Table of Contents

SOLID Design Principles In Common Lisp

Learn how to apply SOLID design principles with Common Lisp and the CLOS via defclass.

v0.1

Copyright (C) 2019 Momozor

This book is released under the Creative Commons Attribution 4.0 International License.

If you find any problem, want to suggest an improvement or commit changes to this book, please visit the project repository and open a pull request or an issue at https://github.com/common-lisp-reserve/solid-design-principles-in-common-lisp

Disclaimer

These principles are not rules. Apply them where you think it does make sense. Don't "over-engineer".

We are going to focus on Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation and Dependency Inversion principles where they makes the most sense to apply with Common Lisp as a dynamic typed language.

What is SOLID?

SOLID is an acronym from

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

It is a set of principles made popular by Robert C. Martin intended to make software designs more understandable, flexible and maintainable. Read further at https://en.wikipedia.org/wiki/SOLID.

S: Single Responsibility

A class should have one, and only one, reason to change.

The idea is to make a class only handle a single relevant responsibility (though, this is up to the programmer to decide).

Martin suggests that we define each responsibility of a class as a reason for change. If you can think of more than one motivation for changing a class, it probably has more than one responsibility.

See below for a SRP violation example

```
(defclass status-report-mailer ()
  ((address
    :initarg :address
    :reader get-address)

   (report
    :initarg :report
    :initform ""
    :reader get-report
    :accessor report)))

;; this is okay. mailer should deliver a report.
(defmethod deliver ((status-report-mailer status-report-mailer))
  (format t
          "send email to ~a with email content/body: ~a~%"
          (get-address status-report-mailer)
          (get-report status-report-mailer)))

;; this is incorrect; mailer class should not be responsible
;; for generating a report
(defmethod generate-report ((status-report-mailer status-report-mailer))
  (let ((r (concatenate 'string
                        "status number: "
                        (write-to-string (random 500))
                        ". this is a status report for slow server boot time "
                        "estimating around "
                        (write-to-string (random 200))
                        " seconds from time to fully boot.")))
    (setf (report status-report-mailer) r)))

(defparameter rm1
  (make-instance 'status-report-mailer
                 :address "dummy@email.com"))

;; generated status and boot time may differ from yours
(generate-report rm1)
;; "status number: 361. this is a status report for slow server
;; boot time estimating around 173
;; seconds from time to fully boot."
(deliver rm1)
;; send email to dummy@email.com with email content/body:
;; status number: 361. this is a status report for
;; slow server boot time estimating around
;; 173 seconds from time to fully boot.
```

As you can see above, `status-report-mailer` class is handling both distinct functionalities. Report generation and report delivery. This will force you to modify `status-report-mailer` class if you wish to set up a new value or generate different kind of report template, which it has nothing to do with.

Let's fix this by moving `generate-report` method into its own class.

```
(defclass status-report-mailer ()
  ((address
    :initarg :address
    :reader address)

   (report
    :initarg :report
    :reader report)))

(defclass status-report-generator () nil)

(defmethod deliver ((status-report-mailer status-report-mailer))
  (format t
          "Send email to ~a with content:
           ~a~%"
          (address status-report-mailer)
          (report status-report-mailer)))

(defmethod generate ((status-report-generator status-report-generator))
  (format nil
          "Status Number: ~a
           Boot Time: ~a"
          (random 500)
          (random 200)))

(defparameter *mailer*
  (make-instance 'status-report-mailer
                 :address "admin@email.com"
                 :report (generate (make-instance
                                    'status-report-generator))))

(deliver *mailer*)
```

O: Open/Closed

Objects or entities should be open for extension, but closed for modification.

What this means is that we should write code that doesn't have to be changed every time the requirements changes. For instance, a class should be easily extendable without modifying the class itself.

Take a look at the open/closed principle violation example below.

