield simpler code.

```
;; Assuming `(:require [clojure.string :as str])`...

;; good
(map #(str/capitalize (str/trim %)) ["top " " test "])

;; better
(map (comp str/capitalize str/trim) ["top " " test "])
```

- Leverage `partial` when it would yield simpler code.

```
;; good
(map #(+ 5 %) (range 1 10))

;; (arguably) better
(map (partial + 5) (range 1 10))
```

- Prefer the use of the threading macros `->` (thread-first) and `->>` (thread-last) to heavy form nesting.

```
;; good
(-> [1 2 3]
    reverse
    (conj 4)
    prn)

;; not as good
(prn (conj (reverse [1 2 3])
           4))

;; good
(->> (range 1 10)
     (filter even?)
     (map (partial * 2)))

;; not as good
(map (partial * 2)
     (filter even? (range 1 10)))
```

- Prefer `..` to `->` when chaining method calls in Java interop.

```
;; good
(-> (System/getProperties) (.get "os.name"))
```

```
;; better
(.. System getProperties (get "os.name"))
```

- Use `:else` as the catch-all test expression in `cond`.

```
;; good
(cond
  (< n 0) "negative"
  (> n 0) "positive"
  :else "zero"))

;; bad
(cond
  (< n 0) "negative"
  (> n 0) "positive"
  true "zero"))
```

- Prefer `condp` instead of `cond` when the predicate & expression don't change.

```
;; good
(cond
  (= x 10) :ten
  (= x 20) :twenty
  (= x 30) :forty
  :else :dunno)

;; much better
(condp = x
  10 :ten
  20 :twenty
  30 :forty
  :dunno)
```

- Prefer `case` instead of `cond` or `condp` when test expressions are compile-time constants.

```
;; good
(cond
  (= x 10) :ten
  (= x 20) :twenty
  (= x 30) :forty
  :else :dunno)

;; better
(condp = x
  10 :ten
  20 :twenty
  30 :forty
  :dunno)

;; best
(case x
  10 :ten
  20 :twenty
  30 :forty
  :dunno)
```

- Use short forms in `cond` and related. If not possible give visual hints for the pairwise grouping with comments or empty lines.

```
;; good
(cond
  (test1) (action1)
  (test2) (action2)
  :else   (default-action))

;; ok-ish
(cond
  ;; test case 1
  (test1)
  (long-function-name-which-requires-a-new-line
    (complicated-sub-form
      (-> 'which-spans multiple-lines)))

  ;; test case 2
```

```
  (test2)
  (another-very-long-function-name
    (yet-another-sub-form
      (-> 'which-spans multiple-lines)))

  :else
  (the-fall-through-default-case
    (which-also-spans 'multiple
                      'lines)))
```

- Use a `set` as a predicate when appropriate.

```
;; good
(remove #{1} [0 1 2 3 4 5])

;; bad
(remove #(= % 1) [0 1 2 3 4 5])

;; good
(count (filter #{\a \e \i \o \u} "mary had a little lamb"))

;; bad
(count (filter #(or (= % \a)
                    (= % \e)
                    (= % \i)
                    (= % \o)
                    (= % \u))
               "mary had a little lamb"))
```

- Use `(inc x)` & `(dec x)` instead of `(+ x 1)` and `(- x 1)`.
- Use `(pos? x)`, `(neg? x)` & `(zero? x)` instead of `(> x 0)`, `(< x 0)` & `(= x 0)`.
- Use `list*` instead of a series of nested `cons` invocations.

```
# good
(list* 1 2 3 [4 5])

# bad
(cons 1 (cons 2 (cons 3 [4 5])))
```

- Use the sugared Java interop forms.

```
;;; object creation
;; good
(java.util.ArrayList. 100)

;; bad
(new java.util.ArrayList 100)

;;; static method invocation
;; good
(Math/pow 2 10)

;; bad
(. Math pow 2 10)

;;; instance method invocation
;; good
(.substring "hello" 1 3)

;; bad
(. "hello" substring 1 3)

;;; static field access
;; good
Integer/MAX_VALUE

;; bad
(. Integer MAX_VALUE)

;;; instance field access
;; good
(.someField some-object)
```

```
;; bad
(. some-object someField)
```

- Use the compact metadata notation for metadata that contains only slots whose keys are keywords and whose value is boolean `true`.

```
;; good
(def ^:private a 5)

;; bad
(def ^{:private true} a 5)
```

- Denote private parts of your code.

```
;; good
(defn- private-fun [] ...)

(def ^:private private-var ...)

;; bad
(defn private-fun [] ...) ; not private at all

(defn ^:private private-fun [] ...) ; overly verbose

(def private-var ...) ; not private at all
```

- To access a private var (e.g. for testing), use the `@#'some.ns/var` form.