Bad

```
(defclass circle ()
  ((radius
    :initarg :radius
    :reader get-radius)))

(defclass area-calculator ()
  ((shapes
    :initarg :shapes
    :reader get-shapes)))

(defmethod total-area ((area-calculator area-calculator))
  (reduce #'+
          (mapcar #'(lambda (x)
                      (* pi
                         (get-radius x)
                         (get-radius x)))
                  (get-shapes area-calculator))))

(defparameter *circle-one*
  (make-instance 'area-calculator
                 :shapes
                 (list (make-instance 'circle :radius 5)
                       (make-instance 'circle :radius 6)
                       (make-instance 'circle :radius 2))))

(total-area *circle-one*) ;; 204.20352248333654d0
```

If we do want `total-area` method to calculate a sum of Rectangle areas instead of Circle, we won't be able to do that due to its specific area calculation formula (a = pi * r^2) for a circle area without modifying `total-area` method.

So how can we go over this limit?

Below code shows a better example.

Good

```
(defclass circle ()
  ((radius
    :initarg :radius
    :reader get-radius)))

(defmethod area ((circle circle))
  (* pi (get-radius circle) (get-radius circle)))

(defclass area-calculator ()
  ((shapes
    :initarg :shapes
    :reader get-shapes)))

(defmethod total-area ((area-calculator area-calculator))
  (reduce #'+
          (mapcar #'area
                  (get-shapes area-calculator))))

(defparameter *circle-one*
  (make-instance 'area-calculator
                 :shapes
                 (list (make-instance 'circle :radius 5)
                       (make-instance 'circle :radius 6)
                       (make-instance 'circle :radius 2))))

(total-area *circle-one*) ;; 204.20352248333654d0
```

As you've noticed, we moved the function to calculate circle area into its Circle class. This way, if we want to calculate a Rectangle shape area (or triangle, etc), we only have to create a new class with its own method to handle Rectangle area calculation.

For example, a new Rectangle class and area method which calculates a simple Rectangle shape area (a = w * h)

```
(defclass rectangle ()
  ((width
    :initarg :width
    :reader get-width)

   (height
    :initarg :height
    :reader get-height)))

(defmethod area ((rectangle rectangle))
  (* (get-width rectangle)
     (get-height rectangle)))
```

Full Better Example

```
(defclass circle ()
  ((radius
    :initarg :radius
    :reader get-radius)))

(defmethod area ((circle circle))
  (* pi (get-radius circle) (get-radius circle)))

(defclass rectangle ()
  ((width
    :initarg :width
    :reader get-width)

   (height
    :initarg :height
    :reader get-height)))

(defmethod area ((rectangle rectangle))
  (* (get-width rectangle)
     (get-height rectangle)))

(defclass area-calculator ()
  ((shapes
    :initarg :shapes
    :reader get-shapes)))

(defmethod total-area ((area-calculator area-calculator))
  (reduce #'+
          (mapcar #'area
                  (get-shapes area-calculator))))

(defparameter *circle-one*
  (make-instance 'area-calculator
                 :shapes
                 (list (make-instance 'circle :radius 5)
                       (make-instance 'circle :radius 3)
                       (make-instance 'circle :radius 12))))

(defparameter *rectangle-one*
  (make-instance 'area-calculator
                 :shapes
                 (list (make-instance 'rectangle :height 5 :width 10)
                       (make-instance 'rectangle :height 9 :width 20)
                       (make-instance 'rectangle :height 23 :width 44))))

(total-area *circle-one*) ;; 559.2034923389832d0
(total-area *rectangle-one*) ;; 1242
```

L: Liskov Substitution

Let *Φ(x) be a property provable about objects x of type T*. Then *Φ(y) should be true for objects y of type S where S is a subtype of T.*

in other words: Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

```
(defclass rectangle ()
  ((width
    :initarg :width
    :initform 0
    :reader get-width
    :accessor width)

   (height
    :initarg :height
    :initform 0
    :reader get-height
    :accessor height)))

(defmethod area ((rectangle rectangle))
  (* (get-width rectangle)
     (get-height rectangle)))

(defclass square (rectangle)
  nil)

(defmethod set-width ((square square) w)
  (setf (width square) w)
  (setf (height square) w))

(defmethod set-height ((square square) h)
  (setf (height square) h)
  (setf (width square) h))

(defparameter square-area (make-instance 'square))

(set-width square-area 5)
(set-height square-area 10)
(area square-area) ;; 100 instead of 50
```

I: Interface Segregation

Clients should not be forced to depend upon interfaces that they do not use.

Because Common Lisp in particular doesn't have interface (but we got ***macros*** to properly add one, but let's put that aside for the advanced)

similar to static-typed language like Java or C#, and due to the Lisp's ability to do multiple inheritance, this principle carry little importance.

But, we'll try simulate this with `defgeneric` just for fun.

So, what is all the fuss about Interface Segregation?

Basically, you don't have to implement and to depend on methods that are irrelevant for the client (eg. a class).

Let's see why this is bad, below.

Bad

```
(defclass bird ()
  nil)

(defgeneric b-eat (bird))
(defgeneric b-sleep (bird))
(defgeneric b-fly (bird))
(defgeneric b-run (bird))

(defclass parrot (bird)
  nil)

(defmethod b-eat ((parrot parrot))
  (format t "~a~%" "the parrot eats"))

(defmethod b-sleep ((parrot parrot))
  (format t "~a~%" "the parrot sleeps"))

(defmethod b-fly ((parrot parrot))
  (format t "~a~%" "the parrot flies"))

;; in some other language, you are forced to implement this
;; even if it does not make sense for this parrot class.
(defmethod b-run ((parrot parrot))
  (format t
          "~a~%"
          "this is wrong! a parrot cannot really run! :("))

(defclass penguin (bird)
  nil)
```

```
(defmethod b-eat ((penguin penguin))
  (format t "~a~%" "the penguin eats"))

(defmethod b-sleep ((penguin penguin))
  (format t "~a~%" "the penguin sleeps"))

(defmethod b-run ((penguin penguin))
  (format t "~a~%" "the penguin runs"))

;; in some other language, you are forced to implement this
;; even if it does not make sense for this penguin class.
;; since when a penguin can fly, huh? This is not a Puffin bird.
(defmethod b-fly ((penguin penguin))
  (format t "~a~%" "this is wrong. a penguin cannot fly! :("))


(defparameter clawy (make-instance 'parrot))
(defparameter pingu (make-instance 'penguin))

(b-eat clawy) ;; the parrot eats
(b-sleep clawy) ;; the parrot sleeps
(b-run clawy) ;; this is wrong!. a parrot cannot really run! :(
(b-fly clawy) ;; the parrot flies

(b-eat pingu) ;; the penguin eats
(b-sleep pingu) ;; the penguin sleeps
(b-run pingu) ;; the penguin runs
(b-fly pingu) ;; this is wrong. a penguin cannot fly! :(
```

A Penguin can't fly. A Parrot can't (let's say) run. These classes doesn't have to depend on `b-fly` and `b-run` methods where they doesn't make sense. You see, both Penguin and Parrot `is a` Bird, both also can eat and sleep, but they don't behave the same way and both differs in abilities and weaknesses (one can fly, but the other doesn't). These extra useless code can lead to redundancy.

Let's make it better, shall we?

Good

```
;; interface
(defclass bird ()
  nil)

;; interface body (mandatory methods to be implemented)
(defgeneric b-eat (bird))
(defgeneric b-sleep (bird))

(defclass flightless-bird (bird)
  nil)

(defgeneric b-run (flightless-bird))

(defclass flying-bird (bird)
  nil)

(defgeneric b-fly (flying-bird))

(defclass parrot (flying-bird)
  nil)

(defmethod b-eat ((parrot parrot))
  (format t "~a~%" "the parrot eats"))

(defmethod b-sleep ((parrot parrot))
  (format t "~a~%" "the parrot sleeps"))

(defmethod b-fly ((parrot parrot))
  (format t "~a~%" "the parrot flies"))

(defclass penguin (flightless-bird)
  nil)

(defmethod b-eat ((penguin penguin))
  (format t "~a~%" "the penguin eats"))

(defmethod b-sleep ((penguin penguin))
  (format t "~a~%" "the penguin sleeps"))

(defmethod b-run ((penguin penguin))
  (format t "~a~%" "the penguin runs"))

(defparameter clawy (make-instance 'parrot))
(defparameter pingu (make-instance 'penguin))

(b-eat clawy) ;; the parrot eats
(b-sleep clawy) ;; the parrot sleeps
(b-fly clawy) ;; the parrot flies

(b-eat pingu) ;; the penguin eats
(b-sleep pingu) ;; the penguin sleeps
(b-run pingu) ;; the penguin runs
```

Better.