- Be careful regarding what exactly do you attach metadata to.

```
;; we attach the metadata to the var referenced by `a`
(def ^:private a {})
(meta a) ;=> nil
(meta #'a) ;=> {:private true}

;; we attach the metadata to the empty hash-map value
(def a ^:private {})
(meta a) ;=> {:private true}
(meta #'a) ;=> nil
```

Naming

> The only real difficulties in programming are cache invalidation and naming things.
>
> -- Phil Karlton

- When naming namespaces favor the following two schemas:
  - `project.module`
  - `organization.project.module`
- Use `lisp-case` in composite namespace segments(e.g. `bruce.project-euler`)
- Use `lisp-case` for function and variable names.

```
;; good
(def some-var ...)
(defn some-fun ...)

;; bad
(def someVar ...)
(defn somefun ...)
(def some_fun ...)
```

- Use `CamelCase` for protocols, records, structs, and types. (Keep acronyms like HTTP, RFC, XML uppercase.)

- The names of predicate methods (methods that return a boolean value) should end in a question mark (e.g., `even?`).

```
;; good
(defn palindrome? ...)

;; bad
(defn palindrome-p ...) ; Common Lisp style
(defn is-palindrome ...) ; Java style
```

- The names of functions/macros that are not safe in STM transactions should end with an exclamation mark (e.g. `reset!`).
- Use `->` instead of `to` in the names of conversion functions.

```
;; good
(defn f->c ...)

;; not so good
(defn f-to-c ...)
```

- Use `*earmuffs*` for things intended for rebinding (ie. are dynamic).

```
;; good
(def ^:dynamic *a* 10)

;; bad
(def ^:dynamic a 10)
```

- Don't use a special notation for constants; everything is assumed a constant unless specified otherwise.
- Use `_` for destructuring targets and formal arguments names whose value will be ignored by the code at hand.

```
;; good
(let [[a b _ c] [1 2 3 4]]
  (println a b c))

(dotimes [_ 3]
  (println "Hello!"))

;; bad
(let [[a b c d] [1 2 3 4]]
  (println a b d))

(dotimes [i 3]
  (println "Hello!"))
```

- Follow `clojure.core`'s example for idiomatic names like `pred` and `coll`.
  - in functions:
    - `f`, `g`, `h` - function input
    - `n` - integer input usually a size
    - `index`, `i` - integer index
    - `x`, `y` - numbers
    - `xs` - sequence
    - `m` - map
    - `s` - string input
    - `re` - regular expression
    - `coll` - a collection
    - `pred` - a predicate closure
    - `& more` - variadic input
    - `xf` - xform, a transducer
  - in macros:
    - `expr` - an expression
    - `body` - a macro body
    - `binding` - a macro binding vector

Collections

> It is better to have 100 functions operate on one data structure
> than to have 10 functions operate on 10 data structures.
>
> -- Alan J. Perlis

- Avoid the use of lists for generic data storage (unless a list is exactly what you need).
- Prefer the use of keywords for hash keys.

```
;; good
{:name "Bruce" :age 30}

;; bad
{"name" "Bruce" "age" 30}
```

- Prefer the use of the literal collection syntax where applicable. However, when defining sets, only use literal syntax when the values are compile-time constants.

```
;; good
[1 2 3]
#{1 2 3}
(hash-set (func1) (func2)) ; values determined at runtime

;; bad
(vector 1 2 3)
(hash-set 1 2 3)
#{(func1) (func2)} ; will throw runtime exception if (func1) = (func2)
```

- Avoid accessing collection members by index whenever possible.
- Prefer the use of keywords as functions for retrieving values from maps, where applicable.

```
(def m {:name "Bruce" :age 30})

;; good
(:name m)

;; more verbose than necessary
(get m :name)

;; bad - susceptible to NullPointerException
(m :name)
```

- Leverage the fact that most collections are functions of their elements.

```
;; good
(filter #{\a \e \o \i \u} "this is a test")

;; bad - too ugly to share
```

- Leverage the fact that keywords can be used as functions of a collection.

```
((juxt :a :b) {:a "ala" :b "bala"})
```

- Avoid the use of transient collections, except for performance-critical portions of the code.
- Avoid the use of Java collections.
- Avoid the use of Java arrays, except for interop scenarios and performance-critical code dealing heavily with primitive types.

Mutation

Refs

- Consider wrapping all I/O calls with the `io!` macro to avoid nasty surprises if you accidentally end up calling such code in a transaction.
- Avoid the use of `ref-set` whenever possible.

```
(def r (ref 0))

;; good
(dosync (alter r + 5))

;; bad
(dosync (ref-set r 5))
```

- Try to keep the size of transactions (the amount of work encapsulated in them) as small as possible.
- Avoid having both short- and long-running transactions interacting with the same Ref.

Agents

- Use `send` only for actions that are CPU bound and don't block on I/O or other threads.
- Use `send-off` for actions that might block, sleep, or otherwise tie up the thread.