We placed them into two distinct categories that also implements `bird`, which is `flightless-bird` and `flying-bird`. Both can eat and sleep, but only the one that implements `flightless-bird` have to also implement `b-run` method. This is the same for the client that implements `flying-bird`. It just need to care for extra `b-fly` method. No need to implement methods that a certain client won't need. Hurray!

D: Dependency Inversion

- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
- **Abstractions should not depend upon details. Details should depend upon asbtractions.**

Dependency Inversion Principle encourages us to create higher level modules with its complex logic in such a way to be reusable and unaffected by any change from the lower level modules in our application. To achieve this kind of behavior in our apps, we introduce abstraction which decouples higher from lower level modules.

Wait! What are these *low-level* and *high-level* modules again?

Well, low-level modules are more specific to the individual components focusing os smaller details of the application. This *low-level* module should be used in *high-level* modules in the application.

While *high-level* modules are more abstract and general in nature. They handle *low-level* modules and decide the logic for what goes where.

Think about a computer CPU (high-level) handling bunch of hardware inputs (low-level) including keyboard and mouse inputs.

See DI violation below.

Bad

```
(defclass printer ()
  ((data-type
    :initarg :data-type
    :reader get-data-type)))

(defmethod print-epub ((printer printer))
  (let ((e (make-instance 'epub-formatter)))
    (process e (get-data-type printer))))

(defmethod print-mobi ((printer printer))
  (let ((m (make-instance 'mobi-formatter)))
    (process m (get-data-type printer))))

(defclass epub-formatter ()
  nil)

(defmethod process ((epub-formatter epub-formatter) data-type)
  (format t "~a~%data-type: ~a~%"
          "epub formatter's process logic goes here"
          data-type))

(defclass mobi-formatter ()
  nil)

(defmethod process ((mobi-formatter mobi-formatter) data-type)
  (format t "~a~%data-type: ~a~%"
          "mobi formatter's process logic goes here"
          data-type))

(defparameter epub-book (make-instance 'printer :data-type "epubs"))
(defparameter mobi-book (make-instance 'printer :data-type "mobis"))

(print-epub epub-book)
;; epub formatter's process logic goes here
;; data-type: epubs

(print-mobi mobi-book)
;; mobi formatter's process logic goes here
;; data-type: mobis
```

`printer` (high-level) class had to depend on `print-epub` and `print-mobi` which are both low-level modules. This breaks the " *High level modules should not depend upon low level modules. Both should depend upon abstractions.*" keypoint.

Let's fix it below.

Good

```
(defclass printer ()
  ((data-type
    :initarg :data-type
    :reader get-data-type)))

(defmethod prints ((printer printer) formatter)
  (let ((f (make-instance formatter)))
    (process f (get-data-type printer))))

(defclass epub-formatter ()
  nil)

(defmethod process ((epub-formatter epub-formatter) data-type)
  (format t "~a~%data-type: ~a~%"
          "epub formatter's process logic goes here"
          data-type))

(defclass mobi-formatter ()
  nil)

(defmethod process ((mobi-formatter mobi-formatter) data-type)
  (format t "~a~%data-type: ~a~%"
          "mobi formatter's process logic goes here"
          data-type))


(defparameter epub-book (make-instance 'printer :data-type "epubs"))
(defparameter mobi-book (make-instance 'printer :data-type "mobis"))

(prints epub-book 'epub-formatter)
;; epub formatter's process logic goes here
;; data-type: epubs

(prints mobi-book 'mobi-formatter)
;; mobi formatter's process logic goes here
;; data-type: mobis
```

We have moved both `print-epub` and `print-mobi` into separate classes. Now they can do their own things, and whenever we need to print an ebook, we are going to pass either `epub-formatter` or `mobi-formatter` to `printer` constructor, then call `prints` to execute the method based on the class we passed.

Now, our classes are not tightly coupled with the lower-tier objects and we can easily reuse the logic from the high-tier modules.