Atoms

- Avoid atom updates inside STM transactions.
- Try to use `swap!` rather than `reset!`, where possible.

```
(def a (atom 0))

;; good
(swap! a + 5)

;; not as good
(reset! a 5)
```

Strings

- Prefer string manipulation functions from `clojure.string` over Java interop or rolling your own.

```
;; good
(clojure.string/upper-case "bruce")

;; bad
(.toUpperCase "bruce")
```

Exceptions

- Reuse existing exception types. Idiomatic Clojure code — when it does throw an exception — throws an exception of a standard type (e.g. `java.lang.IllegalArgumentException`, `java.lang.UnsupportedOperationException`, `java.lang.IllegalStateException`, `java.io.IOException`).
- Favor `with-open` over `finally`.

Macros

- Don't write a macro if a function will do.
- Create an example of a macro usage first and the macro afterwards.
- Break complicated macros into smaller functions whenever possible.
- A macro should usually just provide syntactic sugar and the core of the macro should be a plain function. Doing so will improve

composability.

- Prefer syntax-quoted forms over building lists manually.

Comments

> Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?" Improve the code and then document it to make it even clearer.
>
> -- Steve McConnell

- Endeavor to make your code as self-documenting as possible.
- Write heading comments with at least four semicolons.
- Write top-level comments with three semicolons.
- Write comments on a particular fragment of code before that fragment and aligned with it, using two semicolons.
- Write margin comments with one semicolon.
- Always have at least one space between the semicolon and the text that follows it.

```
;;;; Frob Grovel

;;; This section of code has some important implications:
;;;   1. Foo.
;;;   2. Bar.
;;;   3. Baz.

(defn fnord [zarquon]
  ;; If zob, then veeblefitz.
  (quux zot
        mumble             ; Zibblefrotz.
        frotz))
```

- Comments longer than a word begin with a capital letter and use punctuation. Separate sentences with one space.
- Avoid superfluous comments.

```
;; bad
(inc counter) ; increments counter by one
```

- Keep existing comments up-to-date. An outdated comment is worse than no comment at all.
- Prefer the use of the `#_` reader macro over a regular comment when you need to comment out a particular form.

```
;; good
(+ foo #_(bar x) delta)

;; bad
(+ foo
   ;; (bar x)
   delta)
```

> Good code is like a good joke - it needs no explanation.
>
> -- Russ Olsen

- Avoid writing comments to explain bad code. Refactor the code to make it self-explanatory. ("Do, or do not. There is no try." -- Yoda)

Comment Annotations

- Annotations should usually be written on the line immediately above the relevant code.

- The annotation keyword is followed by a colon and a space, then a note describing the problem.
- If multiple lines are required to describe the problem, subsequent lines should be indented as much as the first one.
- Tag the annotation with your initials and a date so its relevance can be easily verified.

```
(defn some-fun
  []
  ;; FIXME: This has crashed occasionally since v1.2.3. It may
  ;;        be related to the BarBazUtil upgrade. (xz 13-1-31)
  (baz))
```

- In cases where the problem is so obvious that any documentation would be redundant, annotations may be left at the end of the offending line with no note. This usage should be the exception and not the rule.

```
(defn bar
  []
  (sleep 100)) ; OPTIMIZE
```

- Use `TODO` to note missing features or functionality that should be added at a later date.
- Use `FIXME` to note broken code that needs to be fixed.
- Use `OPTIMIZE` to note slow or inefficient code that may cause performance problems.
- Use `HACK` to note "code smells" where questionable coding practices were used and should be refactored away.
- Use `REVIEW` to note anything that should be looked at to confirm it is working as intended. For example: `REVIEW: Are we sure this is how the client does X currently?`
- Use other custom annotation keywords if it feels appropriate, but be sure to document them in your project's `README` or similar.

Existential

- Code in a functional way, using mutation only when it makes sense.
- Be consistent. In an ideal world, be consistent with these guidelines.
- Use common sense.

Postamble

Tooling

There are some tools created by the Clojure community that might aid you in your endeavor to write idiomatic Clojure code.

- Slamhound is a tool that will automatically generate proper `ns` declarations from your existing code.
- kibit is a static code analyzer for Clojure which uses core.logic to search for patterns of code for which there might exist a more idiomatic function or macro.

Contributing

Nothing written in this guide is set in stone. It's my desire to work together with everyone interested in Clojure coding style, so that we could ultimately create a resource that will be beneficial to the entire Clojure community.

Feel free to open tickets or send pull requests with improvements. Thanks in advance for your help!

You can also support the style guide with financial contributions via gittip.

License

Spread the Word

A community-driven style guide is of little use to a community that doesn't know about its existence. Tweet about the guide, share it with your friends and colleagues. Every comment, suggestion or opinion we get makes the guide just a little bit better. And we want to have the best possible guide, don't we?

Cheers,

Bozhidar